Assignment 2.1
CSCI 5448

Jayant Duneja
Jayant.Duneja@colorado.edu

Tilak Singh
Tilak.Singh@colorado.edu

September 20, 2023

Ques 1 (5 points) What does "design by contract" mean in an OO program? What is the difference between implicit and explicit contracts? Provide a text, pseudo code, or code example of each contract type.

Ans. Design by Contract is a software development methodology that emphasizes the creation of explicit and formal agreements (contracts) between software components, which can help the components behave in a predictable manner and help with reliability, debugging and documentation. Essentially, these contracts define the rights and obligations of all components involved in the interaction.[1]

There are two main types of contracts: *implicit contracts* and *explicit contracts*.

A) Implicit contracts

Implicit contracts rely on the understanding of how objects or components should behave within a specific programming language or framework. These can change based on the language or framework that we use.

```
import java.util.*;

//Defining a class Student
class Student {
   private String id;
   private String name;


   //Constructor
   public Student(String id, String name) {
       this.id = id;
       this.name = name;
   }

  //Getters
   public String getId() {
       return id;
   }
   public String getName() {
       return name;
   }

   @Override
   public boolean equals(Object o) {
       if (this == o) return true;
       if (o == null || getClass() != o.getClass()) return false;
       Student student = (Student) o;
       return Objects.equals(id, student.id) &&
               Objects.equals(name, student.name);
```

```
    }

    @Override
    public int hashCode() {
        return Objects.hash(id, name);
    }
}

public class Main {
    public static void main(String[] args) {
        Student student1 = new Student("S001", "Alice");
        Student student2 = new Student("S001", "Alice");

        // Implicit contract: 'equals' method checks for the content equality of
objects
        boolean areEqual = student1.equals(student2);
        System.out.println("Are students equal? " + areEqual); // "true"

        // Implicit contract: 'hashCode' method returns the same value for equal
objects
        System.out.println("Hash code of student1: " + student1.hashCode());
        System.out.println("Hash code of student2: " + student2.hashCode());
    }
}
```

B) Explicit contracts

Explicit contracts, on the other hand, are formally defined in code and are enforced by the programming language or a specialized tool through runtime checks to ensure compliance. A common way to implement explicit contracts is through preconditions and postconditions.

Precondition (Requires clause): Characterizes the responsibility of the program that calls (uses) the method (client code)

Postcondition (Ensures clause): Characterizes the responsibility of the program that implements the method (implementation code in the method body)[2]

```java
// Interface shape that represents an explicit contract.
interface Shape {
    /**
     * Calculates the area of the shape.
     *
     * @return The area of the shape.
     * @precondition None.
     * @postcondition The returned value is non-negative.
     */
    double calculateArea();

    /**
     * Calculates the perimeter of the shape.
     *
     * @return The perimeter of the shape.
     * @precondition None.
     * @postcondition The returned value is non-negative.
     */
    double calculatePerimeter();
}


// Circle class must follow the explicit contract rules as defined in the Shape
interface.
class Circle implements Shape {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    @Override
    public double calculateArea() {
        return Math.PI * radius * radius;
    }

    @Override
    public double calculatePerimeter() {
        return 2 * Math.PI * radius;
    }
}

class Rectangle implements Shape {
    private double width;
    private double height;

    public Rectangle(double width, double height) {
        this.width = width;
```

```
        this.height = height;
    }

    @Override
    public double calculateArea() {
        return width * height;
    }

    @Override
    public double calculatePerimeter() {
        return 2 * (width + height);
    }
}

public class Main {
    public static void main(String[] args) {
        Shape circle = new Circle(5.0);
        Shape rectangle = new Rectangle(4.0, 6.0);

         // Both Circle and Rectangle classes follow the explicit contract rules
as defined in the Shape interface.
        System.out.println("Circle Area: " + circle.calculateArea());
        System.out.println("Circle Perimeter: " + circle.calculatePerimeter());

        System.out.println("Rectangle Area: " + rectangle.calculateArea());
                            System.out.println("Rectangle    Perimeter:    "    +
rectangle.calculatePerimeter());
    }
}
```

Ques 2.(5 points) What are three ways modern Java interfaces differ from a standard OO interface design that describes only abstract method signatures to be implemented? Provide a Java code example of each.

