**Name: Basavraj Jaliminche**

**Student ID: 8800149**

**Email: Bjaliminche0149@conestogac.on.ca**

**Assignment 1**

1. **[2 points] A palindrome is a string that equals its reverse. For example, the strings, “abba” and “racecar”, are palindromes but the string, “prog8610”, is not a palindrome. Write a method that returns true if the given string is a palindrome, or false otherwise.**

---🡪

**public** **class** Palindrome{

**static** **boolean** isPalindrome(String str)

{

**int** i = 0, j = str.length() - 1;

**while** (i < j) {

**if** (str.charAt(i) != str.charAt(j))

**return** **false**;

i++;

j--;

}

**return** **true**;

}

**public** **static** **void** main(String[] args)

{

String str = "Basavraj";

str = str.toLowerCase();

// passing bool function till holding true

**if** (*isPalindrome*(str))

// It is a palindrome condition

System.***out***.print("Yes its palindrome");

**else**

// Not a palindrome condition

System.***out***.print("No its not palindrome");

}

}

1. **Web clients request pages from servers using a URL. e.g. https://www.conestogac.on.ca/index.jsp. URLs are represented and transmitted over the Internet using ASCII; and URL encoding is the conversion of a non-ASCII characters into ASCII, commonly known as percent-encoding. Write a method that accepts a string and performs a simplified percent-encoding by replacing space characters with %20.**

----🡪

**public** **static** String replace(String s1)

{

**int** length = s1.length();

**int** spaceCount = 0, index = 0;

**while**(index < length){

**if** (s1.charAt(index) == ' ')

spaceCount++;

index = index + 1;

}

**if**(spaceCount == 0)

**return** s1;

**int** newLength = length + 2 \* spaceCount;

**char** newCharacters[] = **new** **char** [newLength];

**int** j=0;

**for** (index = 0; index < newLength;)

{

**if** (j < length) {

**if** (s1.charAt(j) == ' ') {

newCharacters[index] = '%';

newCharacters[index + 1] = '2';

newCharacters[index + 2] = '0';

index = index+3;

} **else** {

newCharacters[index] = s1.charAt(j);

index = index + 1;

}

}

j++;

}

String newString = String.valueOf(newCharacters);

**return** newString;

}

1. **Write a method that accepts two strings, s1 and s2, and returns true if s2 is a substring of s1, or false otherwise.**

--🡪

**public** **static** **boolean** checkSubstring(String s1, String s2)

{

**for** (**int** k = 0; k <= s2.length() - s1.length(); k++)

{

**int** l;

**for** (l= 0; j < s1.length(); l++)

**if** (s2.charAt(k + l) != s1.charAt(l))

**break**;

**if** (l == s1.length())

**return** **true**;

}

**return** **false**;

}

1. **Write a method that returns the transpose of a given matrix. Recall that that the transpose matrix operation switches the row and column indices. e.g. the transpose of [ 0 1 2 3 ] is [ 0 2 1 3 ].**

---🡪

**public** **class** MatrixTranspose{

**public** **static** **void** main(String args[]){

**int** matrix[][]={{0,1},{2,3}};

System.***out***.println("Printing Matrix without transpose:");

**for**(**int** i=0;i<2;i++)

{

**for**(**int** j=0;j<2;j++){

System.***out***.print(matrix[i][j]+" ");

}

System.***out***.println();//new line

}

System.***out***.println("Printing Matrix After Transpose:");

**for**(**int** i=0;i<2;i++)

{

**for**(**int** j=0;j<2;j++){

System.***out***.print(matrix[j][i]+" ");

}

System.***out***.println();

}

}

}

1. **Write a method that returns true if the provided string contains entirely unique characters, or false otherwise.**

----🡪

**static** **boolean** uniqueCharacters(String str)

{

**for** (**int** i = 0; i < str.length(); i++)

**for** (**int** j = i + 1; j < str.length(); j++)

**if** (str.charAt(i) == str.charAt(j))

**return** **false**;

**return** **true**;

}

1. **Consider the following two use cases: • Create Account: Unregistered user inputs forename, surname, email address, and password (twice) • Create Item: Administrator or staff inputs item name and category. Create two separate Web forms that would allow unregistered users to create an account and administrators or staff to create an item in the system, respectively. Your solution is to be written using valid HTML, presented using a CSS stylesheet, and the inputted data validated on the client side with HTML and JavaScript.**

