Internal Assessment Test 2 – November 2020

Sub:	Software Architecture & Design Patterns	Sub Code:	17IS72

1a. List out & explain four essential elements of design pattern with small talk MVC example.

Ans.

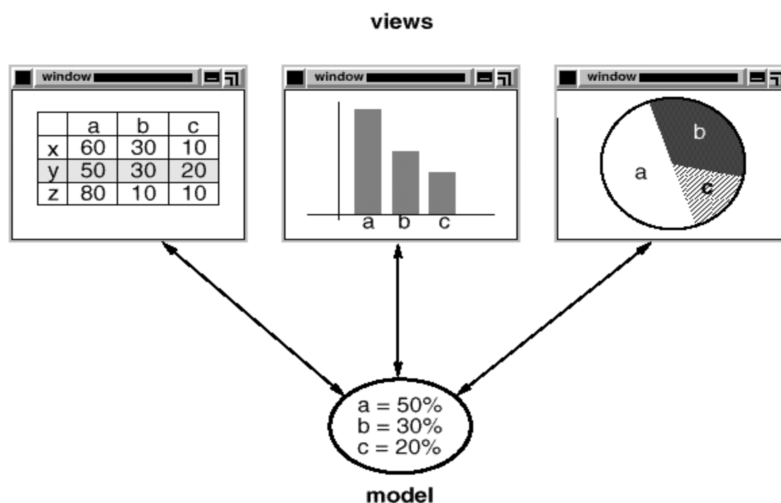
- Christopher Alexander says, "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice".
- Four essential elements of a pattern::

1. The pattern name is a handle we can use to describe a design problem, its solutions, and consequences in a word or two.
2. The problem describes when to apply the pattern.
3. The solution describes the elements that make up the design, their relationships, responsibilities, and collaborations
4. The consequences are the results and trade-offs of applying the pattern.

- The design patterns are *descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context*.

Design Patterns in Smalltalk MVC

- Model is the application object
- View is the model's screen presentation
- Controller defines the way the user interface reacts to user input.
- MVC decouples views and models by establishing a subscribe/notify protocol between them.



- The design is applicable to a more general problem: decoupling objects so that changes to one can affect any number of others without requiring the changed object to know details of the others. (Observer design pattern)
- MVC supports nested views with the CompositeView class, a subclass of View. (Composite design pattern).
- MVC also lets you change the way a view responds to user input without changing its visual presentation by encapsulating the response mechanism in a Controller object. (Strategy design pattern)
- MVC uses Factory Method design pattern to specify the default controller class for a view and Decorator design pattern to add scrolling to a view.

1b. List out the templates used in describing design pattern.
Ans.

- Pattern Name and Classification
- Intent: What does the design pattern do? What is its rationale and intent? What particular design issue or problem does it address?
- Also Known As
- Motivation: A scenario that illustrates a design problem and how the class and object structures in the pattern solve the problem.
- Applicability: What are the situations in which the design pattern can be applied?
- Structure: A graphical representation of the classes in the pattern using a notation based on OMT or UML.
- Participants: The classes and/or objects participating in the design pattern and their responsibilities.
- Collaborations: How the participants collaborate to carry out their responsibilities.
- Consequences: How does the pattern support its objectives? What are the trade-offs and results of using the pattern?
- Implementation: What should you be aware of when implementing the pattern? Are there language-specific issues?
- Sample Code: Code fragments that illustrate how you might implement the pattern in particular object-oriented programming languages.
- Known Uses: Examples of the pattern found in real systems.
- Related Patterns

The Catalog of Design Patterns:

- Abstract Factory provides an interface for creating families of related or dependent objects without specifying their concrete classes.
- Adapter converts the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- Bridge decouples an abstraction from its implementation so that the two can vary independently.
- Builder separates the construction of a complex object from its representation so that the same construction process can create different representations.
- Chain of Responsibility avoids coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

- Command encapsulates a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
- Composite composes objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
- Decorator attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- Facade provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
- Factory Method defines an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- Flyweight uses sharing to support large numbers of fine-grained objects efficiently.
- Interpreter,given a language, defines a represention for its grammar along with an interpreter that uses the representation to interpret sentences in the language.Iterator provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- Mediator defines an object that encapsulates how a set of objects interact.
- Memento, without violating encapsulation, captures and externalizes an object's internal state so that the object can be restored to this state later.
- Observer defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- Prototype specifies the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
- Proxy provides a surrogate or placeholder for another object to control access to it.
- Singleton ensures a class only has one instance, and provide a global point of access to it.
- State allows an object to alter its behavior when its internal state changes. The object will appear to change its class.
- Strategy defines a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
- Template Method defines the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
- Visitor represents an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Creational	**Structural**		**Behavioral**	
Scope	**Class**	Factory Method	Adapter	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Flyweight Observer State Strategy Visitor

1c. Name any four design patterns available in catalog of design pattern
Ans.

* Design patterns are classified by two criteria:
  (1) purpose - Reflects what a pattern does
  - They can be:
    a) creational - concern the process of object creation
    b) Structural - Deal with the composition of classes or objects.
    c) Behavioral - characterize the ways in which classes or objects interact and distribute responsibility
  (2) Scope - Specifies whether the pattern applies primarily to classes or to objects
  - class patterns - deal with relationships between classes and their subclasses which are established through inheritance; so they are static - fixed at compile time.
  - object patterns - deal with object relationships, which can be changed at run-time and are more dynamic.
* Creational class pattern & object pattern—
  Creational class patterns defer some part of object creation to subclasses, while creational object patterns ~~use inheritance to compose~~ defer it to another object.
* Structural class patterns & object patterns -
  Structural class patterns use inheritance to compose classes, while the structural object patterns describe ways to assemble objects.

2a. Name the several approaches to find the design pattern that's rights for your problem.
Ans.

- Design patterns solve many of the day-to-day problems object-oriented designers face, and in many different ways.
- problems & solutions using design patterns are as follows:

i) Finding Appropriate Objects:-

- Object-oriented programs (OOp) are made up of objects — which package both data and the procedures that operate on the data
- Decomposing a system into objects is the hard part in object-oriented design.
- Many objects in a design come from the analysis model. But, object-oriented designs often end up with classes that have no counterparts in the real world.
- Strict modeling of the real world leads to a system that reflects today's realities but not necessarily tomorrow's
- The abstractions that emerge during design are key to making a design flexible
- Design patterns identify less-obvious abstractions.

2) Determining Object Granularity:-

- Objects can vary tremendously in size and number
- Design patterns address this issue

eg:- Facade pattern — describes how to represent subsystems as objects

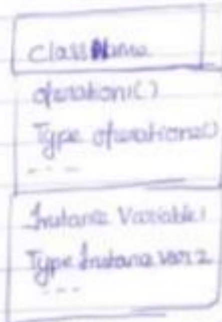
Flyweight pattern - describes how to support huge numbers of objects at the finest granularities

3) Specifying object Interfaces:-

- Signature:- Every operation declared by an object specifies the operation's name, its parameters and return value. This is known as operation's signature
- Interface:- The set of all signatures defined by an object's operations is called the interface to the object.
- Object's interface characterizes the complete set of requests that can be sent to the object. Any request that matches a signature in the object's interface may be sent to the object
- Type:- Name used to denote a particular interface. Subtype & supertype - A subtype interface contains the interface of its supertype.
- Dynamic binding:- The run-time association of a request to an object and one of its operations is known as dynamic binding.
- polymorphism - It simplifies the definitions of clients, decouples objects from each other, and lets them vary their relationships to each other at run-time.
- Design patterns for interfaces-
  - They define interfaces by identifying their key elements and the kinds of data that get sent across an interface
  - They are specify relationships b/w interfaces
  - Eg:- i) Memento pattern - describes how to encapsulate & save the internal state of an object so that the object can be restored to that state later

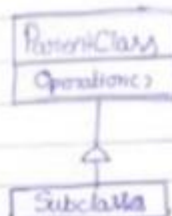
4) Specifying Object Implementation:

- An object's implementation is defined by its class.
- The class specifies the object's internal data and defines the operations the object can perform.



- objects are created by instantiating a class i.e., an object is an instance of the class. This allocates storage for the objects internal data and associates the operations with these data.

- Class inheritance — Defining new classes (subclass) in terms of existing classes (Parent class).
  - This includes the definitions of all the data & operations that the parent class defines.
  - objects that are instances of the subclasses will contain all data defined by the subclass and its parent classes and also perform all operations defined by this subclass and its parents.



2b. Define object-oriented development. Name the various key concepts of OOD.

Ans.

An object-oriented system is made up of interacting objects that maintain their own local state and provide operations on that state. There is a general acceptance that an object is an encapsulation of information. The representation of the state is private and cannot be accessed directly from outside the object.

Object-oriented programming has four basic concepts: encapsulation, abstraction, inheritance and polymorphism. Even if these concepts seem incredibly complex, understanding the general framework of how they work will help you understand the basics of a computer program. Here are the four basic theories and what they entail:

- Encapsulation

- Abstraction
- Inheritance
- Polymorphism

Encapsulation

The different objects inside of each program will try to communicate with each other automatically. If a programmer wants to stop objects from interacting with each other, they need to be encapsulated in individual classes. Through the process of encapsulation, classes cannot change or interact with the specific variables and functions of an object.

Just like a pill "encapsulates" or contains the medication inside of its coating, the principle of encapsulation works in a digital way to form a protective barrier around the information that separates it from the rest of the code. Programmers can replicate this object throughout different parts of the program or other programs.

Abstraction

Abstraction is like an extension of encapsulation because it hides certain properties and methods from the outside code to make the interface of the objects simpler. Programmers use abstraction for several beneficial reasons. Overall, abstraction helps isolate the impact of changes made to the code so that if something goes wrong, the change will only affect the variables shown and not the outside code.

Inheritance

Using this concept, programmers can extend the functionality of the code's existing classes to eliminate repetitive code. For instance, elements of HTML code that include a text box, select field and checkbox have certain properties in common with specific methods.