Three ways that modern Java interfaces differ from a standard OO interface design could be listed as below :-

(1) Implementation of Interfaces [3]
Prior to JDK 8, the interface could not define the implementation. We can now add default implementations for each interface method. This default implementation has a special use and does not affect the intention behind interfaces.

```
interface In1
{
	final int a = 10;
	default void display()
	{
		System.out.println("hello");
	}
}

// A class that implements the interface.
class TestClass implements In1
{
	// Driver Code
	public static void main (String[] args)
	{
		TestClass t = new TestClass();
		t.display();
	}
}
```

(2) Static Methods [4]

Modern Java interfaces can include static methods, which are associated with the interface itself and can be called without an instance of the interface.

```
//interface declaration
interface TestInterface {
    // static method definition
    static void static_print()   {
        System.out.println("TestInterface::static_print ()");
    }
    // abstract method declaration
    void nonStaticMethod(String str);
}

// Interface implementation
class TestClass implements TestInterface {
    // Override interface method
    @Override
    public void nonStaticMethod(String str)   {
        System.out.println(str);
    }
}
public class Main{
    public static void main(String[] args)   {
        TestClass classDemo = new TestClass();

        // Call static method from interface
        TestInterface.static_print();

        // Call overridden method using class object
        classDemo.nonStaticMethod("TestClass::nonStaticMethod ()");
    }
}
```

(3) Functional Interfaces [5]

Modern Java interfaces can be designated as functional interfaces, which are interfaces with a single abstract method (SAM). It can contain any number of default and static methods but it will have exactly one abstract method. Additionally, a functional interface can have declarations of object class methods. It can be declared using - *@FunctionalInterface*

```
//declare a functional interface
@FunctionalInterface        //annotation of functional interface
interface function_Interface{
    void disp_msg(String msg);    // abstract method

    int hashCode();
    String toString();
    boolean equals(Object obj);
}
//implementation of Functional Interface
class FunctionalInterfaceExample implements function_Interface{
    public void disp_msg(String msg){
        System.out.println(msg);
    }
}
class Main{
    public static void main(String[] args) {
        //create object of implementation class and call method
        FunctionalInterfaceExample finte = new
FunctionalInterfaceExample();
        finte.disp_msg("Hello, World!!!");
    }
```

Ques 3.(5 points) Describe the differences and relationship between abstraction and encapsulation. Provide a text, pseudo code, or Java code example illustrating the difference.

Ans. Abstraction and encapsulation are one of the two pillars of object-oriented programming. They can be differentiated as follows :-

Abstraction [3]

- In a single word, abstraction can be defined as a generalization of methods such that the user does not have to worry about the internal implementations of various methods.
- It is used to hide unnecessary complex details from users, resulting in applications that are simpler to maintain, reuse, adapt, and collaborate on with various developers.
- It is achieved by defining abstract classes and methods. An abstract class can't be instantiated, and abstract methods don't have implementations, also they provide a blueprint for subclasses.

Encapsulation [4]

- It provides security to a class's data and methods from any external interference. It involves wrapping data and methods within a class, creating a protective barrier that ensures the class's internal workings remain secure and organized.
- It is achieved by using access modifiers (e.g., private, protected, public) to control the visibility of attributes and methods within a class. It encourages the use of getter and setter methods to interact with an object's state.