****

**All files of HTML CSS JS attached to it .**

****

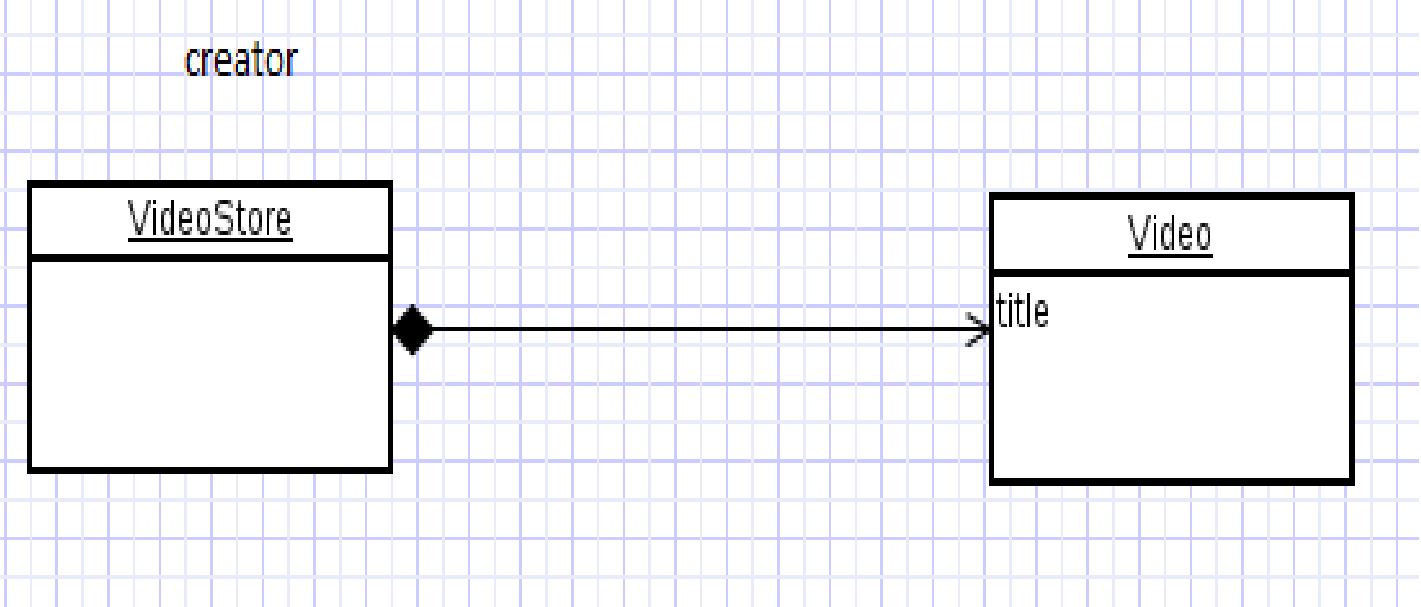
1. **[5 points] Write example implementations of the following GRASP patterns: Creator, Information Expert, High Cohesion, Polymorphism, Pure Fabrication, Indirection.**

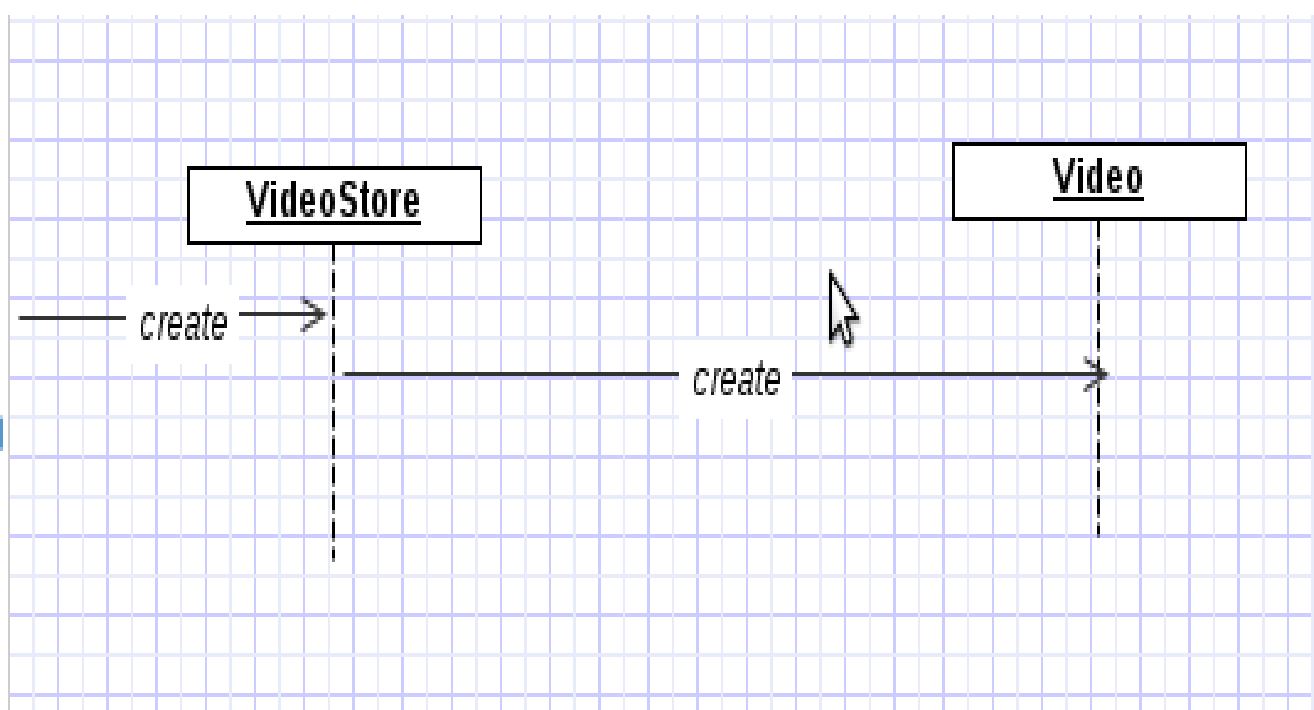
**--🡪** GRASP stands for General Responsibility Assignment Software Patterns. GRASP helps us in deciding which responsibility should be assigned to which object/class.There are 9 grasp patterns .