Instead of redefining the properties and methods for every type of HTML element, you can define them once in a generic object. Naming that object something like "HTMLElement" will cause other objects to inherit its properties and methods so you can reduce unnecessary code.

The main object is the superclass and all objects that follow it are subclasses. Subclasses can have separate elements while adding what they need from the superclass.

Polymorphism

This technique meaning "many forms or shapes" allows programmers to render multiple HTML elements depending on the type of object. This concept allows programmers to redefine the way something works by changing how it is done or by changing the parts in which it is done. Terms of polymorphism are called overriding and overloading.

2c. Explain the benefits & drawback of the paradigm in OOD.
Ans.

Object-oriented programming is a well-adopted programming style that uses interacting objects to model and solve complex programming tasks. Two examples of popular object-

oriented programming languages are Java and C++. Some other well-known object-oriented programming languages include Objective C, Perl, Python, Javascript, Simula, Modula, Ada, Smalltalk, and the Common Lisp Object Standard.

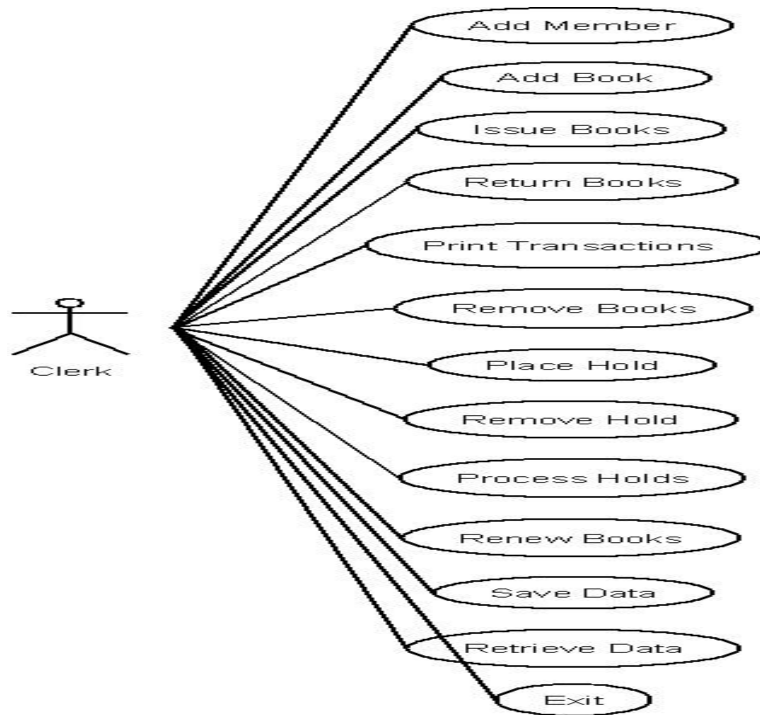
Some of the advantages of object-oriented programming include:

1. Improved software-development productivity: Object-oriented programming is modular, as it provides separation of duties in object-based program development. It is also extensible, as objects can be extended to include new attributes and behaviors. Objects can also be reused within an across applications. Because of these three factors – modularity, extensibility, and reusability – object-oriented programming provides improved software-development productivity over traditional procedure-based programming techniques.
2. Improved software maintainability: For the reasons mentioned above, objectoriented software is also easier to maintain. Since the design is modular, part of the system can be updated in case of issues without a need to make large-scale changes.
3. Faster development: Reuse enables faster development. Object-oriented programming languages come with rich libraries of objects, and code developed during projects is also reusable in future projects.
4. Lower cost of development: The reuse of software also lowers the cost of development. Typically, more effort is put into the object-oriented analysis and design, which lowers the overall cost of development.
5. Higher-quality software: Faster development of software and lower cost of development allows more time and resources to be used in the verification of the software. Although quality is dependent upon the experience of the teams, object-oriented programming tends to result in higher-quality software.

Some of the disadvantages of object-oriented programming include:

1. Steep learning curve: The thought process involved in object-oriented programming may not be natural for some people, and it can take time to get used to it. It is complex to create programs based on interaction of objects. Some of the key programming techniques, such as inheritance and polymorphism, can be challenging to comprehend initially.
2. Larger program size: Object-oriented programs typically involve more lines of code than procedural programs.
3. Slower programs: Object-oriented programs are typically slower than procedure-based programs, as they typically require more instructions to be executed.
4. Not suitable for all types of problems: There are problems that lend themselves well to functional-programming style, logic-programming style, or procedure-based programming style, and applying object-oriented programming in those situations will not result in efficient programs.

3a. List out the business process of the library system.
Ans.



Use-Case for Returning Books

Actions performed by the actor	Responses from the system
1. The member arrives at the return counter with a set of books and gives the clerk the books.	
2. The clerk issues a request to return books.	
	3. The system asks for the identifier of the book.
4. The clerk enters the book identifier.	
	5. If the identifier is valid, the system marks that the book has been returned and informs the clerk if there is a hold placed on the book; otherwise it notifies the clerk that the identifier is not valid. It then asks if the clerk wants to process the return of another book.
6. The clerk notes whether there is a hold on the book and answers in the affirmative or in the negative. If there is a hold, the clerk sets the book aside.	
	7. If the answer is in the affirmative, the system goes to Step 3. Otherwise, it exits.

Use-Case for Issuing Books

Actions performed by the actor	Responses from the system
1. The member arrives at the check-out counter with a set of books and supplies the clerk with his/her identification number.	
2. Clerk issues a request to check out books.	
	3. The system asks for the user id.
4. Clerk inputs the user ID to the system.	
	5. If the ID is valid, the system asks for the ID of the book; otherwise it prints an appropriate message and exits the use-case.
6. The clerk inputs the identifier of a book that the user wants to check out.	
	7. If the ID is valid and the book is issuable to the member, the book is recorded as having been issued to the member; Member is recorded as having possession of the book, a due-date is generated and details of the transaction are displayed. If the book is not issuable, a suitable error message is displayed. The system asks if there are more books.
8. The clerk stamps the due-date, prints out the transaction (if needed) and replies positively or negatively.	
	9. If there are more books for checking out, the system goes back to Step 5; otherwise it exits.
10. The clerk stamps the due date and gives the user the books checked out. The customer leaves the counter.	

3b. Define business rule. List out the rules of the library system.
Ans.
business rules define specific instructions or constraints on how certain day-to-day actions should be performed. For example, business rules can include: A decision-making approval structure for invoice processing where only certain managers can sign off on invoices totaling a specific amount.

lasses of Library Management System :

- Library Management System class –
  It manages all operations of Library Management System. It is central part of organization for which software is being designed.
- User Class –
  It manages all operations of user.
- Librarian Class – It manages all operations of Librarian.
- Book Class –
  It manages all operations of books. It is basic building block of system.
- Account Class –
  It manages all operations of account.
- Library database Class –
  It manages all operations of library database.
- Staff Class –
  It manages all operations of staff.
- Student Class –
  It manages all operations of student.

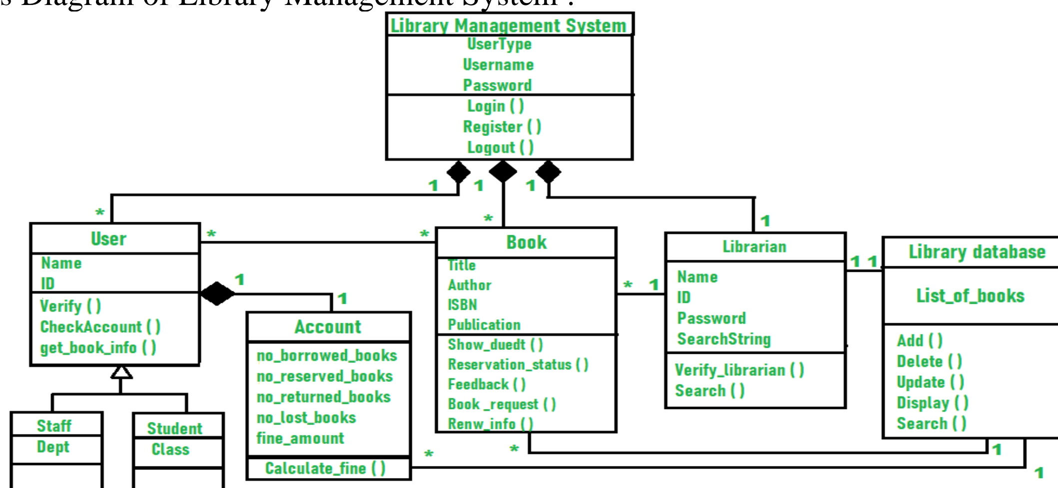
Attributes of Library Management System :

- Library Management System Attributes –
  UserType, Username, Password
- User Attributes –
  Name, Id
- Librarian Attributes –
  Name, Id, Password, SearchString
- Book Attributes –
  Title, Author, ISBN, Publication
- Account Attributes –
  no_borrowed_books, no_reserved_books, no_returned_books, no_lost_books fine_amount
- Library database Attributes –
  List_of_books
- Staff Class Attributes –
  Dept
- Student Class Attributes –
  Class

Methods of Library Management System :

- Library Management System Methods –
  Login(), Register(), Logout()
- User Methods –
  Verify(), CheckAccount(), get_book_info()
- Librarian Methods –
  Verify_librarian(), Search()
- Book Methods –
  Show_duedt(), Reservation_status(), Feedback(), Book_request(), Renew_info()
- Account Methods –
  Calculate_fine()
- Library database Methods –
  Add(), Delete(), Update(), Display(), Search()

Class Diagram of Library Management System :



3c. Explain how do business rules relate to use cases with its four categories

Buried Business Rules

Name: ATM withdrawal

Description: Actor wants to withdraw money from account

Actors:
- Customer
- Customer representative

Use Case Relationships

Pre-Conditions

Basic Flow
1. Use Case begins when Actor presents ATM card
2. Validate customer: is customer in our db, does PIN match
3. Determine if customer has sufficient funds: compare amount requested to total amount in customer account
4. Deduct amount from account
5. Distribute money
6. Distribute receipt

Post Conditions

Alternate Flows

Notes

Dangling Business Rules

Name: ATM withdrawal

Description: Actor wants to withdraw money from account

Actors:
- Customer
- Customer representative

Use Case Relationships

Pre-Conditions

Basic Flow
1. Use Case begins when Actor presents ATM card
2. Validate customer
3. Determine customer has sufficient funds
4. Deduct amount from account
5. Distribute money
6. Distribute receipt

Post Conditions

Alternate Flows

Notes

Business Rules:
- Customer must be valid.
- Customer pin must match
- Requested amount must be less than amount in account.

