Multiple choice questions. Each question is worth 3 points.

1. Which of the below is not a valid design pattern?
   a) Singleton
   b) Composite
   c) Visitor
   d) Java

2. The general purpose of the Singleton pattern is to:
   a) Ensure that no more than one instance of a class exists.
   b) Ensure that only one instance of a class exists at the same time.
   c) Separate objects in a single class from objects in another class.
   d) Control creation of objects in a single class or another class.

3. The consequences of a design pattern are:
   a) The pitfalls of using the particular pattern
   b) The results of choosing a pattern
   c) The time a program using the pattern takes to run
   d) The time it takes to design a program using the pattern

4. A design pattern is:
   a) an algorithm used in object-oriented programming
   b) a data structure used in object-oriented programming
   c) a solution to a common problem in object-oriented programming
   d) a blueprint for a particular kind of class

5. Design pattern is to program as:
   a) Metal is to car
   b) Park is to tree
   c) Blueprint is to be building
   d) Mountain Dew is to soda

6. Most object-oriented programming provides which of these ways to create new objects?
   a) Instantiating a class using one of its constructors
   b) Cloning an existing object
   c) All of the mentioned
   d) None of the mentioned

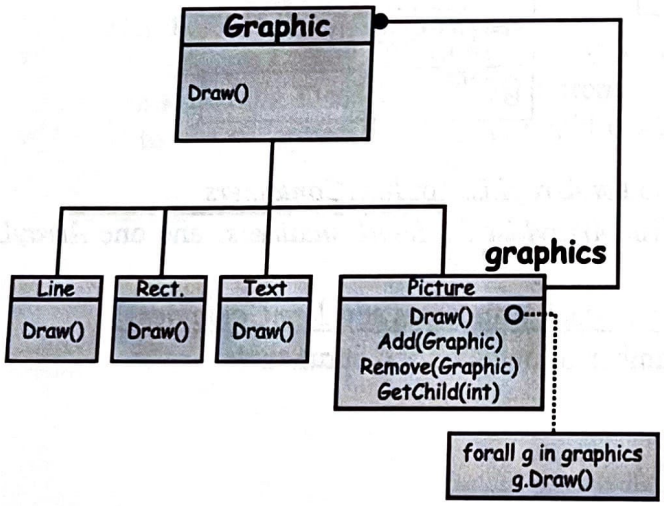
7. Module design is used to maximize cohesion and minimize coupling which of the following is the key to implement this rule?
   a) Inheritance
   b) Polymorphism
   c) Encapsulation
   d) Abstraction

8. Which of the following are true for the singleton class?
   a) Singleton classes should be used whenever it is important that only a single instance of a class exist and that that single instance be widely accessible
   b) The Singleton pattern can also be used, with slight modifications, when a limited number of instances greater than one are desired
   c) Access restrictions are usually easy to add by restricting the visibility of either the class or the factory method
   d) All of the mentioned

9. Which of the following describes the Adapter pattern correctly?
   a) This pattern builds a complex object using simple objects and using a step-by-step approach.
   b) This pattern refers to creating duplicate object while keeping performance in mind.
   c) This pattern works as a bridge between two incompatible interfaces.
   d) This pattern is used when we need to decouple an abstraction from its implementation so that the two can vary independently.

10. What is incorrect among the following?
   a) Make use cases that uniform in size and complexity
   b) Organize use cases by actor, problem domain categories or solution categories
   c) Use cases can last for more than one session
   d) Achieve a stakeholder goal in a use case

