**Lab: Design Patterns**

This manual is intended for enhancing understanding of the subject of Design Patterns.

The design process is typically split into distinct phases: Object Oriented Design (OOD) and Design Patterns. Typically, the time spent on getting the design right before you start programming will almost always save you time in the end. It's much, much easier to make major changes on a design, which is after all just squiggly lines on paper, and then it is to make changes in hundreds or thousands of lines of code.

Students are advised to thoroughly go through this manual rather than only topics mentioned in the lecture as practical aspects are the key to understanding and conceptual visualization of theoretical aspects covered in the lecture notes.

Good Luck for your Enjoyable Laboratory Sessions.

1. **Write a Program to implement Singleton pattern.**

Theory:

* Singleton pattern is one of the simplest design patterns in Java. This type of design pattern comes under creational pattern as this pattern provides one of the best way to create an object.
* This pattern involves a single class which is responsible to creates own object while making sure that only single object get created. This class provides a way to access its only object which can be accessed directly without need to instantiate the object of the class. Enables the creator to defer Product creation to a sub-class.
* A class with only one single possible instance.

1. Private constructor
2. Global access

* Intent: Ensure a class has only one instance, and provide a global point of access to it. Encapsulated "just-in-time initialization" or "initialization on first use".
* Applicability: Use the Singleton pattern when

1. there must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.
2. when the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.

Class Diagram:

**Diagram

Description automatically generated**

**Sample Code:**

Step 1 - Create a Singleton Class, i.e., *SingleObject.java*

public class SingleObject {

//create an object of SingleObject

private static SingleObject instance = new SingleObject();

//make the constructor private so that this class cannot be

//instantiated

private SingleObject(){}

//Get the only object available

public static SingleObject getInstance(){ return instance;

}

public void showMessage(){ System.out.println("Hello World!");

}

}

Step 2 - Get the only object of the singleton class from a different class, i.e., *SingletonPatternDemo.java*

public class SingletonPatternDemo { public static void main(String[] args) {

//illegal construct

//Compile Time Error: The constructor SingleObject() is not visible

//SingleObject object = new SingleObject();

//Get the only object available

SingleObject object = SingleObject.getInstance();

//show the message object.showMessage();

}

}

Step 3 - Verify the output.

Hello World!

1. **Write a program to implement visitor pattern.**

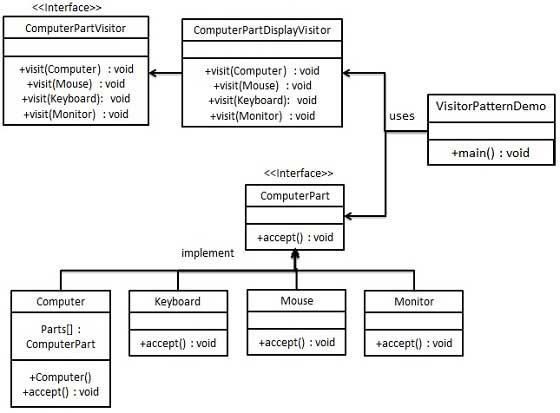
The Visitor Pattern can be difficult to understand at first, but is excellent at separating functionality from an object. So instead of making classes large and bulky, we can separate the unnecessary functionality and visit them only as necessary.

Theory:

* In Visitor pattern, we use a visitor class which changes the executing algorithm of an element class. By this way, execution algorithm of element can vary as and when visitor varies. This pattern comes under behavior pattern category. As per the pattern, element object has to accept the visitor object so that visitor object handles the operation on the element object.
* Intent: Represent 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.
* Applicability: Use the Visitor pattern when
  1. an object structure contains many classes of objects with differing interfaces, and you want to perform operations on these objects that depend on their concrete classes.
  2. many distinct and unrelated operations need to be performed on objects in an object structure, and you want to avoid "polluting" their classes with these operations. Visitor lets you keep related operations together by defining them in one class. When the object structure is shared by many applications, use Visitor to put operations in just those applications that need them.
  3. the classes defining the object structure rarely change, but you often want to define new operations over the structure. Changing the object structure classes requires redefining the interface to all visitors, which is potentially costly. If the object structure classes change often, then it's probably better to define the operations in those classes.

Let’s first practice writing a typical Visitor class using Java language.

Class Diagram:



Step 1 - Define an interface to represent element, i.e., *ComputerPart.java*

public interface ComputerPart {

public void accept(ComputerPartVisitor computerPartVisitor);

}

Step 2 - Create concrete classes extending the above class, i.e., *Keyboard.java*

public class Keyboard implements ComputerPart {

@Override

public void accept(ComputerPartVisitor computerPartVisitor) { computerPartVisitor.visit(this);

}

}

*Monitor.java:*

public class Monitor implements ComputerPart {

@Override

public void accept(ComputerPartVisitor computerPartVisitor) { computerPartVisitor.visit(this);

}

}

*Mouse.java:*

public class Mouse implements ComputerPart {

@Override

public void accept(ComputerPartVisitor computerPartVisitor) { computerPartVisitor.visit(this);

}

}

*Computer.java:*

public class Computer implements ComputerPart {

ComputerPart[] parts;

public Computer(){

parts = new ComputerPart[] {new Mouse(), new Keyboard(), new Monitor()};

}

@Override

public void accept(ComputerPartVisitor computerPartVisitor) { for (int i = 0; i < parts.length; i++) {

parts[i].accept(computerPartVisitor);