* Creator
* Information Expert
* Low Coupling
* Controller
* High Cohesion
* Indirection
* Polymorphism
* Protected Variations
* Pure Fabrication

1. **Creator:**

Consider VideoStore and Video in that store. VideoStore has an aggregation association with Video. I.e, VideoStore is the container and the Video is the contained object. So, we can instantiate video object in VideoStore class.

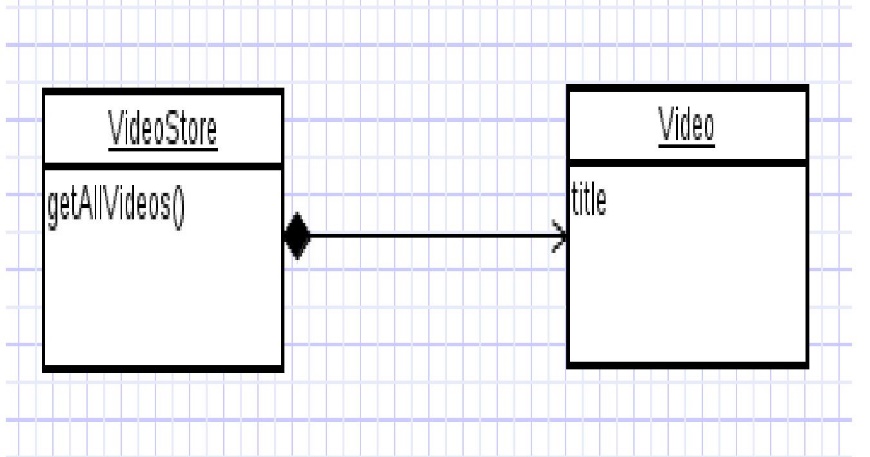


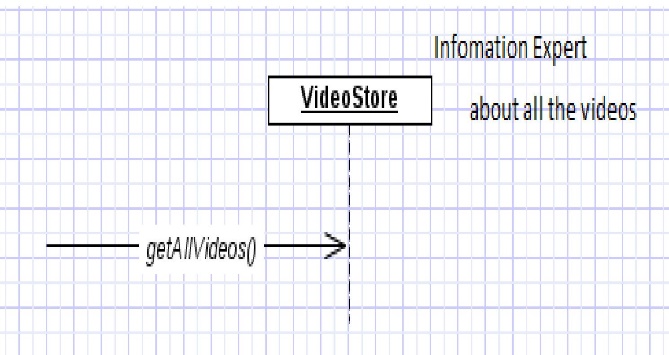


**2.Information Expert:**

Expert principle says – assign those responsibilities to o for which o has the information to fulfill that responsibility. ● They have all the information needed to perform operations, or in some cases they collaborate with others to fulfill their responsibilities.

Assume we need to get all the videos of a VideoStore. Since VideoStore knows about all the videos, we can assign this responsibility of giving all the videos can be assigned to VideoStore class. VideoStore is the information expert.