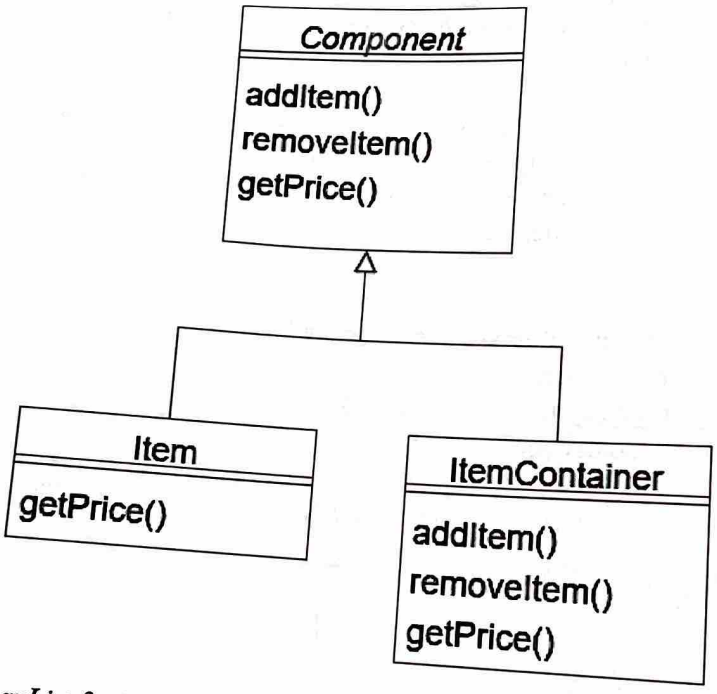
11. Class diagram below represents implementation of which design pattern?



a) Composite – Pro-transparency approach
b) Decorator
c) Visitor
d) Adapter
e) Composite – Pro-safety approach

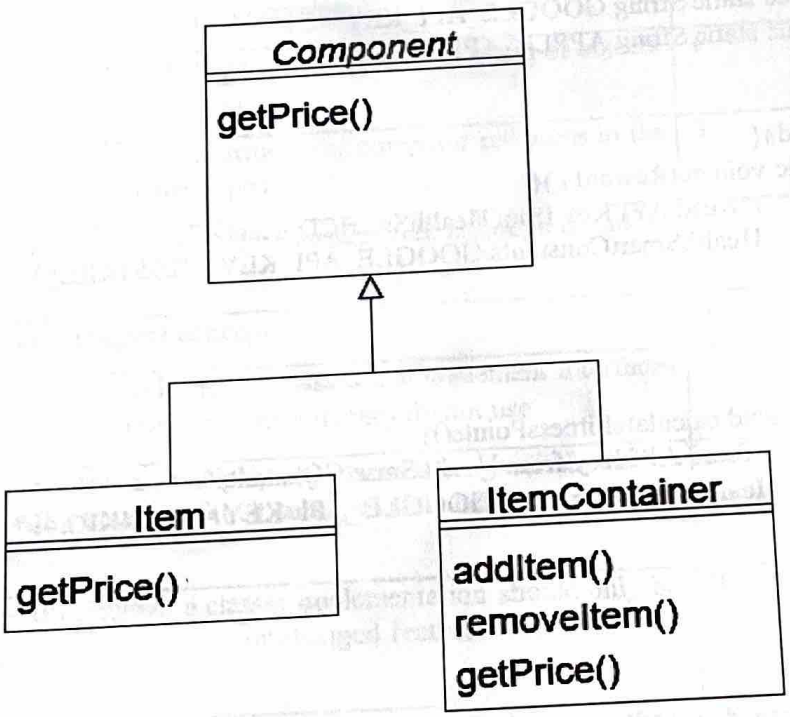
12. Which design patterns converts the interface of a class into another interface clients expect?
    a) Visitor
    b) Decorator
    c) Adapter
    d) Composite

13. For Composite Pattern. In *ItemContainer* class (Composite class) how many *ArrayLists* do we need to manage the children of each *ItemContainer* object?



a) Two - one *ArrayList* for Items and one for *ArrayList* for *ItemContainers*
b) three - one *ArrayList* for Items, one for *ArrayList* for *ItemContainers*, and one *ArrayList* for Components
c) one - one *ArrayList* for Component [can store both Items and *ItemContainers*]
d) cannot determine - depends on the number of items added at run time

14. Which statements are correct about the class design below? [Select all that apply]



a) This is implementing composite pattern
b) Items and Item Containers are treated differently. This design will force us to write if conditions to check the object type
c) Design is violating Liskov Substitution Principle (LSP)
d) This is implementing Decorator design pattern
e) We have to use exception handling to avoid calling add() and remove() methods on an Item object