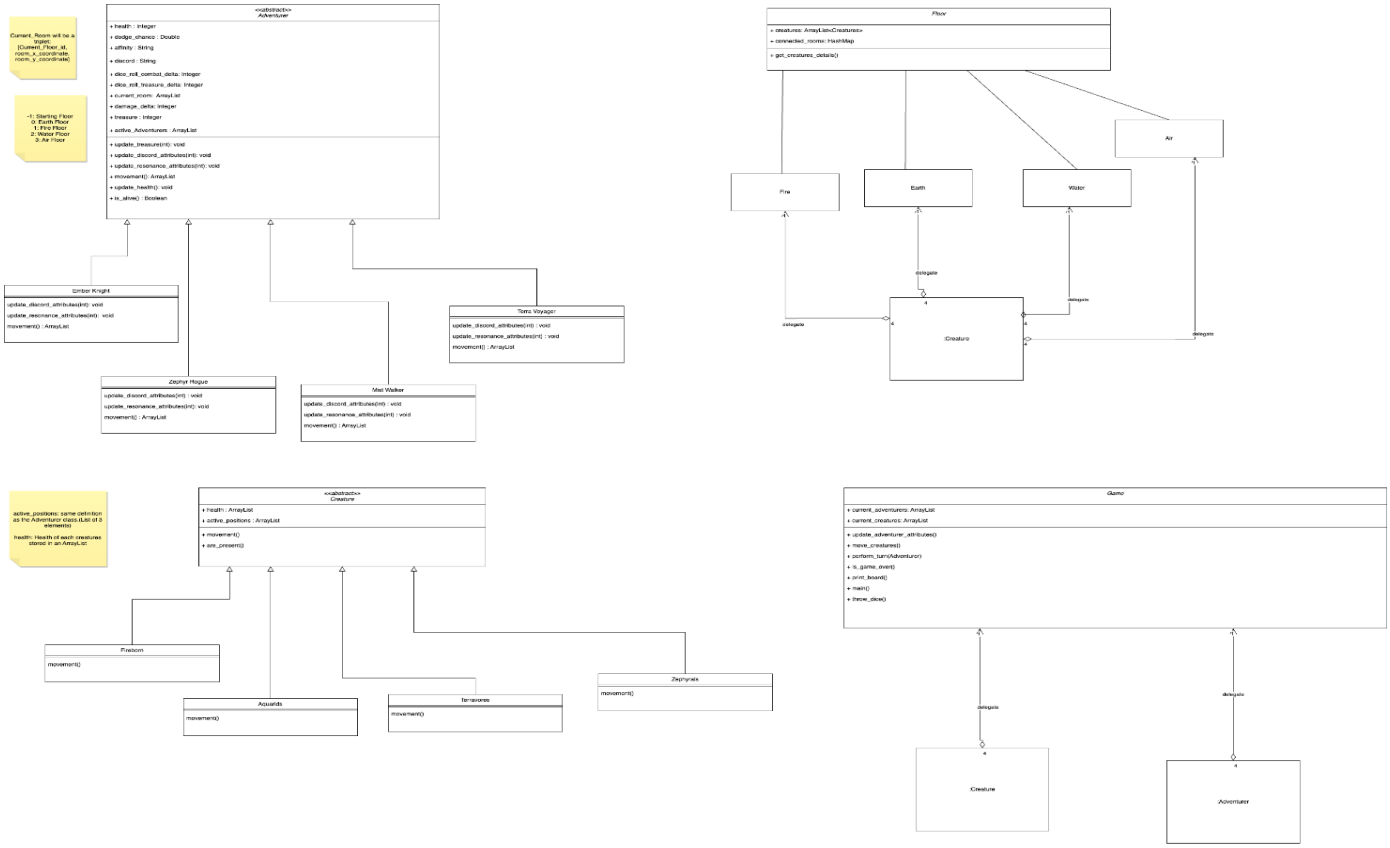
```
// Encapsulation: Car class with private attributes and getter methods
class Car {
    private String make;
    private String model;

    public Car(String make, String model) {
        this.make = make;
        this.model = model;
    }

    public String getMake() {
        return make;
    }

    public String getModel() {
        return model;
    }
}

// Abstraction: Abstract Vehicle class with an abstract method
```

```java
abstract class Vehicle {
    private String registrationNumber;

    public Vehicle(String registrationNumber) {
        this.registrationNumber = registrationNumber;
    }

    public abstract void start();

    public String getRegistrationNumber() {
        return registrationNumber;
    }
}

// Concrete subclass of Vehicle
class ElectricCar extends Vehicle {
    private String make;
    private String model;

    public ElectricCar(String registrationNumber, String make, String model) {
        super(registrationNumber);
        this.make = make;
        this.model = model;
    }

    @Override
    public void start() {
        System.out.println("Starting the electric engine of a " + make + " " +
model + " car.");
    }
}

public class Main {
    public static void main(String[] args) {
        ElectricCar myCar = new ElectricCar("ABC123", "Tesla", "Model 3");

        // Encapsulation: Accessing private attributes through getter methods
        System.out.println("Make: " + myCar.getMake()); // Make: Tesla
        System.out.println("Model: " + myCar.getModel()); // Model: Model 3

        // Abstraction: Using abstract method
        myCar.start(); // Starting the electric engine of a Tesla Model 3 car.

        // Accessing registration number (inherited from Vehicle)
        System.out.println("Registration Number: " +
myCar.getRegistrationNumber()); // Registration Number: ABC123
    }
}
```