Positioned Business Rules

Name: ATM withdrawal

Description: Actor wants to withdraw money from account

Actors:
- Customer
- Customer representative

Use Case Relationships

Pre-Conditions

Basic Flow
1. Use case begins when Actor presents ATM card
2. Validate customer:
   Business Rule: Customer must be Valid.
   Business Rule: Customer pin must match.
3. Determine if Customer has sufficient funds:
   Business Rule: Requested amount must be less than total amount in account
4. Deduct amount from account
5. Distribute money
6. Use case ends with distribute receipt

Post Conditions

4a. List out the guidelines to remember when writing use case.
Ans.

What Use Cases Include	What Use Cases Do NOT Include
• Who is using the website • What the user want to do • The user's goal • The steps the user takes to accomplish a particular task • How the website should respond to an action	• Implementation-specific language • Details about the user interfaces or screens.

Elements of a Use Case

Depending on how in depth and complex you want or need to get, use cases describe a combination of the following elements:

- Actor – anyone or anything that performs a behavior (who is using the system)
- Stakeholder – someone or something with vested interests in the behavior of the system under discussion (SUD)
- Primary Actor – stakeholder who initiates an interaction with the system to achieve a goal

- Preconditions – what must be true or happen before and after the use case runs.
- Triggers – this is the event that causes the use case to be initiated.
- Main success scenarios [Basic Flow] – use case in which nothing goes wrong.
- Alternative paths [Alternative Flow] – these paths are a variation on the main theme. These exceptions are what happen when things go wrong at the system level.

4b. What is domain analysis? Explain the thumb rules and caveats come on handy.
Ans.
Domain analysis is the process by which a software engineer learns background information. He or she has to learn sufficient information so as to be able to understand the problem and make good decisions during requirements analysis and other stages of the software engineering process. The word 'domain' in this case means the general field of business or technology in which the customers expect to be using the software.
Some domains might be very broad, such as 'airline reservations', 'medical diagnosis', and 'financial analysis'. Others are narrower, such as 'the manufacturing of paint' or 'scheduling meetings'. People who work in a domain and who have a deep knowledge of it (or part of it), are called *domain experts*. Many of these people may become customers or users.

To perform domain analysis, you gather information from whatever sources of information are available: these include the domain experts; any books about the domain; any existing software and its documentation, and any other documents he or she can find. The interviewing, brainstorming and use case analysis techniques discussed later in this chapter can help with domain analysis. Object oriented modelling, discussed in the next chapter, can also be of assistance.

As a software engineer, you are not expected to become an expert in the domain; nevertheless, domain analysis can involve considerable work. The following benefits will make this work worthwhile:

- Faster development: You will be able to communicate with the stakeholders more effectively, hence you will be able to establish requirements more rapidly. Having performed domain analysis will help you to focus on the most important issues.
- Better system: Knowing the subtleties of the domain will help ensure that the solutions you adopt will more effectively solve the customer's problem. You will make fewer mistakes, and will know which procedures and standards to follow. The analysis will give you a global picture of the domain of application; this will lead to better abstractions and hence improved designs.
- Anticipation of extensions: Armed with domain knowledge, you will obtain insights into emerging trends and you will notice opportunities for future development. This will allow you to build a more adaptable system.

It is useful to write a summary of the information found during domain analysis. The process of organizing and writing this summary can help you gain a better grasp of the knowledge; the resulting document can help educate other software engineers who join the team later. We suggest that a domain analysis document should be divided into sections such as the following:

A. Introduction: Name the domain, and give the motivation for performing the analysis. The motivation normally is that you are preparing to solve a particular problem by development or extension of a software system.

B. Glossary: Describe the meanings of all terms used in the domain that are either not part of everyday language or else have special meanings. You must master this terminology if you want to be able to communicate with your customers and users. The terminology will appear in the user interface of the software as well as in the documentation. You may be able to refer to an existing glossary in some other document, rather than writing a new glossary. The section is best placed at the start of the domain analysis document so you can subsequently can use the defined terms.

C. General knowledge about the domain: Summarize important facts or rules that are widely known by the domain experts and which would normally be learned as part of their education. Such knowledge includes scientific principles, business processes, analysis techniques, and how any technology works. This is an excellent place to use diagrams; however, where possible point the reader for details to any readily accessible books or other documents. This general knowledge will help you acquire an understanding of the data you may have to process and computations you may have to perform.

D. Customers and users: Describe who will or might buy the software, and in what industrial sectors they operate. Also, describe the other people who work in the domain, even peripherally. Mention their background and attitude as well as how they fit into the organization chart, and relate to each other. These people may become users.

E. The environment: Describe the equipment and systems used. The new system or extensions will have to work in the context of this environment.

F. Tasks and procedures currently performed: Make a list of what the various people do as they go about their work. It is important to understand both the procedures people are supposed to follow as well as the shortcuts they tend to take. For example, if people are supposed to enter certain information on a form, but rarely do, this suggests the information is not useful. Tasks listed in this section may be candidates for automation.

G. Competing software: Describe what software is available to assist the users and customers, including software that is already in use, and software on the market. Discuss its advantages and disadvantages. This information suggests ideas for requirements, and highlights mistakes to avoid.

H. Similarities across domains and organizations: Understanding what is generic versus what is specific will help you to create software that might be more reusable or more widely marketable. Therefore, determine what distinguishes this domain and the customer's organization from others, as well as what they have in common.

4c. Compare business process modeling & use case modeling.
Ans.

A use-case model is a model of how different types of users interact with the system to solve a problem. As such, it describes the goals of the users, the interactions between the users and the system, and the required behavior of the system in satisfying these goals.

A use-case model consists of a number of model elements. The most important model elements are: use cases, actors and the relationships between them.

A use-case diagram is used to graphically depict a subset of the model to simplify communications. There will typically be several use-case diagrams associated with a given model, each showing a subset of the model elements relevant for a particular purpose. The same model element may be shown on several use-case diagrams, but each instance must be consistent. If tools are used to maintain the use-case model, this consistency constraint is

automated so that any changes to the model element (changing the name for example) will be automatically reflected on every use-case diagram that shows that element.

The use-case model may contain packages that are used to structure the model to simplify analysis, communications, navigation, development, maintenance and planning.

Much of the use-case model is in fact textual, with the text captured in the Use-Case Specifications that are associated with each use-case model element. These specifications describe the flow of events of the use case.

The use-case model serves as a unifying thread throughout system development. It is used as the primary specification of the functional requirements for the system, as the basis for analysis and design, as an input to iteration planning, as the basis of defining test cases and as the basis for user documentation

Basic model elements

The use-case model contains, as a minimum, the following basic model elements.

Actor

A model element representing each actor. Properties include the actors name and brief description.

Use Case

A model element representing each use case. Properties include the use case name and use case specification.

Associations

Associations are used to describe the relationships between actors and the use cases they participate in. This relationship is commonly known as a “communicates-association”.

- A business model is a company's core strategy for profitably doing business.
- Models generally include information like products or services the business plans to sell, target markets, and any anticipated expenses.
- The two levers of a business model are pricing and costs.
- When evaluating a business model as an investor, ask whether the idea makes sense and whether the numbers add up.

5a. Define structural pattern. List out consequences of adapter pattern.
Ans.
In Software Engineering, Structural Design Patterns are Design Patterns that ease the design by identifying a simple way to realize relationships between entities.

- Adapter
  Match interfaces of different classes
- Bridge
  Separates an object's interface from its implementation

- Composite
  A tree structure of simple and composite objects
- Decorator
  Add responsibilities to objects dynamically
- Facade
  A single class that represents an entire subsystem
- Flyweight
  A fine-grained instance used for efficient sharing
- Private Class Data
  Restricts accessor/mutator access
- Proxy
  An object representing another object

Adapter Design Pattern
Intent

- Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- Wrap an existing class with a new interface.
- Impedance match an old component to a new system

Problem
An "off the shelf" component offers compelling functionality that you would like to reuse, but its "view of the world" is not compatible with the philosophy and architecture of the system currently being developed.

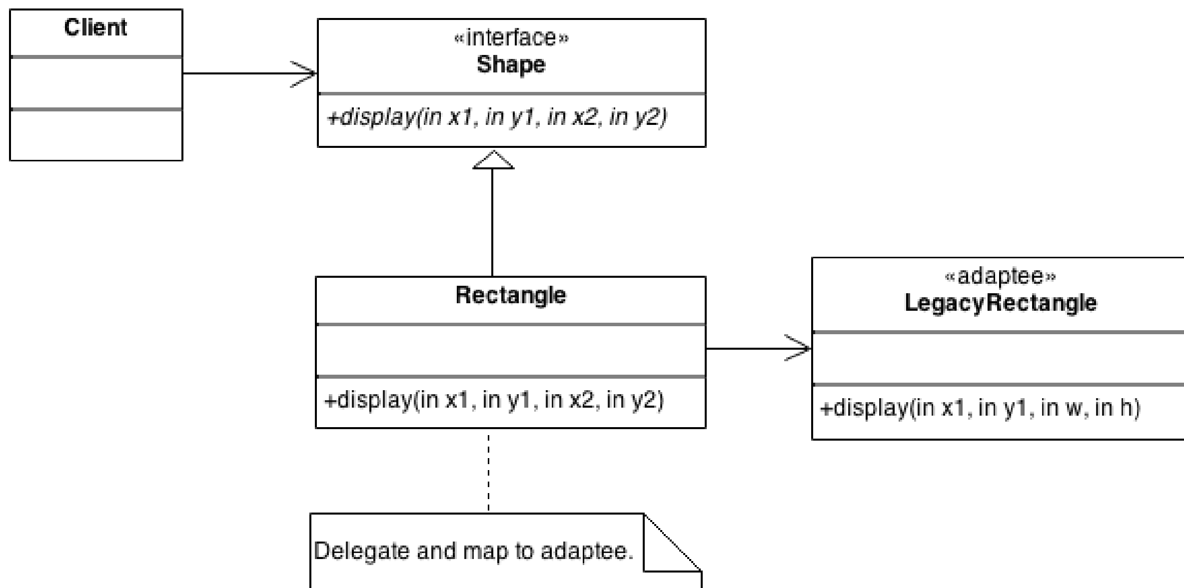
Discussion
Reuse has always been painful and elusive. One reason has been the tribulation of designing something new, while reusing something old. There is always something not quite right between the old and the new. It may be physical dimensions or misalignment. It may be timing or synchronization. It may be unfortunate assumptions or competing standards.