****

****

**3.Low Coupling:**

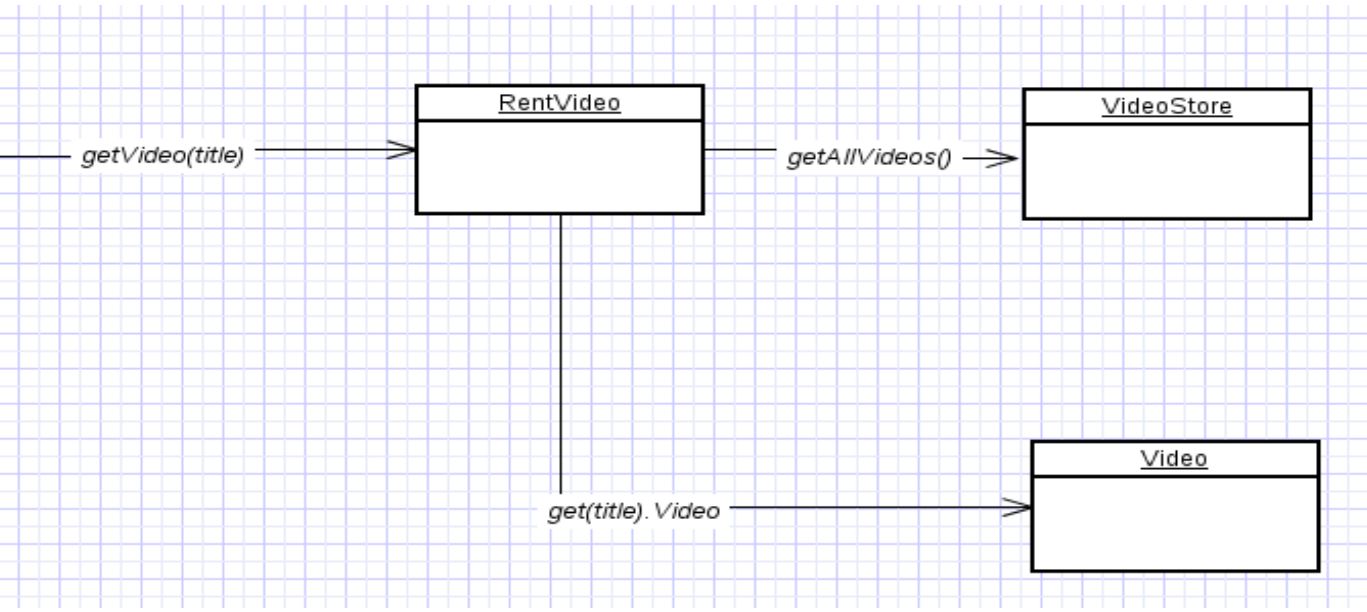
Coupling – object depending on other object. When depended upon element changes, it affects the dependent also.

Low Coupling – How can we reduce the impact of change in depended upon elements on dependent elements. Prefer low coupling – assign responsibilities so that coupling remain low. Minimizes the dependency hence making system maintainable, efficient and code reusable.

Two elements are coupled, if

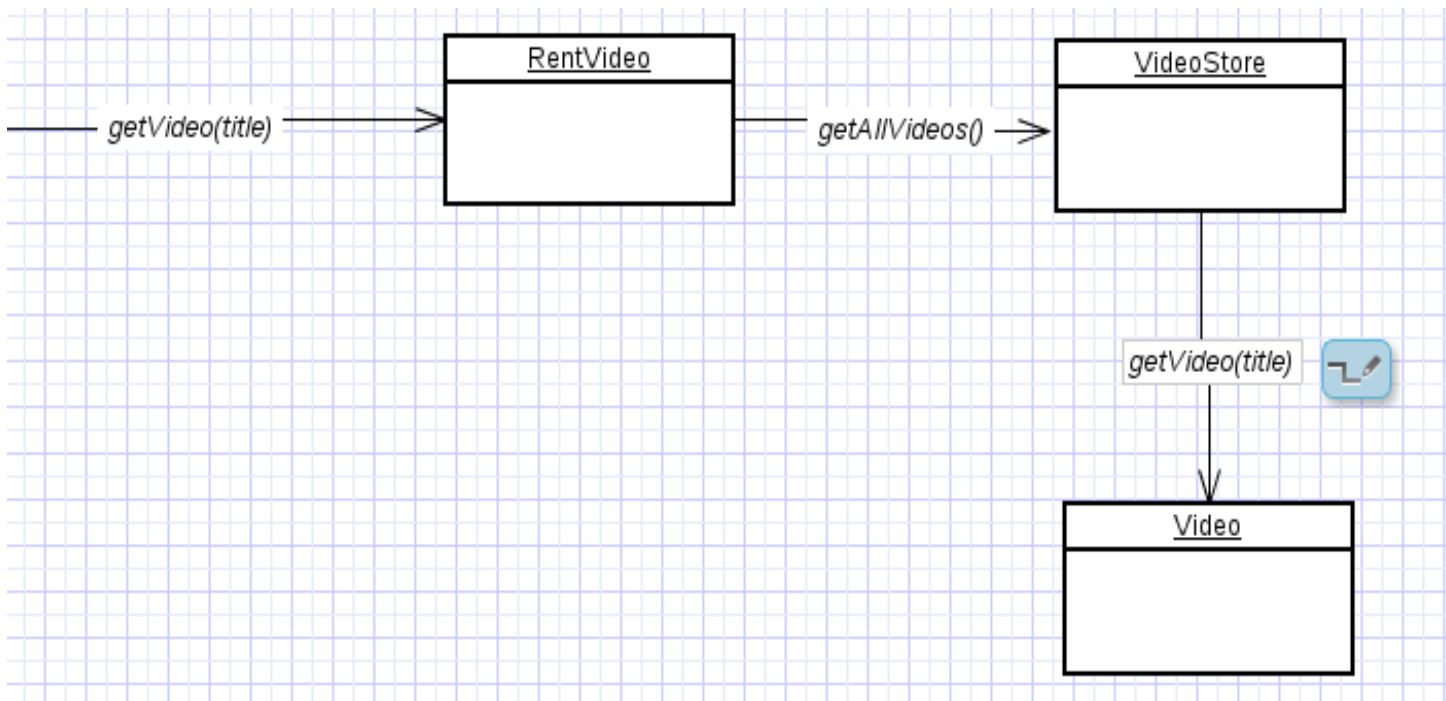
– One element has aggregation/composition association with another element.

