|  |  |
| --- | --- |
|  | Master of Applied Computing  COMP-8117  Advanced Software Engineering Topics |
| School of Computer Science <https://cs.uwindsor.ca> | Dr. Usama Mir – [Usama.Mir@uwindsor.ca](mailto:Usama.Mir@uwindsor.ca)  Office Hours: Wednesday and Thursday (10.00am – 1.00pm EST) |

**Assignment/Lab 2 – GRASP and GoF Design Patterns**

* **Total Marks = 60 (will be adjusted to 3% accordingly)**
* **Submission Date = July 06, 2022**
* **Submission = Only via blackboard**
* **One submission per group**
* **Only one attempt**
* **Copying from Internet and other sources will result in 0 marks**
* **Grading will be done in class for each student after the due date**
* **Submission Format = MS Word**
* **No screenshots allowed. Against each design pattern, the code should be put in the Word file. Ex.**

**Singleton:**

**Code:**

class Singleton {

private static Singleton instance

#This is the constructor which is private and does not allow any instance of this class to be created from outside

private Singleton ()

{

}

public static Singleton getinstance ()

{

if (instance == null)

{

instance = new Singleton()

}

return instance

}

.........

}

**Purpose**

The purpose of this activity is to check your understanding of GRASP (General Responsibility Assignment Software Patterns) and GoF (Gang of Four) design patterns studied in lectures 4 and 5, respectively. You may use Java or Python as your coding languages. To further strengthen your concepts, you may refer to the material available on BB under the lecture slides and notes section. Please keep in mind that only one coding language should be used for the whole activity. For example, if you select “Java” as your preferred language, then you must provide the examples for all the patterns using Java.

**Assignment/Lab Tasks**

In this activity, you must do the following:

**Task 1** – For the following GRASP patterns:

Information Expert, Creator, Controller, Low Coupling, High Cohesion, and Polymorphism

* Explain each pattern’s purpose in your own words
* Provide and explain an example
* Code the example using Java or Python

**(Marks = 5 X 6 = 30)**

**Task 2** - Repeat the activities in Task 1 with the following GoF Patterns:

Singleton, Decorator, Prototype, Composite, Adapter, and Facade

**(Marks = 5 X 6 = 30)**

**ASET ASSIGNMENT-1, Section-1, Group-2**

**Team Members:**

1.Arun Reddy Nalla (Student ID: 110088379)

2.Harsh Bharath Nimmakuri (Student ID: 110088953)

3.Kalyan Venkatesh Poludasu (Student ID: 110089120)

4.Veera Venkata Bharath Kumar Vayitla (Student ID:110088432)

**GRASP patterns:**

**Information Expert:**

In an information expert design pattern, we must decide where to place responsibilities like methods, variables etc. let’s say we have a method to place in a certain class where we have multiple classes now, we have to analyze and look for the best appropriate class and place that method in that class.

**Example Code**:

In the below example we have a bookstore scenario where we have book class, bookstore class, main class. Now we have to add getAllBooks method to this example and have to choose the right fit to place that method. I felt the bookstore class is the right fit to add that method, as all the methods related to book operations are in that class.

//class book

package ase;

public class Book {

private int bookId;

private String bookName;

public Book(int bookId, String bookName) {

super();

this.bookId = bookId;

this.bookName = bookName;

}

public int getBookId() {

return bookId;

}

public void setBookId(int bookId) {

this.bookId = bookId;

}

public String getBookName() {

return bookName;

}

public void setBookName(String bookName) {

this.bookName = bookName;

}

//information expert

package ase;

import java.util.ArrayList;

import java.util.List;

public class BookStore {

public List<Book> getAllBooks()

{

List<Book> book=new ArrayList<Book>();

Book b1=new Book(1,"book1");

Book b2=new Book(2,"book2");

Book b3=new Book(3,"book3");

Book b4=new Book(4,"book4");

book.add(b1);

book.add(b2);

book.add(b3);

book.add(b4);

return book;

}

public int getNumberOfbooks()

{

int noOfBooksCount= getAllBooks().size();

return noOfBooksCount;

}

}

//main class

package ase;

import java.util.List;

public class BookMain {

public static void main(String[] args) {

BookStore bookStore= new BookStore();

List<Book> allBooks=bookStore.getAllBooks();

int noOfbooksCount=bookStore.getNumberOfbooks();