It is like the problem of inserting a new three-prong electrical plug in an old two-prong wall outlet – some kind of adapter or intermediary is necessary.

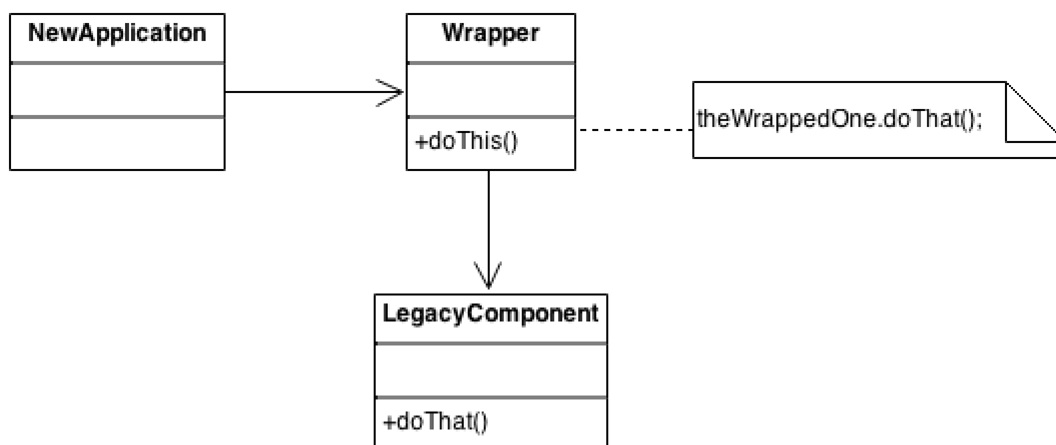
Adapter is about creating an intermediary abstraction that translates, or maps, the old component to the new system. Clients call methods on the Adapter object which redirects them into calls to the legacy component. This strategy can be implemented either with inheritance or with aggregation.

Adapter functions as a wrapper or modifier of an existing class. It provides a different or translated view of that class.

Structure
Below, a legacy Rectangle component's `display()` method expects to receive "x, y, w, h" parameters. But the client wants to pass "upper left x and y" and "lower right x and y". This incongruity can be reconciled by adding an additional level of indirection – i.e. an Adapter object.



The Adapter could also be thought of as a "wrapper".



5b. Describe the two important issues when implementing the fly weight pattern.
Ans.

lyweights may introduce run-time costs associated with transferring, finding, and/or computing extrinsic state, especially if it was formerly stored as intrinsic state. However, such costs are offset by space savings, which increase as more flyweights are shared.

Storage savings are a function of several factors:

- the reduction in the total number of instances that comes from sharing
- the amount of intrinsic state per object
- whether extrinsic state is computed or stored.

The more flyweights are shared, the greater the storage savings. The savings increase with the amount of shared state. The greatest savings occur when the objects use substantial quantities of both intrinsic and extrinsic state, and the extrinsic state can be computed rather than stored.

Then you save on storage in two ways: Sharing reduces the cost of intrinsic state, and you trade extrinsic state for computation time.

5c. Describe the implementation and sample code of adapter pattern.
Ans.

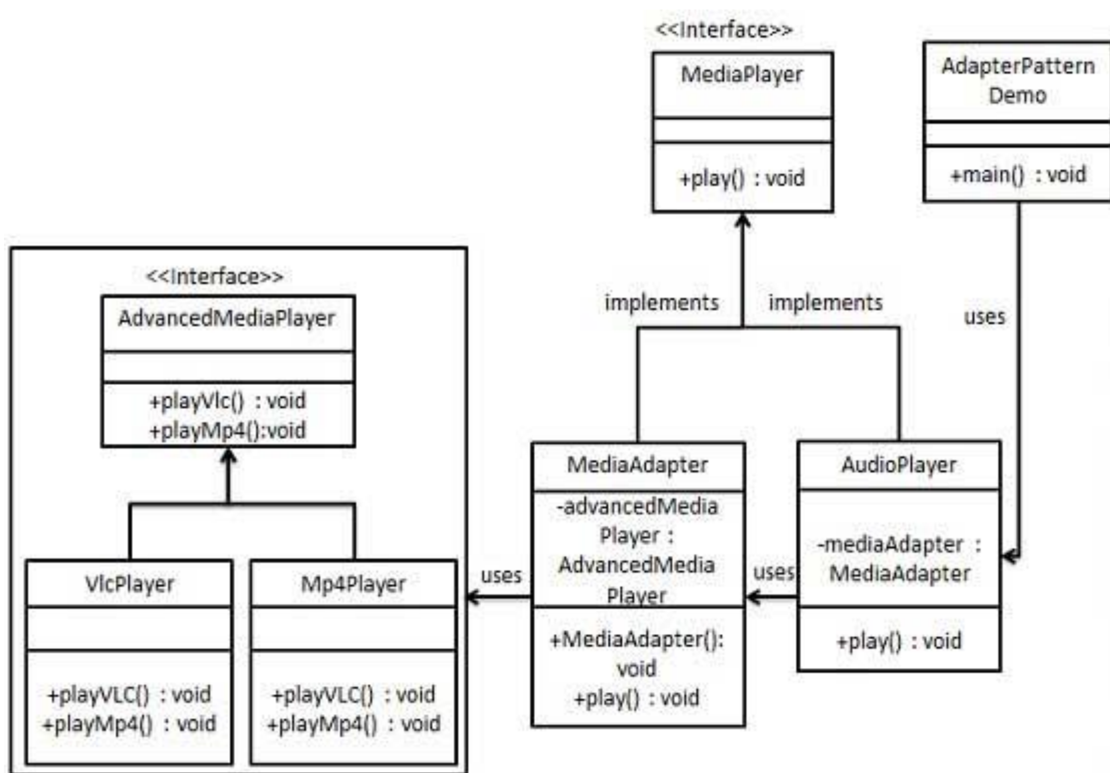
mplementation

We have a *MediaPlayer* interface and a concrete class *AudioPlayer* implementing the *MediaPlayer* interface. *AudioPlayer* can play mp3 format audio files by default.

We are having another interface *AdvancedMediaPlayer* and concrete classes implementing the *AdvancedMediaPlayer* interface. These classes can play vlc and mp4 format files.

We want to make *AudioPlayer* to play other formats as well. To attain this, we have created an adapter class *MediaAdapter* which implements the *MediaPlayer* interface and uses *AdvancedMediaPlayer* objects to play the required format.

AudioPlayer uses the adapter class *MediaAdapter* passing it the desired audio type without knowing the actual class which can play the desired format. *AdapterPatternDemo*, our demo class will use *AudioPlayer* class to play various formats.



Step 1

Create interfaces for Media Player and Advanced Media Player.

MediaPlayer.java

```
public interface MediaPlayer {
   public void play(String audioType, String fileName);
```

```
}
```

AdvancedMediaPlayer.java

```
public interface AdvancedMediaPlayer {
  public void playVlc(String fileName);
  public void playMp4(String fileName);
}
```

Step 2

Create concrete classes implementing the *AdvancedMediaPlayer* interface.

VlcPlayer.java

```
public class VlcPlayer implements AdvancedMediaPlayer{
  @Override
  public void playVlc(String fileName) {
    System.out.println("Playing vlc file. Name: "+ fileName);
  }

  @Override
  public void playMp4(String fileName) {
    //do nothing
  }
}
```

Mp4Player.java

```
public class Mp4Player implements AdvancedMediaPlayer{

  @Override
  public void playVlc(String fileName) {
    //do nothing
  }

  @Override
  public void playMp4(String fileName) {
    System.out.println("Playing mp4 file. Name: "+ fileName);
  }
}
```

Step 3

Create adapter class implementing the *MediaPlayer* interface.

MediaAdapter.java

```
public class MediaAdapter implements MediaPlayer {

  AdvancedMediaPlayer advancedMusicPlayer;

```

```
  public MediaAdapter(String audioType){

    if(audioType.equalsIgnoreCase("vlc") ){
      advancedMusicPlayer = new VlcPlayer();

    }else if (audioType.equalsIgnoreCase("mp4")){
      advancedMusicPlayer = new Mp4Player();
    }
  }

  @Override
  public void play(String audioType, String fileName) {

    if(audioType.equalsIgnoreCase("vlc")){
      advancedMusicPlayer.playVlc(fileName);
    }
    else if(audioType.equalsIgnoreCase("mp4")){
      advancedMusicPlayer.playMp4(fileName);
    }
  }
}
```

Step 4

Create concrete class implementing the *MediaPlayer* interface.

AudioPlayer.java

```
public class AudioPlayer implements MediaPlayer {
  MediaAdapter mediaAdapter;

  @Override
  public void play(String audioType, String fileName) {

    //inbuilt support to play mp3 music files
    if(audioType.equalsIgnoreCase("mp3")){
      System.out.println("Playing mp3 file. Name: " + fileName);
    }

    //mediaAdapter is providing support to play other file formats
    else if(audioType.equalsIgnoreCase("vlc") || audioType.equalsIgnoreCase("mp4")){
      mediaAdapter = new MediaAdapter(audioType);
      mediaAdapter.play(audioType, fileName);
    }

    else{
      System.out.println("Invalid media. " + audioType + " format not supported");
    }
  }
}
```

Step 5

Use the AudioPlayer to play different types of audio formats.

