APPENDIX

SOFTWARE DESIGN PRINCIPLES

Q.1. Discuss Software Design Principles in detail.

Ans. Software Design Principles:

- In Java, the design principles are the set of advice used as rules in design making. In Java, the design principles are similar to the design patterns concept.
- The only difference between the Design Principle and Design Pattern is that the Design Principles are more generalized and abstract where as the design pattern contains much more practical advice and concrete. The design patterns are related to the entire class problems, not just generalized coding practices.

Design Principles in Java

There are some of the most important Design Principles which are as follows:

(I) SOLID Principles:

(i) SRP

(ii) LSP

(iii) ISP

(iv) Open/Closed Principle

(v) DIP

(II) Other Principles:

(i) DRY Principles

(ii) KISS Principle

(iii) Composition Over Inheritance Principle

Let's understand all the principles one by one:

(I) **SOLID Principles:**

(i) **Single Responsibility Principle (SRP):**

- SRP is design principle that stands for the Single Responsibility Principle. The SRP principle says that in one class, there should never be two functionalities.
- The class should only have one and only one reason to be changed. When there is more than one responsibility, there is more than one reason to change that class at the same point.
- So, there should not be more than one separate functionality in that same class that may be affected.

The SRP Principle helps us to:

(a) Inherit from a class without inheriting or implementing the methods which our class doesn't need.

(b) Deal with bugs.

(c) Implement changes without confusing co-dependencies.

(ii) **Liskov Substitution Principle (LSP):**

- LSP is another design principle that stands for Liskov Substitution Principle. The LSP states that the derived class should be able to substitute the base classes without changing our code.
- The LSP is closely related to the SRP and ISP. So, violation of either SRP or ISP can be a violation (or become) of the LSP.
- The reason for violation in LSP because if a class performs more than one function, subclasses extending it are less likely to implement those two or more functionalities meaningfully.

(iii) **Interface Segregation Principle (ISP):**

- ISP is another design principle that stands for Interface Segregation Principle. This principle states that the client should never be restricted to dependent on the interfaces that are not using in their entirety.
- This means that the interface should have the minimal set of methods required to ensure functionality and be limited to only one functionality.

For example:

If we create a Burger interface, we don't need to implement the addCheese() method because cheese isn't available for every Burger type.

Let's assume that all burgers need to be baked and have sauce and define that burger interface in the following way:

```
public interface Burger{
   void addSauce().
   void bakeBurger();
}
```

Now, let's implement the Burger interface in Vegetarian Burger and Cheese Burger classes.

```
public class Vegetarian Burger implements Burger
{
public void addTomatoAndCabbage() {
System.out.println("Adding Tomato and Cabbage
vegitables");
   }
      @Override
      public void addSauce() {
System.out.println("Adding sauce in vegetarian
Burger");
   }
@Override
public void bake() {
System.out.println("Baking the vegetarian Burger");
   }
}
public class CheeseBurger implements Burger {
public void addCheese() {
System.out.println("Adding cheese in Burger");
   }
@Override
public void addSauce() {
System.out.println("Adding sauce in Cheese
Burger");
}
@Override
   public void bake() {
System.out.println("Baking the Cheese Burger");
   }
}
```

The VagitarianBurger has Cabbage and Tomato, whereas the CheeseBurger has cheese but needs to be baked and sauce that is defined in the interface. If the addCheese() and addTomatoandCabbage() methods were located in the Burger interface, both the classes have to implement them even though they don't need both.

(iv) **Open/Close Principle:**

- Open/Closed Principle is another important design principle that comes in the category of the solid principles. This principle states that the methods and objects or classes should be closed for modification but open for modification.
- In simple words, this principle says that we should have to implement our modules and classes with the possible future update in mind.
- By doing that, our modules and classes will have a generic design, and in order to extend their behavior, we would not need to change the class itself.
- We can create new fields or methods but in such a way that we don't need to modify the old code, delete already created fields, and rewrite the created methods.
- The Open/Close principle is mainly used to prevent regression and ensure backward compatibility.