System.out.println("Total number of books : "+noOfbooksCount);

for(int i=0;i<bookStore.getAllBooks().size();i++)

{

System.out.println(allBooks.get(i).getBookId()+" "+allBooks.get(i).getBookName());

}

}

**Creator:**

* Creator is responsible for creating a new instance of a class.
* In simple terms, the one who create an object or a new instance of class
* In this pattern, we have creator class which create a new instance of other classes. Without this creator class other classes can’t be created

**Example Code**:

import java.util.HashMap;

public class CREATOR {

static HashMap<String, Double> hmap = new HashMap<String, Double>();

public static HashMap<String, Double> addItem(String prodSpec,double quantity)

{

hmap.put(prodSpec, quantity);

return hmap;

}

public static void main(String[] args) {

System.out.println(addItem("chocolate", 1.5));

}

}

From the above code, creator class is responsible for instance of addItem

**Controller:**

Controller is responsible for handling the UI event from the user and organize/invoke the necessary operations in the backend system.

ex: In the below example, the ATC acts as a controller invoking each method depending upon the Airplane requests

Diagram

Description automatically generated

**Example Code**:

*<< ATC.java >>*

public class ATC {

private String[][] arrival\_runways = {{"r1", "r2"}, {"a", "a"}};

private String[][] takeoff\_runways = {{"r3", "r4"}, {"a", "a"}};

public void runway\_req(String name, String status) {

if (status == "arrival") {

arrival\_check(name, status);

}

else {

takeoff\_check(name, status);

}

}

private void arrival\_check(String name, String status) {

boolean alloted = false;

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

if (arrival\_runways[1][i] == "a") {

show\_status(arrival\_runways[0][i], name, status);

arrival\_runways[1][i] = "o";

alloted = true;

break;

}

}

if(!alloted){

System.out.println("Runways unavilable for " + status + ", " + name + " Please hold!");

}

}

private void takeoff\_check(String name, String status) {

boolean alloted = false;

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

if (takeoff\_runways[1][i] == "a") {

show\_status(takeoff\_runways[0][i], name, status);

takeoff\_runways[1][i] = "o";

alloted = true;

break;

}

}

if(!alloted){

System.out.println("Runways unavilable for " + status + ", " + name + " Please hold!");

}

}

public void show\_status(String runway, String name, String status) {

System.out.println(runway + " allocated for " + name + " " + status);

}

}

*<< Airplane.java >>*

public class Airplane {

private String flight\_no = null;

private String req = null;

private ATC atc;

public Airplane() {

atc = new ATC();

}

public void get\_status(String name, String req) {

atc.runway\_req(name, req);

}

public String getFlight\_no() {

return flight\_no;

}

public void setFlight\_no(String flight\_no) {

this.flight\_no = flight\_no;

}

public String getReq() {

return req;

}

public void setReq(String req) {

this.req = req;

}

public static void main(String[] args) {

// TODO Auto-generated method stub

Airplane f1 = new Airplane();

Airplane f2 = new Airplane();

f1.setFlight\_no("A352"); f1.setReq("arrival");

f2.setFlight\_no("S2301"); f2.setReq("takeoff");

// requesting ATC for run ways status

f1.get\_status(f1.getFlight\_no(), f1.getReq());

f2.get\_status(f2.getFlight\_no(), f2.getReq());

f1.get\_status(f1.getFlight\_no(), f1.getReq());

}

}

**Low Coupling:**

· Coupling is the degree in which one element is interconnected or dependent upon another element.

· Coupling should be low so that one element doesn't depend on too many other elements.

· We can easily make changes on an element because changes made to an element doesn’t impact remaining elements.

· Since our modules are not interdependent on one another it is easier to develop, design, and test code and, we can easily reuse our code.

**Example Code**:

· Consider a person looking for accommodation and willing to pay rent for accommodation.

· The person is having many options to look for an accommodation. Its either house, apartment...etc.

· We are implementing low coupling in our program so that each accommodation is independent of another.

· We have a rent Interface with display method in it.

· Let us create two classes, House and Apartment which implements the rent Interface.

· We passed House and Apartment objects as references rather than directly as objects.

· We pass the necessary object to carry out the process . In this instance, we don't need to make many changes to the code even if the requirement changes in the future.