15. What's wrong with the code below?

```
public class HealthSmartConstants{
        public static String GOOGLE_API_KEY = "";
        public static String APPLE_API_KEY = "";
}

class Rewards{
        public void getRewards(){
                /*Read API Key from HealthSmartCOnstants*/
                HealthSmartConstants.GOOGLE_API_KEY = "0954KRE45"
        }
}

class Fittness{
        public void calculateFitnessPoints(){
                /*Read API Key from HealthSmartCOnstants*/
                HealthSmartConstants.GOOGLE_API_KEY = "0954KRE49"
        }
}
```

a) Data structure is passed as a parameter, but the called module operates on some but not all of the individual components
b) One class is passing element of control to another
c) Nothing wrong. Classes are not coupled with one another
d) Two classes are having write access to Global Data

TRUE/FALSE Questions (Each question is worth 1.5 point)	True	False
16. Operations in a UML class diagram are typically represented as methods in object-oriented programming.	T	
17. Composite pattern is used where we need to treat a group of objects in similar way as a single object.	T	
18. Abstraction is a technique for structuring computer programs in the simplest/cleanest components possible?	T	
19. An abstract class is a class that be instantiated, although it can be used as a base class.		F
20. Is design pattern a logical concept?	T	
21. In Interface Segregation Principle, classes that implement interfaces should be forced to implement methods they do not use.		F
22. In Liskov Substitution Principle, objects in a program should be replaceable with instances of their subtypes and alter the correctness of that program.		F
23. In Open/Closed Principle, a classes implementation should only be modified to correct errors; new or changed features would require a subclass be created.	T	
24. The critique of the Single Responsibility may make code more complex and decomposed for an eventuality that never comes.	T	
25. In Dependency Inversion Principle, Abstractions should depend on details; details should not depend on abstractions.		F

26. **Match the following Cohesion types with their description (7.5 points)**

Cohesion	Description
1. Coincidental cohesion	a. elements perform logically related tasks
2. Logical cohesion	b. elements share I/O
3. Temporal cohesion	c. elements cooperate to carry out a single function
4. Communication cohesion	d. elements must be used at approximately the same time
5. Functional cohesion	e. elements are in the same module for no particular reason

1 E 2 A 3 D 4 B 5 C

Answer:

- Coincidental cohesion [elements are in the same module for no particular reason]
- Logical cohesion [elements perform logically related tasks]
- Temporal cohesion [elements must be used at approximately the same time]
- Communication cohesion [elements share I/O]
- Functional cohesion [elements cooperate to carry out a single function]

27. **Match the following Coupling types with their description (7.5 points)**

Coupling	Description
1. Content coupling	a. Modules exchanging elements, but the receiving end doesn't act on all elements. For example, a module receiving an array via its interface but not using all its elements
2. Common coupling	b. Modules exchanging elements, and the receiving end use all of them
3. Control coupling	c. Modules directly can access the content of each other's, without using an interface.
4. Stamp coupling	d. Modules controlling the logic (control flow) of other ones
5. Data coupling	e. Modules changing common variables with bigger scope (like global variables).

1 C 2 E 3 D 4 A 5 B

- Content coupling - Modules directly can access the content of each others, without using an interface.
- Common coupling - Modules changing common variables with bigger scope (like global variables).
- Control coupling - Modules controlling the logic (control flow) of other ones.
- Stamp coupling - Modules exchanging elements, but the receiving end doesn't act on all elements. For example, a module receiving an array via its interface but not using all its elements.
- Data coupling - Modules exchanging elements, and the receiving end use all of them.

28. **Complete the getInstance() part if the code snippet below to implement Singleton design pattern (Grad: 3, Undergrad: 5 points).**

```
/*TODO: Add Code for getInstance method*/
```

```
/*TODO: Add Code for getInstance method*/
Lazy:
public static Dashboard getInstance() {
  if (instance == null) {
   instance = new Dashboard();
  }
  return instance;
}
Eager:
private Dashboard instance = new Dashboard();
public static Dashboard getInstance() {
  return instance;
}
```