(v) **Dependency Inversion Principle (DIP):**

- Another most important design principle is DIP, i.e., Dependency Inversion Principle. This principle states that the low and high levels are decoupled so that the changes in low-level modules don't require rework of high-level modules.
- The high-level and low-level modules should not be dependent on each other. They should be dependent on the abstraction, such as interfaces.
- The Dependency Inversion Principle also states that the details should be dependent on the abstraction, not abstraction should be dependent on the details.

(II) **Other Principles:**

(i) **Don't Repeat Yourself Principle (DRY):**

- The DRY Principle stands for the Don't Repeat Yourself Principle. It is one of the common principles for all programming languages.

- DRY principle say within a system, each piece of logic should have a single unambiguous representation.

Let's take an example of the DRY principle

```
public class Animal {
  public void eatFood() {
System.out.println("Eat Food");
  }
}
public class Dog extends Animal {
  public void woof() {
System.out.println("Dog Woof! ");
  }
}
public class Cat extends Animal {
  public void meow() {
System.out.println("Cat Meow!");
  }
}
```

Both Dog and Cat speak differently, but they need to eat food. The common functionality in both of them is Eating food so, we can put this common functionality into the parent class, i.e., Animals, and then we can extend that parent class in the child class, i.e., Dog and Cat.

Now, each class can focus on its own unique logic, so there is no need to implement the same functionality in both classes.

```
Dog obj1 = new Dog();
obj1.eatFood();
obj1.woof();
Cat obj2 = new Cat();
obj2.eatFood();
obj2.meow();
```

After compile and run the above code, we will see the following output:

Output:

```
Eat food
Cat Meow!
Eat food
Dog Woof!
```

Violation of the DRY Principle

Violations of the DRY principles are referred to as WET solutions. WET is an abbreviation for the following things:

Write everything twice
We enjoy typing
Write every time
Waste everyone's twice.

These violations are not always bad because repetitions are sometimes advisable to make code less inter-dependent, more readable, inherently dissimilar classes, etc.

(ii) **Keep It Simple and Stupid Principle (KISS):**

- Kiss Principle is another designing principle that stands for Keep It Simple and Stupid Principle. This principle is just a reminder to keep our code readable and simple for humans. If several use cases are handled by the method, we need to split them into smaller functions.
- The KISS principle states that for most cases, the stack call should not severely affect our program's performance until the efficiency is not extremely important.
- On the other hand, the lengthy and unreadable methods will be very difficult for human programmers to maintain and find bugs.
- The violation of the DRY principle can be done by ourselves because if we have a method that performs more than one task, we cannot call that method to perform one of them. So, we will create another method for that.

(iii) **Composition Over Inheritance Principle:**

- The Composition Over Inheritance Principle is another important design principle in Java. This principle helps us to implement flexible and maintainable code in Java.
- The principle states that we should have to implement interfaces rather than extending the classes. We implement the inheritance when
- The class need to implement all the functionalities. The child class can be used as a substitute for our parent class. In the same way, we use Composition when the class needs to implement some specific functionalities.

ITERATOR MEDIATOR

Q.2. Explain Iterator Mediator in detail with Example.

Ans. Iterator Mediator:

- Iterate Mediator implements another EIP and will split the message into number of different messages derived from the parent message by finding matching elements for the XPath expression specified.
- New messages will be created for each and every matching element and processed in parallel (default behavior) using either the specified sequence or endpoint.
- Created messages can also be set to process sequentially by setting the optional 'sequential' attribute to 'true'. Parent message can be continued or dropped in the same way as in the clone mediator.
- The 'preservePayload' attribute specifies if the original message should be used as a template when creating the splitted messages, and defaults to 'false', in which case the splitted messages would contain the split elements as the SOAP body.
- The optional 'id' attribute can be used to identify the iterator which created a particular splitted message when nested iterate mediators are used.
- This is particularly useful when aggregating responses of messages that are created using nested Iterate Mediators.

Syntax:

```
<iterate [id="id"] [continueParent=(true | false)]
[preservePayload= (true | false)] [sequential=(true
| false)] (attachPath="xpath")? expression="xpath">
<target [to="uri"] [soapAction="qname"]
[sequence="sequence_ref"]
[endpoint="endpoint_ref"]>
<sequence>
    (mediator)+
</sequence>?
<endpoint>
    endpoint
</endpoint>?
</target>+
</iterate>
```

UI Configuration

Iterate Mediator can be configured with the following options:

(1) Iterate ID:

(Optional) This can be used identify messages created by this iterate mediator, useful when nested iterate mediators are used.