· Think about if we'll need to add more accommodations, like hostels, as needs alter in the future. We can easily alter the code now without undergoing major change.

interface rent

{

public void display();

}

class House implements rent

{

@Override

public void display()

{

System.out.println("This is House");

}

}

class Apartment implements rent

{

@Override

public void display()

{

System.out.println("This is Apartment");

}

}

public class Accommodation

{

rent r;

public Accommodation(rent r)

{

this.r=r;

}

public void display()

{

r.display();

}

public static void main(String[] args)

{

House h=new House();

Apartment a=new Apartment();

Accommodation r1=new Accommodation(h);

Accommodation r2=new Accommodation(a);

r1.display();

r2.display();

}

}

**Output:**

This is House

This is Apartment

**High cohesion:**

High cohesion is an evaluative pattern that tries to keep objects focused, manageable, and understandable.

In general, high cohesion is used to support low coupling.

**Example Code**:

import java.util.Arrays;

import java.util.List;

class course {

String name;

int id;

int classroom;

public List<Object> getDetails(int id,String name, int classroom)

{

this.name = name;

this.id=id;

this.classroom=classroom;

return Arrays.asList(id, name, classroom);

}

}

class review {

int rating;

public int getRating(int rating)

{

this.rating = rating;

return rating;

}

}

public class highCohesion {

public static void main(String[] args)

{

course n = new course();

List<Object> c = n.getDetails(8110, "java", 101);

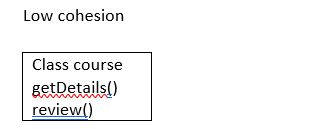
System.out.println(c);

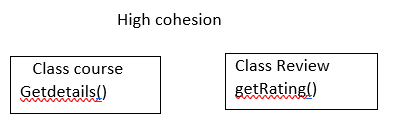
review a = new review();

System.out.println(a.getRating(5));

}

}





In the above image, we can see that in low cohesion, only one class is responsible all code that are not shared, reducing the possibility of reusability and maintenance. In high cohesion, however, there is a separate class for review and course details to execute, resulting in improved usability and maintenance.

**Polymorphism**:

Simply we can define polymorphism as existing in many forms. If we have polymorphic classes, if the method does not vary across the classes, then we can place that method in the parent class and in the child class we can override that method based on child class requirement. If the method varies or specific to some class, then place that method in the related class.

**Example Code**:

//parent class

class Car {

public void displayVersionInfo() {

System.out.println("this is basic version of car");

}

}

//child class

class CarHighEnd extends Car {

@Override

public void displayVersionInfo() {

System.out.println("this is Hign-end version of this car");

}

}

class Main {

public static void main(String[] args) {

// create an object of Car class

CarHighEnd v2 = new CarHighEnd();

v2.displayVersionInfo();

// create an object of CarHighEnd class

Car v1 = new Car();

v1.displayVersionInfo();

}

}

**Gang Of Four:**

**Singleton:**

The singleton pattern guarantees that there is only one instance of the class and offers a single point of access to it globally. To achieve a singleton pattern, we have to follow the following steps:

1. Create a private static instance variable

2. Make the constructor private, which blocks other classes to create an instance of this class.

3. Create a public static method to return an instance of this class.

**Example Code**:

package ase;

//singleton class

public class Singleton {

// creating private static instance

private static Singleton singleInstance= new Singleton();

//making constructor as private

private Singleton()

{}

//public static method to return instance

public static Singleton getSingleInstance()

{

return singleInstance;

}

}

//main class

package ase;

public class SingletonExampleGOF {

public static void main(String[] args) {

Singleton object1= Singleton.getSingleInstance();

Singleton object2= Singleton.getSingleInstance();

//we will get the same hashcode as they point to same instance

System.out.println("hashcode of object1 : "+object1.hashCode());

System.out.println("hashcode of object2 : "+object2.hashCode());

}

}

**Decorator:**

One of the most common structural patterns is the decorator pattern.

This pattern changes the functionality of an object dynamically at runtime without affecting the objects existing functionality.

In short, by wrapping the object, this pattern adds additional functionalities to it.