AdapterPatternDemo.java

```
public class AdapterPatternDemo {
   public static void main(String[] args) {
      AudioPlayer audioPlayer = new AudioPlayer();

      audioPlayer.play("mp3", "beyond the horizon.mp3");
      audioPlayer.play("mp4", "alone.mp4");
      audioPlayer.play("vlc", "far far away.vlc");
      audioPlayer.play("avi", "mind me.avi");
   }
}
```

Step 6

Verify the output.

```
Playing mp3 file. Name: beyond the horizon.mp3
Playing mp4 file. Name: alone.mp4
Playing vlc file. Name: far far away.vlc
Invalid media. avi format not supported
```

6a. Define the intent of Bridge pattern. Mention the consequences of bridge pattern.
Ans.
Intent

- Decouple an abstraction from its implementation so that the two can vary independently.
- Publish interface in an inheritance hierarchy, and bury implementation in its own inheritance hierarchy.
- Beyond encapsulation, to insulation

Use the Bridge pattern when:

- you want run-time binding of the implementation,
- you have a proliferation of classes resulting from a coupled interface and numerous implementations,
- you want to share an implementation among multiple objects,
- you need to map orthogonal class hierarchies.

Consequences include:

- decoupling the object's interface,
- improved extensibility (you can extend (i.e. subclass) the abstraction and implementation hierarchies independently),
- hiding details from clients.

Bridge is a synonym for the "handle/body" idiom. This is a design mechanism that encapsulates an implementation class inside of an interface class. The former is the body, and the latter is the handle. The handle is viewed by the user as the actual class, but the work is done in the body. "The handle/body class idiom may be used to decompose a complex abstraction into smaller, more manageable classes. The idiom may reflect the sharing of a single resource by multiple classes that control access to it (e.g. reference counting)."

6b. What is decorator pattern? Explain with neat sketch various participants of decorator pattern.
Ans.

Decorator pattern allows a user to add new functionality to an existing object without altering its structure. This type of design pattern comes under structural pattern as this pattern acts as a wrapper to existing class.

This pattern creates a decorator class which wraps the original class and provides additional functionality keeping class methods signature intact.

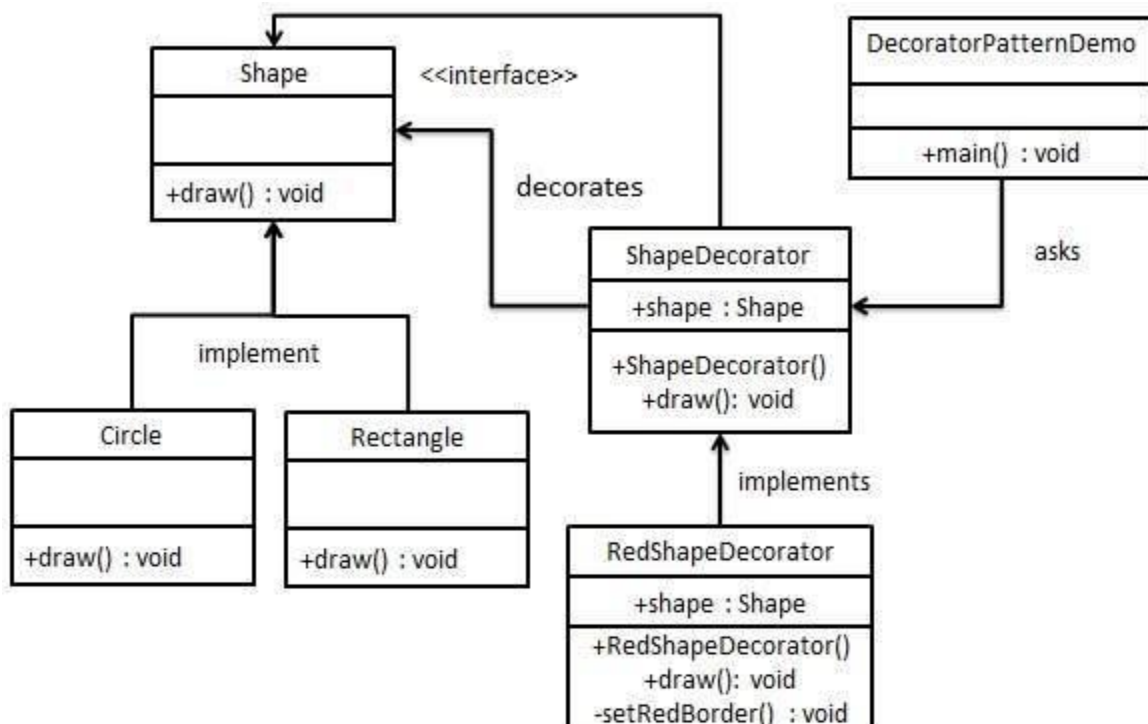
We are demonstrating the use of decorator pattern via following example in which we will decorate a shape with some color without alter shape class.

Implementation

We're going to create a *Shape* interface and concrete classes implementing the *Shape* interface. We will then create an abstract decorator class *ShapeDecorator* implementing the *Shape* interface and having *Shape* object as its instance variable.

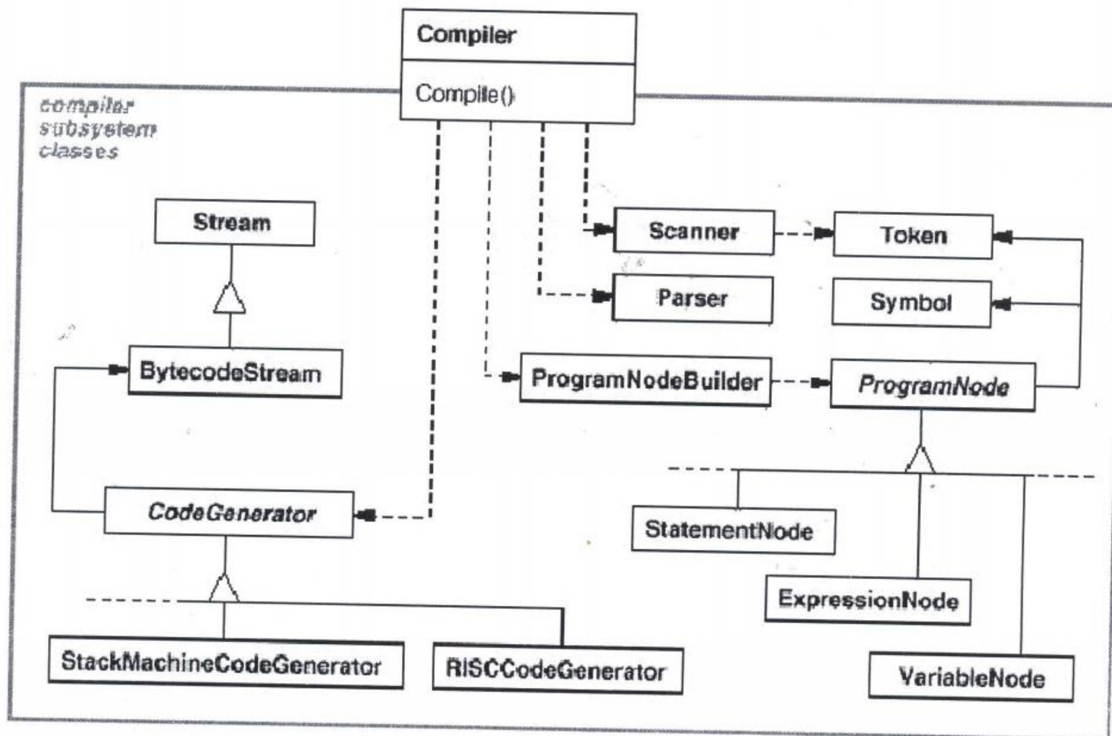
RedShapeDecorator is concrete class implementing *ShapeDecorator*.

DecoratorPatternDemo, our demo class will use *RedShapeDecorator* to decorate *Shape* objects.



6c. Explain with necessary diagram how compiler façade makes life easier for most programmers.

Ans.

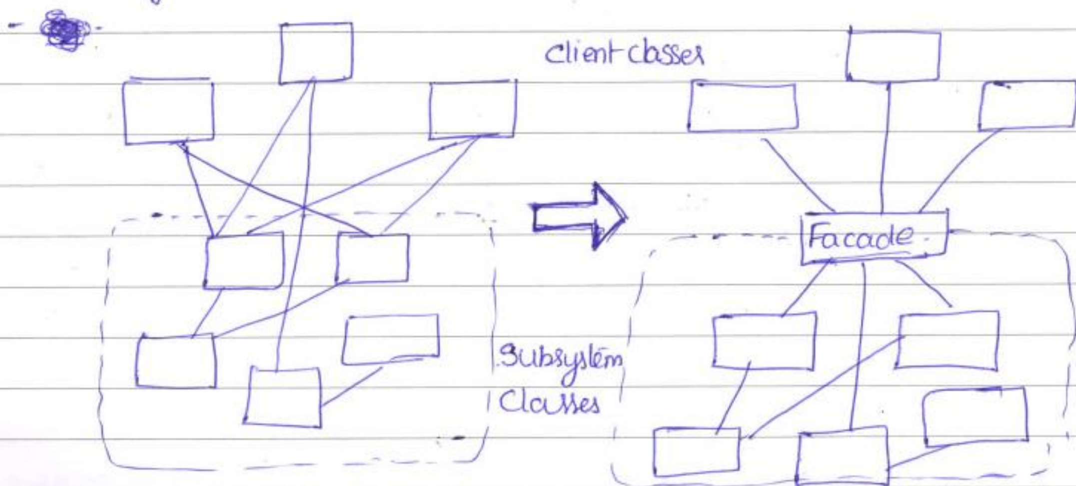


Compiler
Compile()
compiler subsystem classes
Stream
BytecodeStream
CodeGenerator
StackMachineCodeGenerator
RISCCodeGenerator
Scanner
Token
Parser
Symbol
ProgramNodeBuilder
ProgramNode
StatementNode
ExpressionNode
VariableNode

* Motivation:

- Structuring a system into subsystems helps reduce complexity.
- Clients of a subsystem may need to interact with a number of subsystem classes for their needs. This leads to a high degree of coupling b/w the client objects and the subsystem.
- Facade pattern provides a higher level, simplified interface for a subsystem resulting in reduced complexity & dependency.
- Clients interact with Facade object to deal with the subsystem instead of interacting directly with subsystem classes.