(2) Sequential Mediation:

(True/False), Specify whether splitted messages should be processed sequentially or in parallel. Default value is 'false' (parallel processing).

(3) Continue Parent:

(True/False), Specify whether the original message should be continued or dropped, this is default to 'false'.

(4) Preserve Payload:

(True/False), Specify whether the original message should be used as a template when creating the splitted messages or not, this is default to 'false'.

(5) Iterate Expression:

Xpath expression that matches the elements which you want to split the message from. You can specify the namespaces that you used in the xpath expression by clicking the namespace link in the right hand side.

(6) Attach Path:

(Optional), You can specify an xpath expression for elements that the splitted elements (as expressed in Iterate expression) are attached to, to form new messages

For more information about target please refer Target.

Example:

```
<iterate expression="//m0:getQuote/m0:request"
preservePayload="true"
attachPath="//m0:getQuote"
    xmlns:m0="http://services.samples">
<target>
```

```
<sequence>
<send>
<endpoint>
<address
uri="http://localhost:9000/services/SimpleStockQ
uoteService"/>
</endpoint>
</send>
</sequence>
</target>
</iterate>
```

EXPECTATIONS FROM DESIGN PATTERNS

Q.3. What are the more Expectations from Design Pattern?

OR What are the latest development made in Design Pattern?

Ans. Expectations from Design Patterns:

- None of the design patterns in this course describes new or unproven designs. They are either part of the object-oriented community or are elements of some successful object-oriented systems, neither of which is easy for inexperienced developers to learn from.
- Although these designs are not new, they are captured in a new and accessible way, that is As a catalog of design patterns having a consistent format.
- Despite the size of the course and website, the design patterns at this site capture only a fraction of what an expert might know.
- It does not contain patterns dealing with concurrency, distributed programming or real-time programming.
- It does contain any application domain-specific patterns. It does not tell you how to build user interfaces, how to write device drivers, or how to use an object-oriented database. Each of these areas has its own patterns and are currently being modeled by various developers.

Some Development in Design Pattern are as follows:

(1) **Quizzes and Exercises:**

- Throughout the GOFPatterns website, (Gang of Four Patterns), you will find multiple-choice quizzes and hands-on exercises.
- These learning checks will allow you to assess what you have learned.
- Some of the exercises in this course require you to copy and paste text between a text editor and your web browser. This is easily accomplished on a Windows, Linux and MAC platforms.

(2) **Slide Show Widget:**

- A SlideShow is web component that presents a series of images that you can flip through, either forward or backward.
- In this course, we will be using Slide Shows to illustrate some of the commonly used Gang of Four Patterns.

(3) **MouseOver Tooltip:**

- Whenever you see this graphic within the course, a tooltip that explains or dissects some element of a design pattern will follow.
- Move your mouse cursor over the elements of the image to display the explanations.

(4) **Course glossary:**

- Many of the terms used in the course are defined in a glossary.
- The glossary can be reached from the following link Design Patterns-Glossary or from the home page Gofpatterns Home Page.

MODEL VIEW CONTROLLER

Q.4. Write a note on Model View Controller.

Ans. Model View Controller:

MVC Pattern stands for Model-View-Controller Pattern. This pattern is used to separate application's concerns. Each word in MVC Represent something which are as follows:

(i) **Model:**

Model represents an object or JAVA POJO carrying data. It can also have logic to update controller if its data changes.

(ii) **View:**

View represents the visualization of the data that model contains.

(iii) Controller:

Controller acts on both model and view. It controls the data flow into model object and updates the view whenever data changes. It keeps view and model separate.

Implementation of MVC:

We are going to create a Student object acting as a model. Student View will be a view class which can print student details on console and Student Controller is the controller class responsible to store data in Student object and update view Student View accordingly.

MVC Pattern Demo, our demo class, will use Student Controller to demonstrate use of MVC pattern.