– One element implements/extends other element.



here class Rent knows about both VideoStore and Video objects. Rent is depending on both the classes.

VideoStore and Video class are coupled, and Rent is coupled with VideoStore. Thus providing low coupling.

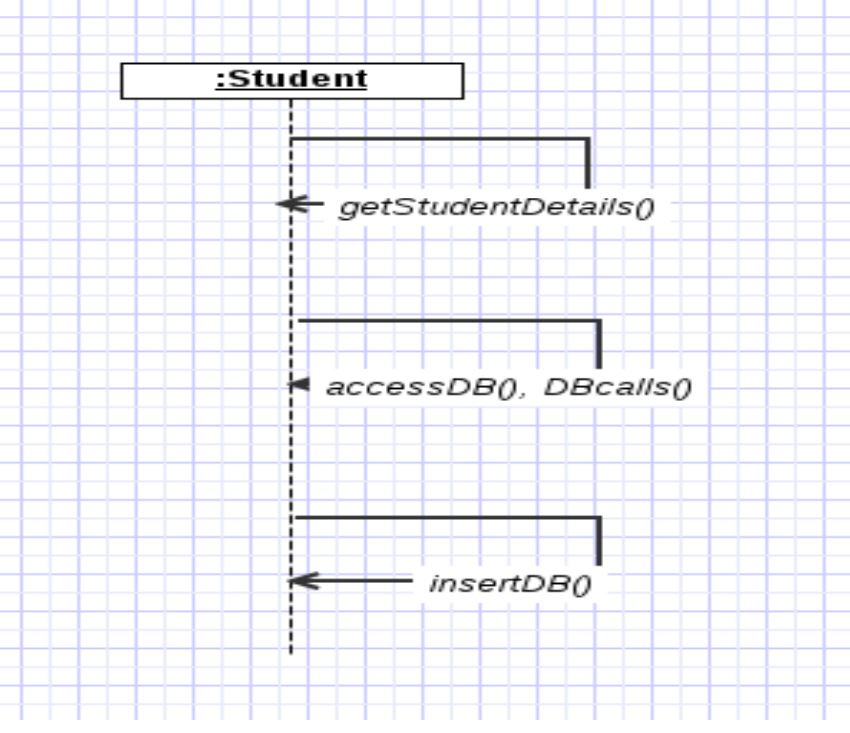


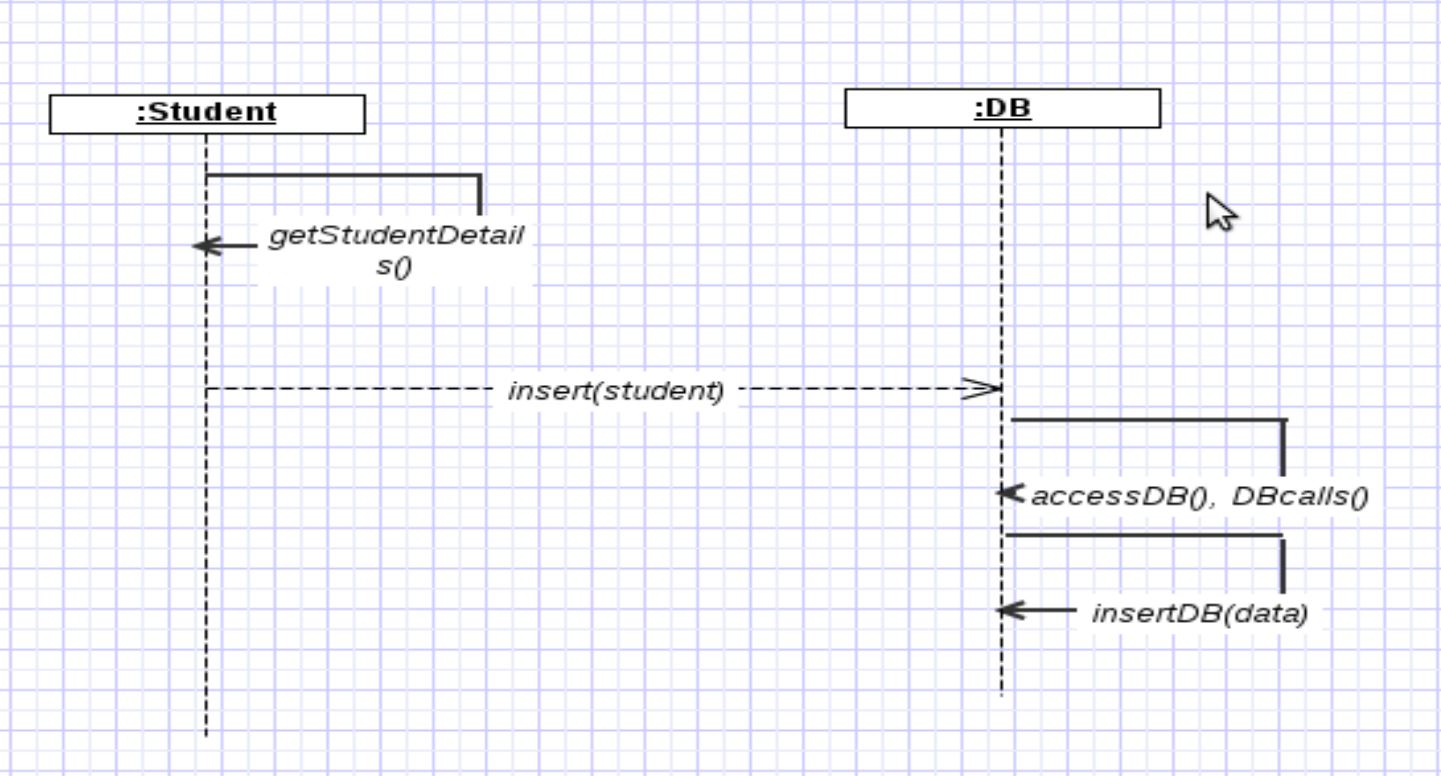
**4. High Cohesion**

Related responsibilities in to one manageable unit. Prefer high cohesion. Clearly defines the purpose of the element ●.Benefits

– Easily understandable and maintainable.

– Code reuse

****

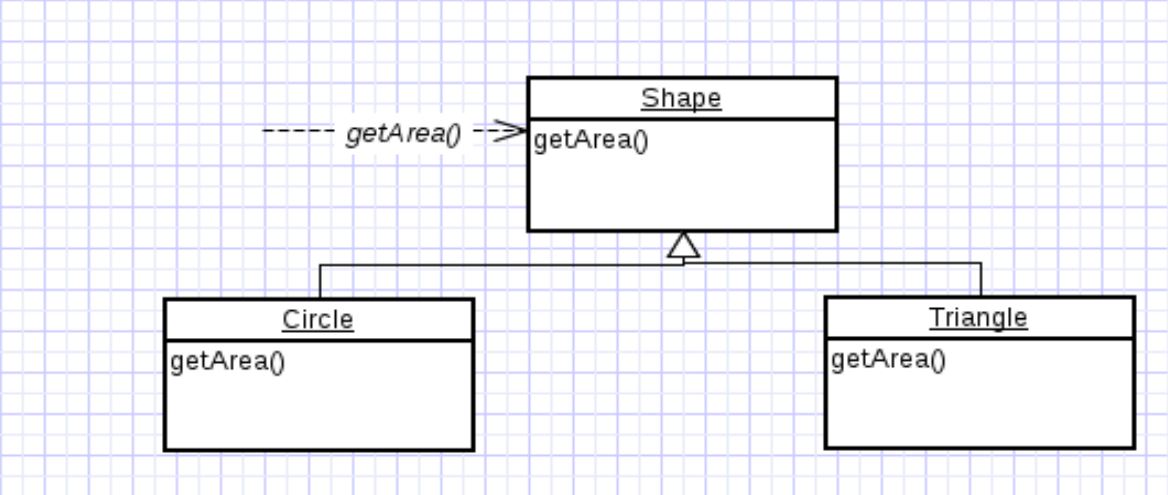
****

**5. Polymorphism:**

Polymorphism guides us in deciding which object is responsible for handling those varying elements.

● Benefits: handling new variations will become easy.

the getArea() varies by the type of shape, so we assign that responsibility to the subclasses.

****

By sending message to the Shape object, a call will be made to the corresponding sub class object – Circle or Triangle.

**6. Pure Fabrication**

Fabricated class/ artificial class – assign set of related responsibilities that doesn't represent any domain object.

● Provides a highly cohesive set of activities.

● Behavioral decomposed – implements some algorithm.

● Examples: Adapter, Strategy ● Benefits: High cohesion, low coupling and can reuse this class.

**Example:**

Suppose we Shape class, if we must store the shape data in a database. If we put this responsibility in Shape class, there will be many database related operations thus making Shape incohesive. So, create a fabricated class DBStore which is responsible to perform all database operations. Similarly logInterface which is responsible for logging information is also a good example for Pure Fabrication.