}

computerPartVisitor.visit(this);

}

}

Step 3 - Define an interface to represent visitor, i.e., *ComputerPartVisitor.java*

public interface ComputerPartVisitor {

public void visit(Computer computer); public void visit(Mouse mouse); public void visit(Keyboard keyboard); public void visit(Monitor monitor);

}

Step 4 - Create concrete visitor implementing the above class, i.e., *ComputerPartDisplayVisitor.java*

public class ComputerPartDisplayVisitor implements ComputerPartVisitor {

@Override

public void visit(Computer computer) { System.out.println("Displaying Computer.");

}

@Override

public void visit(Mouse mouse) { System.out.println("Displaying Mouse.");

}

@Override

public void visit(Keyboard keyboard) { System.out.println("Displaying Keyboard.");

}

@Override

public void visit(Monitor monitor) { System.out.println("Displaying Monitor.");

}

}

Step 5 - Use the *ComputerPartDisplayVisitor* to display parts of *Computer*, i.e., *VisitorPatternDemo.java*

public class VisitorPatternDemo { public static void main(String[] args) {

ComputerPart computer = new Computer(); computer.accept(new ComputerPartDisplayVisitor());

}

}

Step 6 - Verify the output.

Displaying Mouse. Displaying Keyboard. Displaying Monitor.

Displaying Computer.

**Conclusion: Thus ,we have executed visitor pattern successfully.**

1. **Setting up the Visitor Pattern in Javascript**

Traditionally, the Visitor Pattern is applied to strongly typed languages, where polymorphism allows for one method with different input types to distinguish which method gets called. Javascript is a weakly typed language, and using the Visitor Pattern is not quite the same.

Here is a typical Visitor class using the strongly typed JAVA language (which skips the visitor interface to limit the total number of classes):

The clear difference between each of the Visit functions is the type of the input. We aren’t able to distinguish the type of the input from within the argument, so different visitor objects are created for each required visitor.

public class AutomobileVisitor

{

public void Visit(Automobile variable)

{

//this method is usually abstracted into an interface that this class implements

//automobile is an abstract class in this case, so no need for operations

}

public void Visit(Truck variable)

{

//Truck inherits Automobile

}

public void Visit(Car variable)

{

//Car inherits Automobile

}

public void Visit(MonsterTruck variable)

{

//MonsterTruck inherits Automobile

}

}

Step 1- Let’s first create a new file, i.e. visitorDemo.js, with code at below so that each object will have an Accept(variable) method, and each object will have a different name:

var CarVisitor = function()

{

var visit = function(carVariable)

{

//do some operations on carVariable

}

}

var TruckVisitor = function()

{

var visit = function(truckVariable)

{

//do some operations on truckVariable

}

}

var MonsterTruckVisitor = function()

{

var visit = function(monsterTruckVariable)

{

//do some operations on monsterTruckVariable

}

}

Step 2 - Implementing the previous displayed code requires some definition for what carVariable, truckVariable, and monsterTruckVariable are. So let’s define them:

Note that this pattern implements the double dispatch method, i.e., accept() in the same way as other languages. An accept method with a visitor for an input that immediately visits the automobile.

var carVariable = function()

{

var seats = 5;

var doors = 4;

this.accept = function(visitorObject)

{

visitorObject.visit(this);

}

}

var truckVariable = function()

{

var towPackage = true;

var doors = 2;

this.accept = function(visitorObject)

{

visitorObject.visit(this);

}

}

var monsterTruckVariable = function()

{

var looksLikeADragon = true;

var doors = 1.5;

this.accept = function(visitorObject)

{

visitorObject.visit(this);

}

}

Step 3 - Let’s add some functionality to our visitors. In the same file, add the following code:

var CarVisitor = function()

{

this.visit = function(car)

{

if(car.seats > 2)

{

console.log('this is clearly a car for old people');

}

else

{

console.log('My bet is this car has at least 2 cylinders');

}

}

}

var TruckVisitor = function()

{

this.visit = function(truckVar)

{

if(truckVar.towPackage)

{

console.log('we need to buy a boat')

}

}

}

var MonsterTruckVisitor = function()

{

this.visit = function(monsterTruckVar)

{

if(monsterTruckVar.looksLikeADragon)

{

console.log('that is a badass monster truck')

}

else

{

console.log('loser')

}

}

}

Note that each Visitor uses a different property in their respective Visit method.

Step 4 - Verify the output.

1. Add the following code at the end of the file:

var myCar = new carVariable();

myCar.seats = 2;

myCar.accept(new CarVisitor());

1. Run the file in Terminal, using the command: node visitorDemo.js, and you should see the following messafge:

My bet is this car has at least 2 cylinders

1. You may also try to add the following code at the end, and repeat the above step:

var myMonsterTruck = new monsterTruckVariable();

myMonsterTruck.looksLikeADragon = false;

myMonsterTruck.accept(new MonsterTruckVisitor());

1. Can you use a MonsterTruckVisitor on a Car object? Try and find out on your own by adding the following lines in your code:

var myCar2 = new carVariable();

myCar2.seats = 2;

myCar2.accept(new MonsterTruckVisitor());

1. **Submission**

Submit the link to a repo containing three folders (i.e., Part1, Part2, Part3, respectively), each folder containing the code produced from the above activities in each Part.