Step 1:

Create Model.

Student.java

```
public class Student {
   private String rollNo;
   private String name;
   public String getRollNo() {
      return rollNo;
   }
   public void setRollNo(String rollNo) {
      this.rollNo = rollNo;
   }
   public String getName() {
      return name;
   }
   public void setName(String name) {
      this.name = name;
   }
}
```

Step 2:

Create View.

StudentView.java

```
public class StudentView {
   public void printStudentDetails(String
         studentName, String studentRollNo){
      System.out.println("Student: ");
      System.out.println("Name: " + studentName);
      System.out.println("Roll No: " +
                              studentRollNo);
   }
}
```

Step 3:

Create Controller.

StudentController.java

```
public class StudentController {
   private Student model;
   private StudentView view;
   public StudentController(Student model,
                            StudentView view){
      this.model = model;
      this.view = view;
   }
   public void setStudentName(String name){
      model.setName(name);
   }
   public String getStudentName(){
      return model.getName();
   }
   public void setStudentRollNo(String rollNo){
      model.setRollNo(rollNo);
   }
   public String getStudentRollNo(){
      return model.getRollNo();
   }
   public void updateView(){
      view.printStudentDetails(model.getName(),
                               model.getRollNo());
   }
}
```

Step 4:

Use the StudentController methods to demonstrate MVC design pattern usage.

MVCPatternDemo.java

```
public class MVCPatternDemo {
   public static void main(String[] args) {
      //fetch student record based on his roll no
                              from the database
```

```
        Student model = retriveStudentFrom
                                    Database();
        //Create a view : to write student details on
                                    console
        StudentView view = new StudentView();
        StudentController controller = new Student
                            Controller(model, view);
        controller.updateView();
        //update model data
        controller.setStudentName("John");
        controller.updateView();
    }
    private static Student retriveStudentFrom
                                    Database(){
        Student student = new Student();
        student.setName("Robert");
        student.setRollNo("10");
        return student;
    }
}
```

Step 5:

Verify the output.

```
Student:
Name: Robert
Roll No: 10
Student:
Name: John
Roll No: 10
```

DATA ACCESS OBJECT PATTERN

Q.5. Describe Data Access Object Pattern in detail.

Ans. Data Access Object Pattern:

Data Access Object Pattern or DAO Pattern is used to separate low level data accessing API or operations from high level business services. Following are the participants in Data Access Object Pattern:

(i) **Data Access Object Interface:**
Data Access Object Interface defines the standard operations to be performed on a model object(s).

(ii) **Data Access Object Concrete Class:**
Data Access Object Concrete Class implements above interface. This class is responsible to get data from a data source which can be database / xml or any other storage mechanism.

(iii) **Model Object or Value Object:**
Model Object or Value Object is simple POJO containing get/set methods to store data retrieved using DAO class.

Implementation of DAO Pattern:

We are going to create a Student object acting as a Model or Value Object. StudentDao is Data Access Object Interface. StudentDaoImpl is concrete class implementing Data Access Object Interface. DaoPatternDemo, our demo class, will use StudentDao to demonstrate the use of Data Access Object pattern.

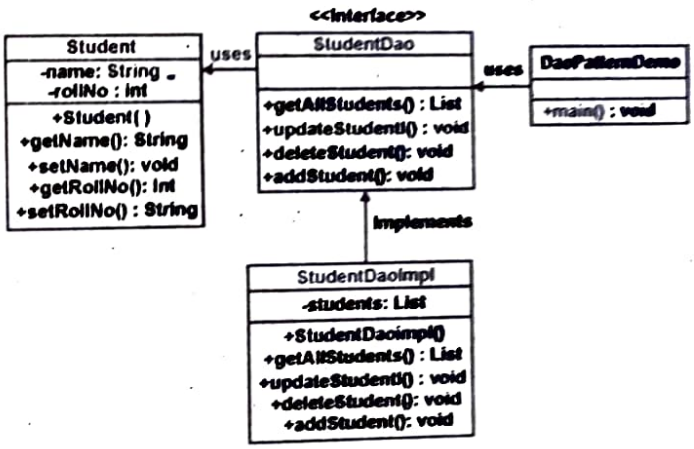

Fig. Data Access Object Pattern UML

Step 1:

Create Value Object.