-



- Eg:- A programming environment that gives applications access to its compiler subsystem.
  Subsystem classes — Parser, Scanner, ProgramNode,

• Some specialized applications might need to access these classes directly. But, most clients of a compiler merely want only to compile some code. And, for them, the low-level interfaces in the compiler subsystem only complicate their task.

• To provide a higher-level interface that can shield clients from these classes, the compiler subsystem also includes a Compiler class, which defines a unified interface to the compiler's functionality. This is a Facade.

7a. Mention & explain benefits of MVC pattern.

Ans.

Advantages Of Using MVC Framework

1. Faster Development Process:

MVC supports rapid and parallel development. If an MVC model is used to develop any particular web application then it is possible that one programmer can work on the view while the other can work on the controller to create the business logic of the web application. Hence this way, the application developed using the MVC model can be completed three times faster than applications that are developed using other development patterns.

2. Ability To Provide Multiple Views:

In the MVC Model, you can create multiple views for a model. Today, there is an increasing demand for new ways to access your application and for that MVC development is certainly a great solution. Moreover, in this method, Code duplication is very limited because it separates data and business logic from the display.

3. Support For Asynchronous Technique:

The MVC architecture can also integrate with the JavaScript Framework. This means that MVC applications can be made to work even with PDF files, site-specific browsers, and also with desktop widgets. MVC also supports an asynchronous technique, which helps developers to develop an application that loads very fast.

4. The Modification Does Not Affect The Entire Model:

For any web application, the user interface tends to change more frequently than even the business rules of the .net development company. It is obvious that you make frequent changes in your web application like changing colors, fonts, screen layouts, and adding new device support for mobile phones or tablets. Moreover, Adding a new type of view are very easy in

the MVC pattern because the Model part does not depend on the views part. Therefore, any changes in the Model will not affect the entire architecture.

5. MVC Model Returns The Data Without Formatting:

MVC pattern returns data without applying any formatting. Hence, the same components can be used and called for use with any interface. For example, any kind of data can be formatted with HTML, but it could also be formatted with Macromedia Flash or Dream viewer.

6. SEO Friendly Development Platform:

MVC platform supports the development of SEO friendly web pages or web applications. Using this platform, it is very easy to develop SEO-friendly URLs to generate more visits from a specific application. This development architecture is commonly used in Test-Driven Development applications. Moreover, Scripting languages like JavaScript and jQuery can be integrated with MVC to develop feature-rich web applications.

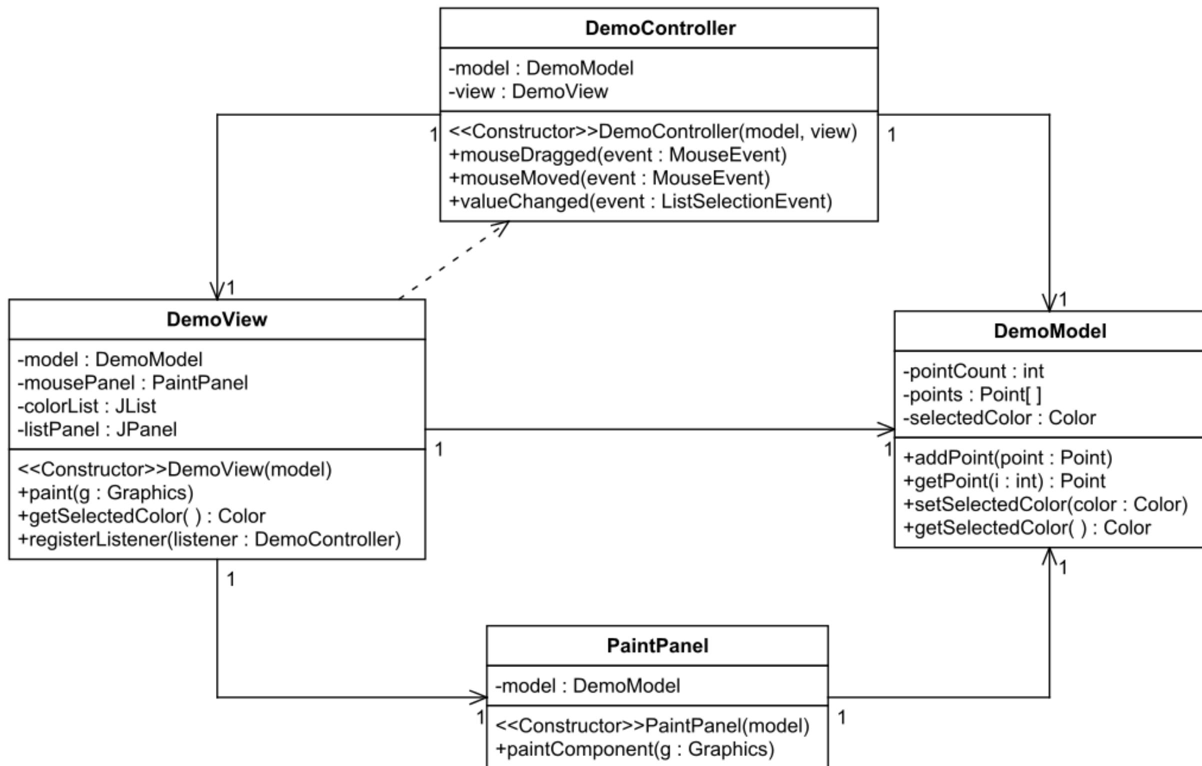
Thus, the MVC design pattern is surely a great approach to building software applications. The MVC framework is easy to implement as it offers above given numerous advantages. Projects that are developed with the help of the MVC model can be easily developed with lesser expenditure and within less time too. Above all, its power to manage multiple views makes MVC the best architecture pattern for developing web applications.

7b. With suitable use case tables explain analyzing a simple drawing program of MVC pattern

Ans.

The DemoMVC class holds the main method that starts the application. DemoModel implements the model part of the application. DemoView and PaintPanel implement the view part of the application. DemoController implements the controller part of the application.

The following UML diagram shows unidirectional associations because, for example, DemoController has a DemoModel attribute, but not vice versa. A dependency is shown from DemoView to DemoController because DemoView's registerController method has a DemoController parameter.



DemoModel

A DemoModel object has instance variables for storing an array of Points, the number of Points, and a Color. [A Point object has instance variables x and y that can be directly accessed, e.g., point.x and point.y.] DemoModel also has methods for updating and accessing the points and the color. This implementation has some serious shortcomings that you are encouraged to fix. For example, it would be better to use ArrayList<Point> instead of a fixed-size Point array.

DemoController

A DemoController object has instance variables for a DemoModel and a DemoView, which are initialized by the constructor. DemoController implements the MouseMotionListener interface (mouseDragged and mouseMoved methods) and the ListSelectionListener interface (valueChanged method).

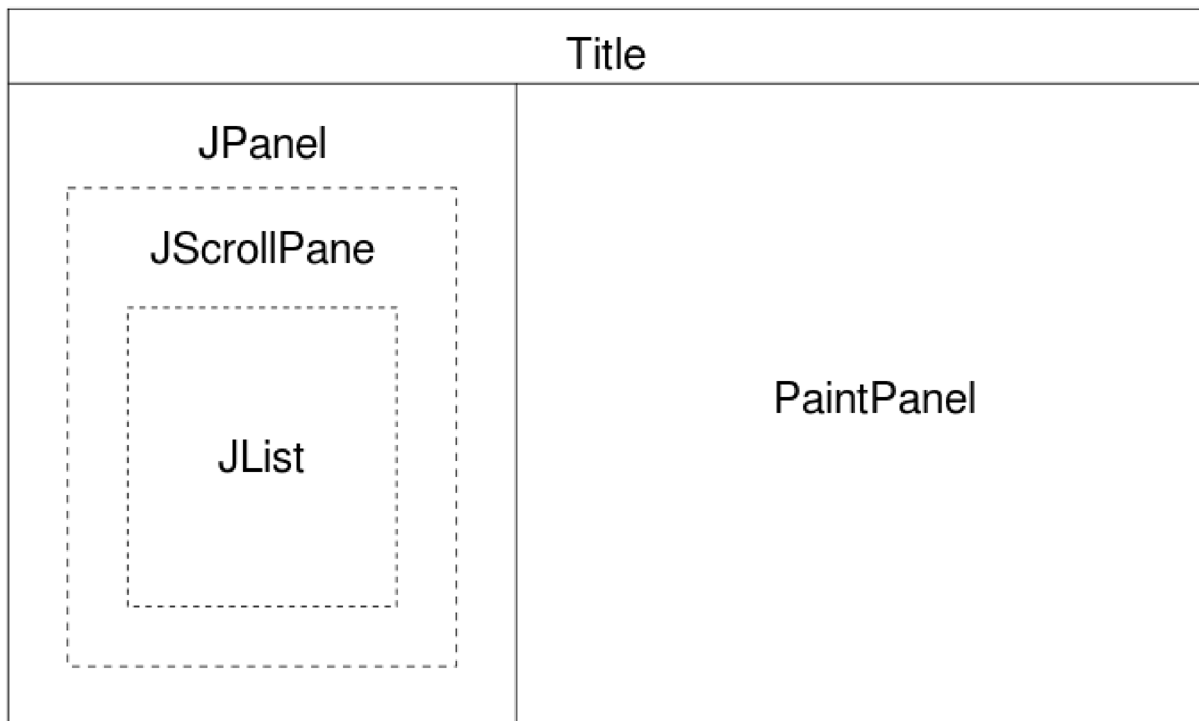
The intent is that the mouseDragged method will be called whenever the user draws with the mouse, and the valueChanged method will be called whenever the user selects a color. The mouseMoved method is needed to implement the MouseMotionListener interface, but does not need any code.