Ques 4.(5 points) Draw a UML class diagram for the ARCANE simulation described in Project 2.2. The class diagram should contain any classes, abstract classes, or interfaces you plan to implement to make the system work. Classes should include any essential methods or attributes (not including constructors). Delegation or inheritance links should be clear. Multiplicity and accessibility tags are optional. The design of this UML diagram should be a team activity and should help with eventual code development.

UML Link :-
https://drive.google.com/file/d/1p0trkFw1ftZast9fcF618H2jutk9hW5_/view?usp=sharing

UML attached below.


Current_Room will be a triplet: (Current_Floor_id, room_x_coordinate, room_y_coordinate)
-1: Starting Floor
0: Earth Floor
1: Fire Floor
2: Water Floor
3: Air Floor
«abstract»
Adventurer
+ health : Integer
+ dodge_chance : Double
+ affinity : String
+ discord : String
+ disc_roll_combat_delta: Integer
+ disc_roll_treasure_delta: Integer
+ current_room: ArrayList
+ damage_delta: Integer
+ treasure : Integer
+ active_Adventurers : ArrayList
+ update_treasure(int): void
+ update_discord_attributes(int): void
+ update_resonance_attributes(int): void
+ movement(): ArrayList
+ update_health(): void
+ is_alive() : Boolean
Ember Knight
update_discord_attributes(int): void
update_resonance_attributes(int): void
movement() : ArrayList
Tora Voyager
update_discord_attributes(int) : void
update_resonance_attributes(int) : void
movement() : ArrayList
Zephyr Rogue
update_discord_attributes(int) : void
update_resonance_attributes(int): void
movement() : ArrayList
Mist Walker
update_discord_attributes(int) : void
update_resonance_attributes(int) : void
movement() : ArrayList
Floor
+ creatures: ArrayList<Creatures>
+ connected_rooms: HashMap
+ get_creatures_details()
Fire
Earth
Water
Air
delegate
1
*
/Creature
active_positions: same definition as the Adventurer class (List of 2 elements)
health: Health of each creatures stored in an ArrayList
«abstract»
Creature
+ health : ArrayList
+ active_positions : ArrayList
+ movement()
+ are_present()
Firelorn
movement()
Aquarids
movement()
Terrasores
movement()
Zephyrals
movement()
Game
+ current_adventurers: ArrayList
+ current_creatures: ArrayList
+ update_adventurer_attributes()
+ move_creatures()
+ perform_turn(Adventurer)
+ is_game_over()
+ print_board()
+ main()
+ throw_dice()
/Creature
Adventurer