Student.java

```
public class Student {
   private String name;
   private int rollNo;
   Student(String name, int rollNo){
      this.name = name;
      this.rollNo = rollNo;
   }
   public String getName() {
      return name;
   }
   public void setName(String name) {
      this.name = name;
   }
```

```
    public int getRollNo() {
        return rollNo;
    }
    public void setRollNo(int rollNo) {
        this.rollNo = rollNo;
    }
}
```

Step 2:

Create Data Access Object Interface.

StudentDao.java

```
import java.util.List;
public interface StudentDao {
   public List<Student> getAllStudents();
   public Student getStudent(int rollNo);
   public void updateStudent(Student student);
   public void deleteStudent(Student student);
}
```

Step 3:

Create concrete class implementing above interface.

StudentDaoImpl.java

```
import java.util.ArrayList;
import java.util.List;
public class StudentDaoImpl implements
StudentDao {
   //list is working as a database
   List<Student> students;
   public StudentDaoImpl(){
      students = new ArrayList<Student>();
      Student student1 = new Student("Robert",0);
      Student student2 = new Student("John",1);
      students.add(student1);
      students.add(student2);
   }
   @Override
   public void deleteStudent(Student student) {
      students.remove(student.getRollNo());
      System.out.println("Student: Roll No " +
student.getRollNo() + ", deleted from database");
   }
//retrive list of students from the database
@Override
public List<Student> getAllStudents() {
   return students;
}
@Override
public Student getStudent(int rollNo) {
   return students.get(rollNo);
}
@Override
public void updateStudent(Student student) {
   students.get(student.getRollNo()).setName
                              (student.getName());
   System.out.println("Student: Roll No " + student
      .getRollNo() + ", updated in the database");
   }
}
```

Step 4:

Use the StudentDao to demonstrate Data Access Object pattern usage.

DaoPatternDemo.java

```
public class DaoPatternDemo {
   public static void main(String[] args) {
      StudentDao studentDao = new Student
                                    DaoImpl();
      //print all students
      for (Student student : studentDao.get
                                AllStudents()) {
         System.out.println("Student: [RollNo : " +
         student.getRollNo() + ", Name : " +
                     student.getName() + " ]");
      }
      //update student
      Student student =studentDao.getAllStudents()
                                        .get(0);
      student.setName("Michael");
      studentDao.updateStudent(student);
      //get the student
      studentDao.getStudent(0);
      System.out.println("Student: [RollNo : " +
               student.getRollNo() + ", Name :
                  " + student.getName() + " ]");

   }
}
```

Step 5:

Verify the output.

Student: [RollNo : 0, Name : Robert]

Student: [RollNo : 1, Name : John]

Student: Roll No 0, updated in the database

Student: [RollNo : 0, Name : Michael]

TRANSFER OBJECT DESIGN PATTERN

Q.6. Write in detail about Transfer Object Design Pattern.

Ans. Transfer Object Design Pattern:

- The Transfer Object pattern is used when we want to pass data with multiple attributes in one shot from client to server. Transfer object is also known as Value Object.
- Transfer Object is a simple POJO class having getter/setter methods and is serializable so that it can be transferred over the network. It does not have any behavior.
- Server Side business class normally fetches data from the database and fills the POJO and send it to the client or pass it by value. For client, transfer object is read-only. Client can create its own transfer object and pass it to server to update values in database in one shot.

Following are the entities of this type of Design Pattern:

(i) **Business Object:**
Business Service fills the Transfer Object with data.

(ii) **Transfer Object:**
Simple POJO having methods to set/get attributes only.

(iii) **Client:**
Client either requests or sends the Transfer Object to Business Object.

Implementation of Transfer Object Design Pattern:

We are going to create a StudentBO as Business Object,Student as Transfer Object representing our entities.

TransferObjectPatternDemo, our demo class, is acting as a client here and will use StudentBO and Student to demonstrate Transfer Object Design Pattern.