1. **Indirection :**

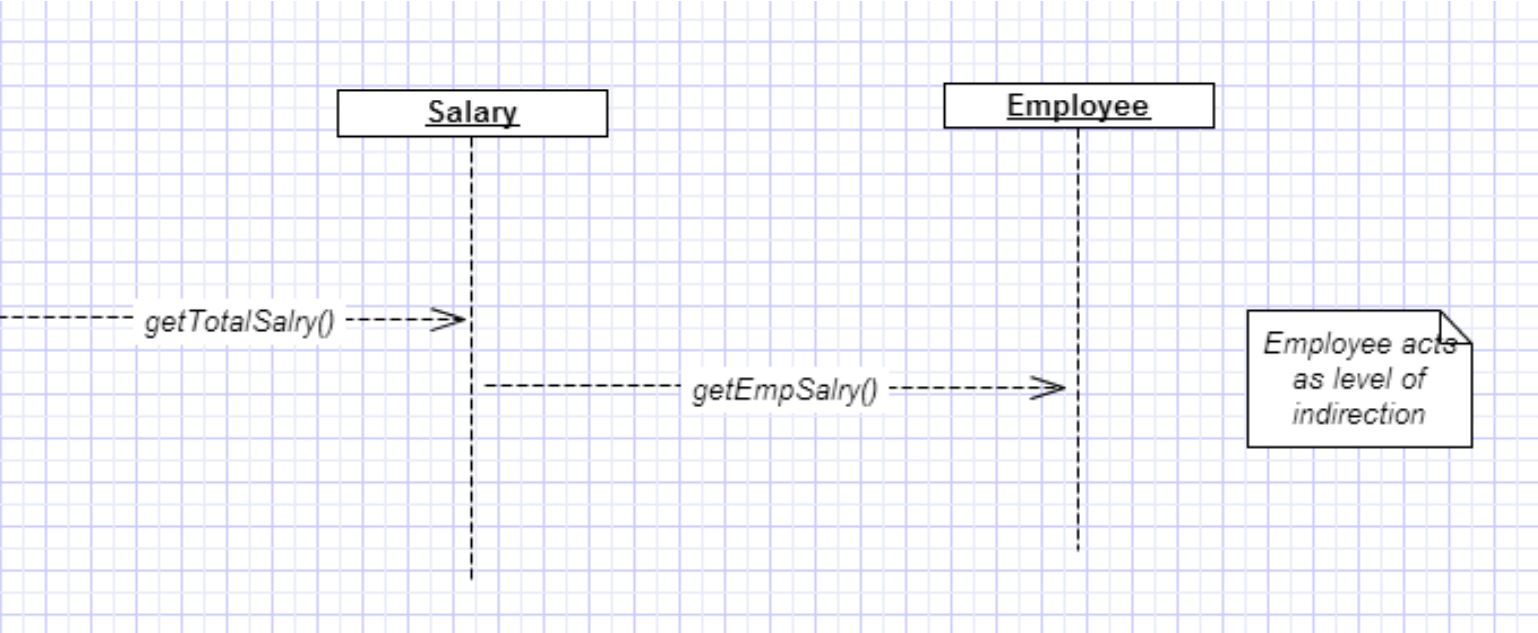
Indirection introduces an intermediate unit to communicate between the other units, so that the other units are not directly coupled.

● Benefits: low coupling

● Example: Adapter, Facade, Obserever

**Example :**

Here polymorphism illustrates indirection .Class Employee provides a level of indirection to other units of the system.

****

1. **Write example implementations of the following creational patterns: Abstract Factory, Factory Method, and Singleton.**

**---🡪** Factory Method:

Factory pattern is one of the most used design patterns in Java. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object. In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.

Example:

We're going to create a Shape interface and concrete classes implementing the Shape interface. A factory class ShapeFactory is defined as a next step. FactoryPatternDemo, our demo class will use ShapeFactory to get a Shape object. It will pass information (CIRCLE / RECTANGLE / SQUARE) to ShapeFactory to get the type of object it needs.



//Create an interface

**public** **interface** Shape {

**void** draw();

}

//Create concrete classes implementing the same interface.

**public** **class** Rectangle **implements** Shape {

@Override

**public** **void** draw() {

System.***out***.println("Inside Rectangle::draw() method.");

}

}

**public** **class** Square **implements** Shape {

@Override

**public** **void** draw() {

System.***out***.println("Inside Square::draw() method.");

}

}

**public** **class** Circle **implements** Shape {

@Override

**public** **void** draw() {

System.***out***.println("Inside Circle::draw() method.");

}

}

//Create a Factory to generate object of concrete class based on given information.

**public** **class** ShapeFactory {

//use getShape method to get object of type shape

**public** Shape getShape(String shapeType){

**if**(shapeType == **null**){

**return** **null**;

}

**if**(shapeType.equalsIgnoreCase("CIRCLE")){

**return** **new** Circle();

} **else** **if**(shapeType.equalsIgnoreCase("RECTANGLE")){

**return** **new** Rectangle();

} **else** **if**(shapeType.equalsIgnoreCase("SQUARE")){

**return** **new** Square();

}

**return** **null**;

}

}

//Use the Factory to get object of concrete class by passing an information such as type

**public** **class** FactoryPatternDemo {

**public** **static** **void** main(String[] args) {

ShapeFactory shapeFactory = **new** ShapeFactory();

//get an object of Circle and call its draw method.