29. Anything wrong with the class below. If yes, then correct it. (Grad: 4, Undergrad: 5)

```
public class Tractor{
    //Tractor variables
    private String make;
    private String model;
    private double horsePower;
    //Constructor
    public Tractor(String ipMake, String ipModel, double ipHorsePower){
        make = ipMake;
        model = ipModel;
        horsePower = ipHorsePower;
    }

    public String getMake(){
        return make;
    }

    public void setMake(String ipMake){
        make = ipMake;
    }

    public String getModel(){
        return model;
    }

    public void setModel(String ipModel){
        make = ipModel;
    }

    public String getHorsePower(){
        return horsePower;
    }

    public void setHorsePower(String ipHorsePower){
        make = ipHorsePower;
    }
}
```

Ans: Nothing wrong. Class is performing a number of actions, each with its own entry point, with independent code for each action

30. Consider the code below implementing the decorator design pattern.

```
public abstract class Beverage {
 String description = "Unknown Beverage";
 public String getDescription() {
  return description;
 }

 public abstract double cost();
}


public abstract class CondimentDecorator extends Beverage {
 public abstract String getDescription();
}

public class DarkRoast extends Beverage {
 public DarkRoast() {
  description = "Dark Roast Coffee";
 }

 public double cost() {
  return .99;
 }
}

public class Mocha extends CondimentDecorator {
 Beverage beverage;

 public Mocha(Beverage beverage) {
  this.beverage = beverage;
 }

 public String getDescription() {
  return beverage.getDescription() + ", Mocha";
 }

 public double cost() {
  return .20 + beverage.cost();
 }
}
```

<Continued on next page...>

```
public class Whip extends CondimentDecorator {
 Beverage beverage;

 public Whip(Beverage beverage) {
  this.beverage = beverage;
 }

 public String getDescription() {
  return beverage.getDescription() + ", Whip";
 }

 public double cost() {
  return .10 + beverage.cost();
 }
}
```

Write code to complete the implementation of Class DecoratorTester. (Grad: 5, Undergrad: 7 points)

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

      /*TODO: Add code to create a dark roast with double mocha and whip cream*/
```

Option 1:

```
Beverage beverage2 = new DarkRoast();
      beverage2 = new Mocha(beverage2);
      beverage2 = new Whip(beverage2);
      beverage2 = new Whip(beverage2);
```

Option 2:

```
Beverage bev = new Whip(new Whip (new Mocha (DarkRoast())));:
```

31. **Open closed Principle requires code to be open for extension and closed for modification. The code snippet does satisfy OCP? If yes, add new feature as an extension. (Grad: 5, Undergrad: 8 points)**

```
public class Vehicle {
  public double calculateValue() {...}
}
public class Car extends Vehicle {
  public double calculateValue() {
    return this.getValue() * 0.8;
}
public class Truck extends Vehicle{
  public double calculateValue() {
    return this.getValue() * 0.9;
}
```

/*TODO: Add code here*/

```
public class Vehicle {
  public double calculateValue() {...}
}
public class Car extends Vehicle {
  public double calculateValue() {
    return this.getValue() * 0.8;
}
public class Truck extends Vehicle{
  public double calculateValue() {
    return this.getValue() * 0.9;
}
```

32. **The code will work, for now, but what if we wanted to add another engine type, let's say a diesel engine? Does this code will require refactoring the Car class? If yes, then refactor it? (Grad: 8 points, Undergrad: 5 (Bonus))**

```
public class Engine {
   public void start() {...}
}

public class Car {
   private Engine engine;
   public Car(Engine e) {
      engine = e;
   }
   public void start() {
      engine.start();
   }
}
```

/*TODO: If yes, add code here*/

```
public interface Engine {
   public void start();
}
public class Car {
   private Engine engine;
   public Car(Engine e) {
      engine = e;
   }
   public void start() {
      engine.start();
   }
}
public class PetrolEngine implements Engine {
  public void start() {...}
}
public class DieselEngine implements Engine {
  public void start() {...}
}
```

Bonus:

33. **The single responsibility principle states that each class should have one responsibility, one single purpose. Does the below class implementation violate Single Responsibility? If yes, how to resolve? (5 points)**

```
public class Vehicle {
    public void printDetails(){}
    public double calculateValue(){}
    public void addVehicleToDB(){}
}
```

/*TODO: If yes, add code here*/

```
public class PrintVehicle {
    public void printDetails(){}
}
public class VehiclePriceCalculation
{
    public double calculateValue(){}
}
public class VehicleInventory {
    public void addVehicleToDB(){}
}
```