The mouseDragged method adds the Point from the MouseEvent to the model and repaints the window. The valueChanged method obtains the selected color from the view (the event object is not very useful, plus the view knows what the colors are), updates the model's color, and repaints the window.

DemoView

A DemoView object has instance variables for the window's components, plus two class constants for the color selection: an array of Strings to be displayed, and an array of Colors corresponding to the Strings. DemoView is a subclass of JFrame. The constructor organizes the window, and the methods help to coordinate with the controller and the model.

The constructor sets up the window components as follows:



The JFrame consists of two JPanels (PaintPanel is a subclass of JPanel). The PaintPanel is added to the "center" so that it will use up available space. The JList is added to the left/west JPanel, embedded in a JScrollPane.

The registerListener method is used to register the controller with the specific components that the user interacts with. The getSelectedColor method allows the controller to find out what color in the JList was selected by the user.

The paint method might be a little mysterious because there is no paint method call in the code. This method will be called as part of a call to the repaint method of the JFrame(see the methods in DemoController). No code should directly call the paint method.
The paint method sets the background color of the JPanel that contains the JList and calls super.paint(g) so that all the components will be drawn.

PaintPanel

The PaintPanel class implements the rest of the view. It is used to draw the points that the user has drawn. PaintPanel extends JPanel and overrides the paintComponent method to draw the points. paintComponent is another "mysterious" method; it will be called as part of the repaint call. This method loops through all the Points in the model. Note

that model.getPoint(i) returns null if there no Point at index i. For each Point, the fillOval method of the Graphics object g is called. The Graphics class has many useful methods for drawing figures.

The sequence of events that happens when the user drags the mouse in the PaintPanel is as follows:

- A MouseEvent will be created and dispatched to the mouseDragged method of the event handler. Note that when the window was initialized, the DemoController was registered as an event handler for the PaintPanel.
- The mouseDragged method of the DemoController obtains the Point from the MouseEvent, adds the point to the model, and repaints the view.
- The repainting of the window will be scheduled in coordination with the other GUI events. For example, the user might have dragged the mouse quickly and several MouseEvents might already be in the queue. Athough each of these events will result in a repaint call, only one repainting will be scheduled (very inefficient otherwise).
- When the repainting actually occurs, the paint method in DemoView and the paintComponent method in PaintPanel will be called. These need calls to super.paint and super.paintComponent respectively so that the whole window is drawn properly.

7c. Explain interaction diagram for the bridge pattern between the two classes

Ans.

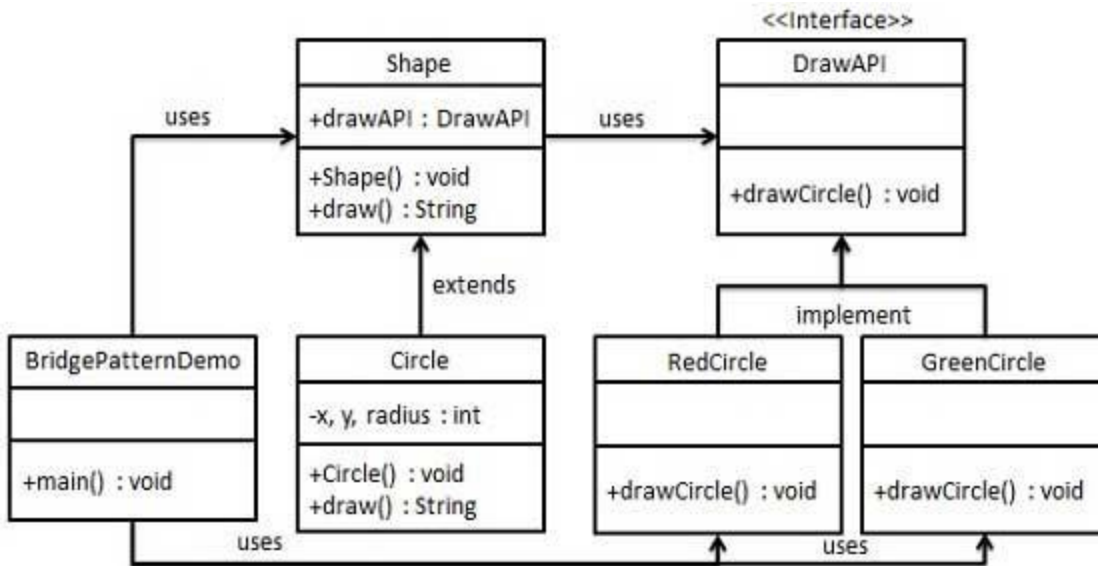
Bridge is used when we need to decouple an abstraction from its implementation so that the two can vary independently. This type of design pattern comes under structural pattern as this pattern decouples implementation class and abstract class by providing a bridge structure between them.

This pattern involves an interface which acts as a bridge which makes the functionality of concrete classes independent from interface implementer classes. Both types of classes can be altered structurally without affecting each other.

We are demonstrating use of Bridge pattern via following example in which a circle can be drawn in different colors using same abstract class method but different bridge implementer classes.

Implementation

We have a *DrawAPI* interface which is acting as a bridge implementer and concrete classes *RedCircle*, *GreenCircle* implementing the *DrawAPI* interface. *Shape* is an abstract class and will use object of *DrawAPI*. *BridgePatternDemo*, our demo class will use *Shape* class to draw different colored circle.



8a. Explain the sequence of operations for adding a label which deals with the environmental variables.

Ans.

3 (a) Drawing Incomplete Items

(1) Incomplete items might be rendered differently from complete items. For eg: For a line, after the first click, the UI would track the mouse movement and draw a line between the first click point and the current mouse location, this line keeps shifting as the user moves the mouse.

(2) Some fields in an incomplete item might not have proper values. Rendering an incomplete item ~~might not ha~~ could be more tricky. An incomplete line, for eg, might have one of the end points null. In such cases, it is inefficient to use the same render method for both incomplete items and complete items because that method will need to check whether the fields are valid and take appropriate actions to handle these special cases.

8b. Explain the design of the controller subsystem with controller class diagram.

Ans.
Subsystems

- subsystem -- a smaller, simpler part of a larger system
  - a subsystem is made of a number of solution domain classes
  - often one developer or development team is responsible for one subsystem
- service -- a set of related operations that share a common purpose
- subsystem interface -- a set of operations of a subsystem available to other subsystems.
  - One subsystem provides services to others, specified through its interface
  - Application Programmer Interface (API) -- refinement of general subsystem interface
- *subsystem decomposition* -- the activity of identifying subsystems, their services, and their relationships to each other

Coupling and Cohesion

- coupling -- the strength of dependencies between two subsystems
  - strongly coupled == changes to one subsystem likely to affect the other
  - loosely coupled == relatively independent (as long as the interface doesn't change)
  - Goal: Strive for *loose couplings*. Don't share attributes; use operations and a well-specified interface
- cohesion / coherence -- strength of dependencies within a subsystem
  - High cohesion: subsystem contains related objects performing similar tasks
  - Low cohesion: subsystem contains a number of unrelated objects
  - Goal: Strive for *high cohesion*

9a. what is remote object? Explain Java Remote Method Invocation
Ans.
Definition

A remote object is an object that defines methods that can be called by a client located in a remote Java Virtual Machine (JVM). A remote object implements one or more remote interfaces that declare remote methods of the object.

Use

Consider the following guidelines when you implement the remote interface inherited by the remote object:

- The implementation class can implement one or more remote interfaces.
- The implementation class can inherit other remote objects.

- The implementation class can declare methods that are not declared by a remote object; those methods can be called locally only.

The RMI (Remote Method Invocation) is an API that provides a mechanism to create distributed application in java. The RMI allows an object to invoke methods on an object running in another JVM.

The RMI provides remote communication between the applications using two objects *stub* and *skeleton*.

Understanding stub and skeleton

RMI uses stub and skeleton object for communication with the remote object.

A remote object is an object whose method can be invoked from another JVM. Let's understand the stub and skeleton objects:

stub

The stub is an object, acts as a gateway for the client side. All the outgoing requests are routed through it. It resides at the client side and represents the remote object. When the caller invokes method on the stub object, it does the following tasks:

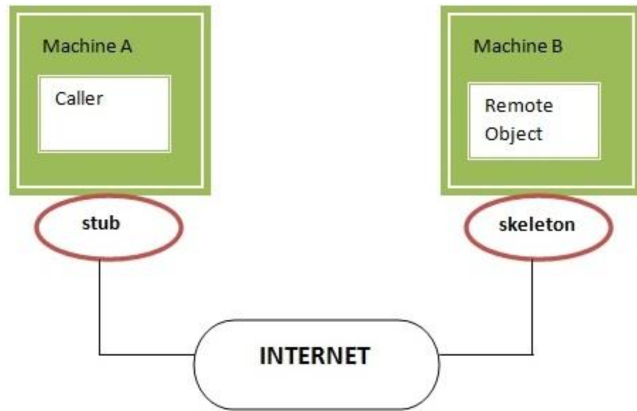
1. It initiates a connection with remote Virtual Machine (JVM),
2. It writes and transmits (marshals) the parameters to the remote Virtual Machine (JVM),
3. It waits for the result
4. It reads (unmarshals) the return value or exception, and
5. It finally, returns the value to the caller.

skeleton

The skeleton is an object, acts as a gateway for the server side object. All the incoming requests are routed through it. When the skeleton receives the incoming request, it does the following tasks:

1. It reads the parameter for the remote method
2. It invokes the method on the actual remote object, and
3. It writes and transmits (marshals) the result to the caller.

In the Java 2 SDK, an stub protocol was introduced that eliminates the need for skeletons.