Shape shape1 = shapeFactory.getShape("CIRCLE");

//call draw method of Circle

shape1.draw();

//get an object of Rectangle and call its draw method.

Shape shape2 = shapeFactory.getShape("RECTANGLE");

//call draw method of Rectangle

shape2.draw();

//get an object of Square and call its draw method.

Shape shape3 = shapeFactory.getShape("SQUARE");

//call draw method of square

shape3.draw();

}

}

2. **Abstract Factory:**

Abstract Factory patterns work around a super-factory which creates other factories. This factory is also called as factory of factories. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

In Abstract Factory pattern an interface is responsible for creating a factory of related objects without explicitly specifying their classes. Each generated factory can give the objects as per the Factory pattern.

We are going to create a Shape interface and a concrete class implementing it. We create an abstract factory class AbstractFactory as next step. Factory class ShapeFactory is defined, which extends AbstractFactory. A factory creator/generator class FactoryProducer is created.AbstractFactoryPatternDemo, our demo class uses FactoryProducer to get a AbstractFactory object. It will pass information (CIRCLE / RECTANGLE / SQUARE for Shape) to AbstractFactory to get the type of object it needs.



//Create an interface for Shapes.

**public** **interface** Shape {

**void** draw();

}

//Create concrete classes implementing the same interface.

**public** **class** RoundedRectangle **implements** Shape {

@Override

**public** **void** draw() {

System.***out***.println("Inside RoundedRectangle::draw() method.");

}

}

**public** **class** RoundedSquare **implements** Shape {

@Override

**public** **void** draw() {

System.***out***.println("Inside RoundedSquare::draw() method.");

}

}

**public** **class** Rectangle **implements** Shape {

@Override

**public** **void** draw() {

System.***out***.println("Inside Rectangle::draw() method.");

}

}

//Create an Abstract class to get factories for Normal and Rounded Shape Objects.

**public** **abstract** **class** AbstractFactory {

**abstract** Shape getShape(String shapeType) ;

}

//Create Factory classes extending AbstractFactory to generate object of concrete class based on given information.

**public** **class** ShapeFactory **extends** AbstractFactory {

@Override

**public** Shape getShape(String shapeType){

**if**(shapeType.equalsIgnoreCase("RECTANGLE")){

**return** **new** Rectangle();

}**else** **if**(shapeType.equalsIgnoreCase("SQUARE")){

**return** **new** Square();

}

**return** **null**;

}

}

**public** **class** RoundedShapeFactory **extends** AbstractFactory {

@Override

**public** Shape getShape(String shapeType){

**if**(shapeType.equalsIgnoreCase("RECTANGLE")){

**return** **new** RoundedRectangle();

}**else** **if**(shapeType.equalsIgnoreCase("SQUARE")){

**return** **new** RoundedSquare();

}

**return** **null**;

}

}

//Create a Factory generator/producer class to get factories by passing an information such as Shape

**public** **class** FactoryProducer {

**public** **static** AbstractFactory getFactory(**boolean** rounded){

**if**(rounded){

**return** **new** RoundedShapeFactory();

}**else**{

**return** **new** ShapeFactory();

}

}

}

//Use the FactoryProducer to get AbstractFactory in order to get factories of concrete classes by passing an information such as type.

**public** **class** AbstractFactoryPatternDemo {

**public** **static** **void** main(String[] args) {

//get shape factory

AbstractFactory shapeFactory = FactoryProducer.*getFactory*(**false**);

//get an object of Shape Rectangle

Shape shape1 = shapeFactory.getShape("RECTANGLE");

//call draw method of Shape Rectangle

shape1.draw();

//get an object of Shape Square

Shape shape2 = shapeFactory.getShape("SQUARE");

//call draw method of Shape Square

shape2.draw();

//get shape factory

AbstractFactory shapeFactory1 = FactoryProducer.*getFactory*(**true**);

//get an object of Shape Rectangle

Shape shape3 = shapeFactory1.getShape("RECTANGLE");

//call draw method of Shape Rectangle

shape3.draw();

//get an object of Shape Square

Shape shape4 = shapeFactory1.getShape("SQUARE");

//call draw method of Shape Square

shape4.draw();

}

}

**3.Singleton.:**

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 ways to create an object.This pattern involves a single class which is responsible to create an object while making sure that only single object gets 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.

We are going to create a SingleObject class. SingleObject class have its constructor as private and have a static instance of itself.SingleObject class provides a static method to get its static instance to outside world. SingletonPatternDemo, our demo class will use SingleObject class to get a SingleObject object.



//Create a Singleton Class.

**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!");

}

}

//Get the only object from the singleton class.

**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();

}

}