**Example Code**:

Bike.java

The interface class defining the methods used in our program

package decorator;

public interface bike {

public void assemble();

}

BasicBike

BasicBike is an basic implementation for the bike interface method

package decorator;

public class BasicBike implements bike {

@Override

public void assemble() {

System.out.print("Basic Bike.");

}

}

BikeDecorator

BikeDecorator class implements bike (interface) . decorator child classes has access to it.

package decorator;

public class BikeDecorator implements bike {

protected bike bike;

public BikeDecorator(bike b){

this.bike=b;

}

@Override

public void assemble() {

this.bike.assemble();

}

}

ElectricBike.java

Extending the BikeDecorator and changing the functionalities according to ElectricBike

package decorator;

public class ElectricBike extends BikeDecorator {

public ElectricBike(bike b) {

super(b);

}

@Override

public void assemble(){

super.assemble();

System.out.print(" Electric bike");

}

}

SportsBike.java

Extending the BikeDecorator and changing the functionalities according to SportsBike

package decorator;

public class SportsBike extends BikeDecorator {

public SportsBike(bike b) {

super(b);

}

@Override

public void assemble(){

super.assemble();

System.out.print(" Sports bike");

}

}

Bikes.java

package decorator;

public class Bikes {

public static void main(String[] args) {

// TODO Auto-generated method stub

System.out.println("model 1");

bike sportsBike = new SportsBike(new BasicBike());

sportsBike.assemble();

System.out.println("\nmodel 2");

bike electricBike = new ElectricBike(new BasicBike());

electricBike.assemble();

System.out.println("\nmodel 3");

bike SportsElectricbike = new SportsBike(new ElectricBike(new BasicBike()));

SportsElectricbike.assemble();

}

}

**Prototype:**

· Prototype is creational GoF Pattern because this pattern offers best approach to create an object.

· This pattern is used to copy an existing object(cloning) instead of creating new instance from beginning.

· This pattern is used when directly creating an object is expensive. It saves cost as well as time.

**Example Code**:

· Consider example of replanting a tree of same species

· We are here creating a new plant object of same species from the existing plant object of same species.

· The copy() method is implemented by the Plant class. This method duplicates the species field and generates a new Plant object. The newly cloned Plant object is returned.

interface clone

{

public clone copy();

}

class Plant implements clone

{

String species;

public Plant(String s)

{

this.species=s;

}

@Override

public clone copy()

{

return new Plant(species);

}

public String toString()

{

return "Plant planted is " + species;

}

}

public class Prototype

{

public static void main(String[] args)

{

Plant plant1=new Plant("Mango");

System.out.println("plant 1:"+ plant1);

Plant plant2=(Plant)plant1.copy();

System.out.println("plant 2:"+plant2);

}

}

**Output:**

plant 1:Plant planted is Mango

plant 2:Plant planted is Mango

**Composite:**

· Composite is structural GoF Pattern because this pattern creates group of elements into a tree structure.

· This pattern is used to treat group of objects that are handled in the same manner as a single object.

· These structures can be handled as if they were separate objects and ask each node to perform a task.

**Example Code**:

· Consider we want to develop a hierarchical structure of each department in a university.

· In this example MAC and CS departments are leaf components.

· Head Department is the composite class since it contains methods for adding elements as well as a collection of Department component parts.

· By iterating through the list of leaf elements and calling the proper method for each one, the composite printDepartmentName() function is implemented.

import java.util.\*;

interface Department

{

void printDepartmentName();

}

class MAC implements Department

{

public void printDepartmentName()

{

System.out.println(getClass().getSimpleName());

}

}

class CS implements Department

{

public void printDepartmentName()

{

System.out.println(getClass().getSimpleName());

}

}

class HeadDepartment implements Department

{

public Integer id;

public String name;

public List<Department> childDepartments;

public HeadDepartment(Integer id, String name)

{

this.id=id;

this.name=name;

this.childDepartments=new ArrayList<>();

}

public void printDepartmentName()

{

childDepartments.forEach(Department::printDepartmentName);

}

public void addDepartment(Department department)

{

childDepartments.add(department);

}

}

public class Composite

{

public static void main(String args[])

{

Department mac=new MAC();

Department cs=new CS();

HeadDepartment headDepartment=new HeadDepartment(1,"Head department");

headDepartment.addDepartment(mac);

headDepartment.addDepartment(cs);

headDepartment.printDepartmentName();

}

}

**Output:**

MAC

CS

**Adapter:**

It a Structural Design Pattern which acts as a connector between two objects of different interfaces collaborate.

One of the real world example is of Travel Adapter, which help in charging devices with different sockets.