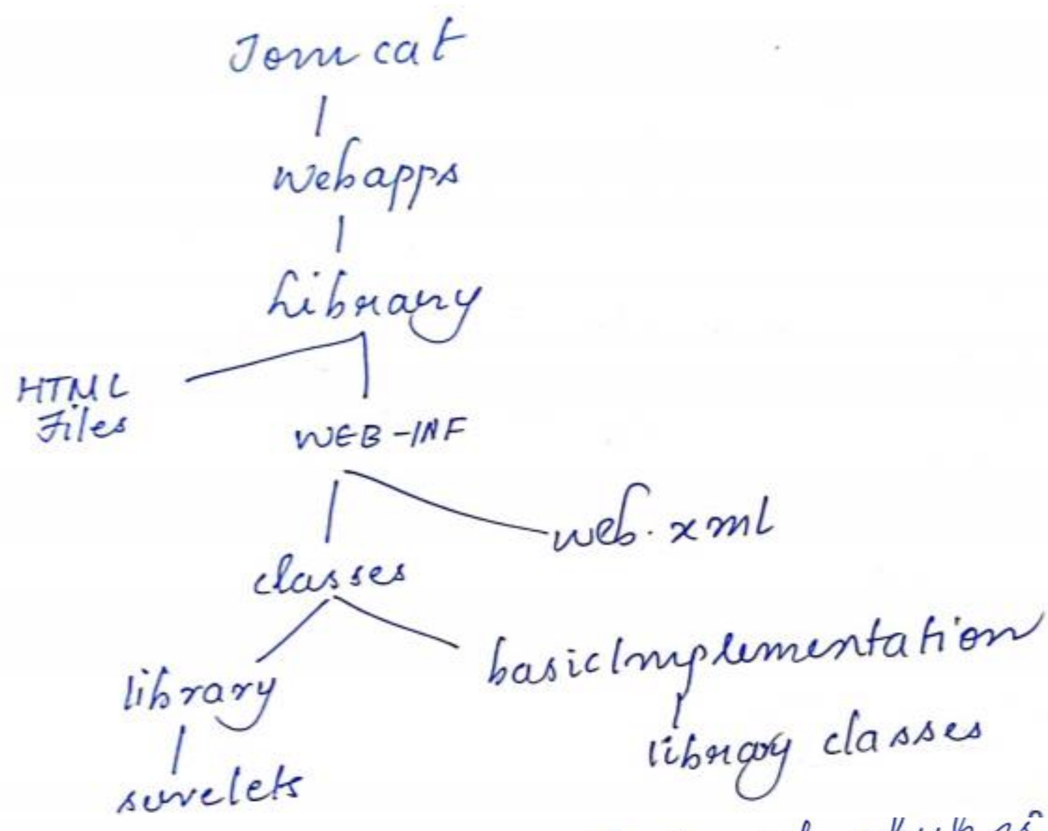


9b. Explain implanting an object oriented system on the web.
Ans.

Different classes of W3Object support different operational interfaces, which are obtained through the use of *interface inheritance*. Abstract classes are used to define an interface to a particular object abstraction, without specifying any particular implementation of the operations. Different classes of W3Objects may share conformance to a particular abstract interface, but may implement the operations differently, in a manner appropriate to the

particular class. For example, consider a Manageable interface, including a migrate operation, for moving objects from one location to another. While the same interface is appropriate for many classes of W3Objects, the implementations may differ; for example, migration of a hypertext object may require some internal link manipulation operations in addition to the operations required by, say, a text file.

9c. Write short notes on servlet container.
Ans.

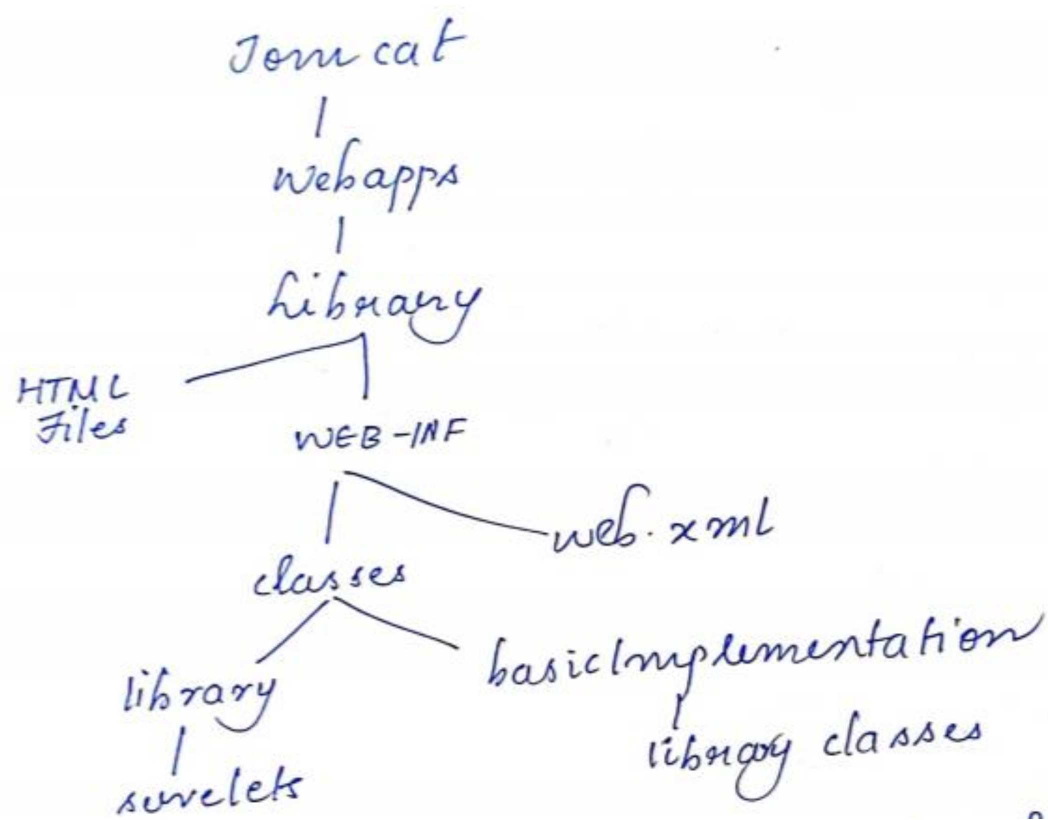


The server runs on Apache Tomcat which is a servlet container. A servelet container is a program that supports servelet execution. The servelets themselves are registered of the servelet container.
URL requests made by a user are converted to specific servelet request by the servelet container. The servelet container is responsible for initiali-zing the servelets and delivering requests made by

<servlet - mapping>
<servelet - name> loginServelet </servelet-name>
<url- pattern>/login </url-pattern>
</servelet - mapping>
The string login is mapped to the servelet name loginServelet.

<servelet>
<servlet-name> login Servlet</servlet-name>
<servlet-class>library.login</servlet-class>
</servlet>
In the web.xml file a servelet such as library IssueBook Initialization is mapped from the <serveletname> of IssueBookInitializationServelet which in turn, is mapped from the URL pattern.

10a. Explain the class diagram for library servlets.
Ans.

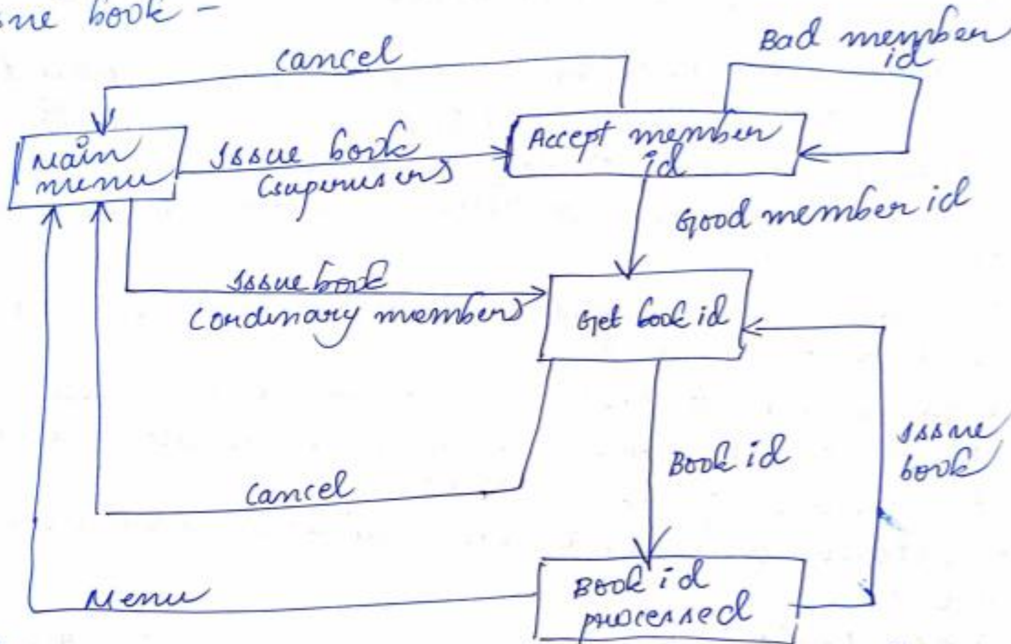


10b. Explain simplified sequence diagram for removing books
Ans.

v) save data: When the data is written to disk no further input is required from the user the system should carry out the task and print a message about the outcome.

vi) Retrieve data: The requirements are the same as saving data.

vii) Issue book -



viii) Place, hold, remove hold, Print Transactions
The requirements for these are similar to issuing

ix) Renew books - For each book user has a choice to renew the system must display title, due date and renewal request message.

10c. Define sessions & session object of servlets.

Ans.

35 The world wide web is the most popular medium for hosting distributed applications. Increasingly, people using the web to book airline tickets, purchase goods etc.
The browser acts as a general purpose client that can interact w/ any application that talks to it using HTTP.
One major characteristic of web based application system is that the client being a general purpose program typically does no application related computation at all. Of course it is possible to ship a Java applet w/ a web page and have the applet do some computation, but this is not hugely popular. Typically the browser recieves web pages from server in HTML and displays the contents acc to the format, a no of tags and values for the tags, specified in it.
The HTML program shipped from a server to a client often needs to be customized to suit the content.
eg:- when we make a reservation on a flight we expect the system to display the details of the flight on which we made the reservation. This requires HTML code to be dynamically constructed. This is done by code at the server. For server side processing there are competing technologies such as JSP, Java servlets, ASP & PHP.