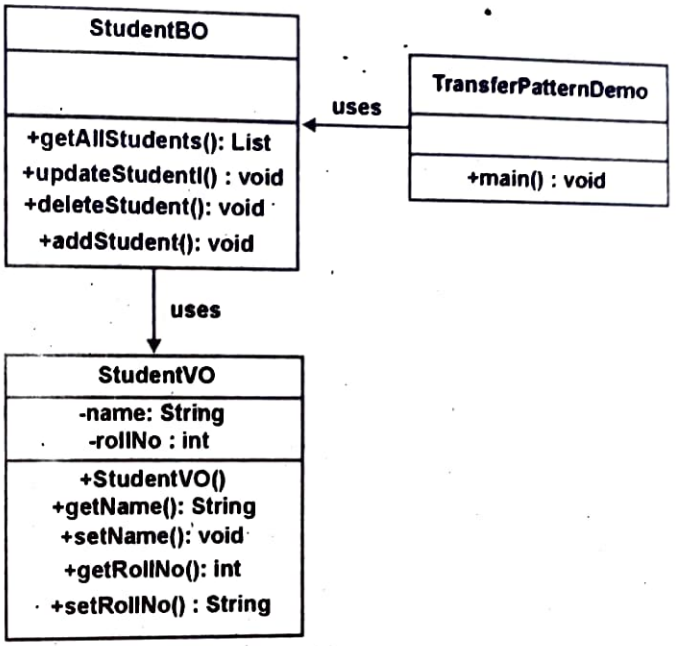

Fig. Transfer Object Pattern UML Diagram

Step 1:

Create Transfer Object.

StudentVO.java

```
public class StudentVO {
   private String name;
   private int rollNo;
   StudentVO(String name, int rollNo){
      this.name = name;
      this.rollNo = rollNo;
   }
   public String getName() {
      return name;
   }
   public void setName(String name) {
      this.name = name;
   }
   public int getRollNo() {
      return rollNo;
   }
   public void setRollNo(int rollNo) {
      this.rollNo = rollNo;
   }
}
```

Step 2:

Create Business Object.

StudentBO.java

```
import java.util.ArrayList;
import java.util.List;
public class StudentBO {
  //list is working as a database
  List<StudentVO> students;
  public StudentBO(){
     students = new ArrayList<StudentVO>();
     StudentVO student1 = new StudentVO
                                  ("Robert",0);
     StudentVO student2 = new StudentVO
                                  ("John",1);
     students.add(student1);
     students.add(student2);
  }
  public void deleteStudent(StudentVO student) {
     students.remove(student.getRollNo());
     System.out.println("Student: Roll No " +
                        student.getRollNo() + ",
                        deleted from database");
  }
  //retrive list of students from the database
  public List<StudentVO> getAllStudents() {
     return students;
  }
  public StudentVO getStudent(int rollNo) {
     return students.get(rollNo);
  }
  public void updateStudent(StudentVO student) {
     students.get(student.getRollNo())
                .setName(student.getName());
     System.out.println("Student: Roll No " +
                        student.getRollNo() +",
                       updated in the database");
  }
}
```

Step 3:

Use the StudentBO to demonstrate Transfer Object Design Pattern.

TransferObjectPatternDemo.java

```
public class TransferObjectPatternDemo {
  public static void main(String[] args) {
     StudentBO studentBusinessObject = new
                                  StudentBO();

     //print all students
     for (StudentVO student : studentBusiness
                        Object.getAllStudents()) {
     System.out.println("Student: [RollNo : " +
          student.getRollNo() + ", Name : "
                + student.getName() + " ]");
     }
     //update student
     StudentVO student = studentBusiness
                Object.getAllStudents().get(0);
     student.setName("Michael");
     studentBusinessObject.update
                                  Student(student);
     //get the student
     student = studentBusinessObject.get
                                  Student(0);
     System.out.println("Student: [RollNo : " +
               student.getRollNo() + ", Name : "
                    + student.getName() + " ]");
  }
}
```

Step 4:

Verify the output.

```
Student: [RollNo : 0, Name : Robert ]
Student: [RollNo : 1, Name : John ]
Student: Roll No 0, updated in the database
Student: [RollNo : 0, Name : Michael ]
```