Adaper Design pattern mainly has 5 components

|  |  |
| --- | --- |
| Target Interface | main interface to be used by client |
| Adaptee interface | interface to which the adapter connects target interface |
| Adapter class | This class is a wrapper class which implements the desired target interface and modifies the specific request available from the Adaptee class |
| Adaptee class | the class which is used by the Adapter class to reuse the existing functionality and modify them for desired use |
| Client | the class which will interact with adapter class |

Diagram

Description automatically generated

**Adapter Example**

**Example Code**:

in the above example we have:

1. Target interface : battery\_tank

2. Adaptee interface : fuel\_tank

3. Adapter class : Fuel\_Adapter

4. Adaptee class : Diesel\_car

5. Client : EV\_Adapter

*<<EV\_Adapter.java>>*

interface fuel\_tank{

// adaptee interface

public void capacity();

public void type\_of\_fuel();

}

class Diesel\_car implements fuel\_tank{

public void capacity() {

System.out.println("tank capacity is 22l of diesel");

}

public void type\_of\_fuel() {

System.out.println("Fuel type: Diesel");

}

}

interface battery\_tank{

// target interface

public void kWh();

public void type\_of\_input();

}

class Hybrid\_car implements battery\_tank{

public void kWh() {

System.out.println("Full EV capacity if 60kWh");

}

public void type\_of\_input() {

System.out.println("Fuel Type: Electric Battery");

}

}

class Fuel\_Adapter implements battery\_tank{

Diesel\_car temp;

// ref to the object we're trying to adapt

public Fuel\_Adapter(Diesel\_car temp) {

this.temp = temp;

}

//translate the methods

public void kWh() {

temp.capacity();

}

public void type\_of\_input() {

temp.type\_of\_fuel();

}

}

public class EV\_Adapter {

public static void main(String args[]) {

Diesel\_car fiat = new Diesel\_car();

fiat.capacity();

fiat.type\_of\_fuel();

battery\_tank lumo = new Hybrid\_car();

lumo.kWh();

lumo.type\_of\_input();

//out of EV stations so need to switch to Diesel tank

battery\_tank lumos = new Fuel\_Adapter(fiat);

lumos.kWh();

lumos.type\_of\_input();

}

}

**Facade:**

It is a Structural Design Pattern which provides a simple interface to the client over a complex subsystem.

Its main objective is to minimize the dependence on subsystem

Ex. A Customer Service Representative helping a customer/user in ordering items, billing process etc of a large supermarket/warehouse without disclosing the complexities.

Diagram

Description automatically generated

**Example Code**:

In the above example, we’ve the ‘Settings\_Facade’ class providing a simple interface for all the different setting classes.

*<< Settings\_app.java>>*

interface settings{

public void show\_available\_settings();

}

class Camera implements settings{

public void show\_available\_settings(){

System.out.println("Record video format\nPicture mode\nFilters");

}

}

class Display implements settings{

public void show\_available\_settings(){

System.out.println("Display Resolution\nDisplay color format\nWallpaper");

}

}

class App implements settings{

public void show\_available\_settings(){

System.out.println("Uninstall\nmodify Permissions\nChange icon");

}

}

class Files implements settings{

public void show\_available\_settings(){

System.out.println("Delete files\nSort\nRefresh");

}

}

class Clock implements settings{

public void show\_available\_settings(){

System.out.println("Adjust time\nchange TimeZone\nset alarm");

}

}

class Settings\_Facade {

private Camera c;

private Display d;

private App a;

private Files f;

private Clock cl;

public Settings\_Facade() {

c = new Camera();

d = new Display();

a = new App();

f = new Files();

cl = new Clock();

}

public void camera\_settings() { c.show\_available\_settings(); }

public void display\_settings() { d.show\_available\_settings(); }

public void app\_settings() { a.show\_available\_settings(); }

public void files\_settings() { f.show\_available\_settings(); }

public void clock\_settings() { cl.show\_available\_settings(); }

}

public class Settings\_app{

public static void main(String args[]) {

Settings\_Facade client\_1 = new Settings\_Facade();

Settings\_Facade client\_2 = new Settings\_Facade();

Settings\_Facade client\_3 = new Settings\_Facade();

client\_1.camera\_settings(); client\_2.app\_settings(); client\_3.clock\_settings();

}

}