Project 2 - part 1

Team member: Srinivas Akhil Mallela, Likhitha Katakam**,** Vishal Prabhachandar
Section: CSCI 5448-001

1. 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

Answer:

Design by contract is an approach for designing software where the schema put in place by the public methods and class/subclass designs in the OO programs are respected. Method overloading is one such example that is powerful and dangerous but could break the Design by contract principle as there is a possibility that a subclass may change the behavior of a method such that it no longer follows the contract established by a superclass. This will break abstraction and reduce the maintainability of code.

Implicit design by contract is when compliance is not enforced. Examples are Java abstract class & Interface. Explicit design by contract is when compliance can be enforced in some languages like Eiffiel where the specification for a service/routine can be explicitly written in code. Libraries are available for languages like Python, Java to enable explicit design by contract.

Code snippet for implicit contract :
The Java Interface Vehicle sets up the abstract methods but it's up to the implementing Bicycle class to follow the contract and not break abstraction.

```
interface Vehicle {

    // all are the abstract methods.
    void changeGear(int a);
    void speedUp(int a);
    void applyBrakes(int a);
}

class Bicycle implements Vehicle{

    int speed;
    int gear;

     // to change gear
    @Override
    public void changeGear(int newGear){

        gear = newGear;
    }

    // to increase speed
    @Override
    public void speedUp(int increment){

        speed = speed + increment;
    }

    // to decrease speed
    @Override
    public void applyBrakes(int decrement){

        speed = speed - decrement;
    }

    public void printStates() {
         System.out.println("speed: " + speed
              + " gear: " + gear);
    }
}
```

Code snippet for explicit contract :
Eiffel provides assertions like require and ensure to enforce design by contract programming

```
note
    description: "Simple bank accounts"

class
    ACCOUNT

feature -- Access

    balance: INTEGER
            -- Current balance

    deposit_count: INTEGER
            -- Number of deposits made since opening
        do
            ... As before ...
        end

feature -- Element change

    deposit (sum: INTEGER)
            -- Add `sum' to account.
        require
            non_negative: sum >= 0
        do
            ... As before ...
        ensure
            one_more_deposit: deposit_count = old deposit_count + 1
            updated: balance = old balance + sum
        end

feature {NONE} -- Implementation

    all_deposits: DEPOSIT_LIST
            -- List of deposits since account's opening.

invariant
    consistent_balance: (all_deposits /= Void) implies
            (balance = all_deposits . total)
    zero_if_no_deposits: (all_deposits = Void) implies
            (balance = 0)

end -- class ACCOUNT
```

---

2. What are three ways modern Java interfaces differ from a standard OO interface design that describes only function signatures and return types to be implemented? Provide a Java code example of each.

Answer:

The three ways modern Java interfaces differ from the traditional object-oriented interface design are as follows:

1. Before the release of Java 8, interfaces could only have constant variables and abstract methods. However, with the release of Java 8 interfaces could have ***default*** methods.

A default method can be defined in the interface body with the *default* keyword. It is required that default methods have a body and have their access modifiers implicitly *public.* The benefit of adding the default method is that it allows developers to add new functionality to interfaces without breaking the old version of the code, as it doesn't require classes to implement this.

Code snippet for *default* method in the interface:

```
interface MyInterface  {
   void calculateArea();
   default void print() {
     System.out.println("This is default method");
   }
 }

class DefaultMethod implements MyInterface {
   public void calculateArea() {
      System.out.println("Calculate area in this method");
    }
   public static void main(String args []) {
      DefaultMethod obj = new DefaultMethod();
      obj.print();
      obj.calculateArea();
    }
}
```

As we can see in the above code, the MyInterface interface is implemented by the DefaultMethod class, however, it's not required that the class implement the default method and can be used directly. The above code will output "this is default method" and "Calculate area in this method" respectively.

2. Java 8 also introduced another feature to interfaces. It allowed defining ***static*** methods in interfaces, this allowed for calling those methods without the object of the implementing class. The static method in the interface had some definitive features, they were supposed to have a body and the access modifiers of the method had to be implicitly public. These methods are called using the interface name and as they are static, they are not allowed to be overridden.

Code snippet for *static* method in the interface:

```
interface MyInterface  {
   void calculateArea();
   static void print() {
      System.out.println("This is static method");
    }
 }

class StaticMethod implements MyInterface {
    public void calculateArea() {
      System.out.println("Calculate area in this method");
    }
   public static void main(String args []) {
      StaticMethod obj = new StaticMethod();
      obj.calculateArea();
      MyInterface.print();
    }
}
```

In the above example, we have a static method defined in the interface MyInterface and it can be directly called using the interface name as seen in the main function of the code. The output for the following code would be, "Calculate area in this method" followed by "This is a static method".

3. Java 9 introduced ***private*** methods and ***private static*** methods in interfaces. They were introduced to improve the code re-usability, and generally serve as a utility method that helps other defined methods inside the interface. The private methods were required to

have a body, they could be accessed from within the interface only and as they are private they cannot be overridden in the implementing class.

Code snippet for *private* and *private static* methods in the interface:

```
interface MyInterface  {
   default void print() {
      System.out.println("This is default method");
      // calling private methods
      message();
      detail();
    }
   private void message() {
     System.out.println("This is private method");
    }
   private static void detail() {
     System.out.println("This is private static method");
    }
 }

class PrivateMethods implements MyInterface {
   public static void main(String args []) {
      PrivateMethods obj = new PrivateMethods();
      obj.print();
    }
 }
```

The above example shows the implementation of a private method - message and private static method - detail. The code outputs “This is default method”, “This is private method”, “This is private static method” respectively.

---

3. Describe the differences and relationship between abstraction and encapsulation. Provide a java code example that illustrates the difference.

Answer:

Abstraction is a property by virtue of which only the essential details are displayed to the user and hiding the complex details or implementation using abstract classes and interfaces. Whereas Encapsulation is the process of combining data and functions into a single unit called class. The data is not accessed directly, it is accessed through the functions present inside the class. Both are complementary concepts. Abstraction focuses on the observable behavior of an object and encapsulation is a tool for doing abstraction. Abstraction focuses mainly on what should be done, whereas encapsulation focuses primarily on how something should be done. Abstraction is used mainly during the design level of an application and encapsulation is used at the implementation level of an application.

Encapsulation code snippet: Using helper methods to access private variables

```
public class Encapsulate {

    // private variable
    private String Name;

   //getter
    public String getName() {
        return Name;
    }

    //setter
    public void setName(String newName) {
        Name = newName;
    }

}
```

Abstraction code snippet: Hiding information of Animal class. Objects of animal class cannot be created.

```
// Abstract class
abstract class Animal {
```

```
    // Abstract method does not have a body
    public abstract void animalSound();
    // Normal method
    public void sleep() {
        System.out.println("Zzz");
    }
}

// Subclass
class Pig extends Animal {
    public void animalSound() {
        // The body of animalSound() is provided here
        System.out.println("The pig says: wee wee");
    }
}

class Main {
    public static void main(String[] args) {
        Pig myPig = new Pig(); // Create a Pig object
        myPig.animalSound();
        myPig.sleep();
    }
}
```

---

4. UML diagram for the FNMS simulation.

In case the below UML is not clear, kindly look at the below-attached links for a pdf or png file.

- PDF
- PNG
- Github

References

- Class Slides
- Interfaces in Java
- Eiffel - design by contract
- Difference between abstraction and encapsulation
- Encapsulation vs Abstraction
- Interface features in modern Java

FNMS

Team member: Srinivas Akhil Mallela, Likhitha Katakam, Vishal Prabhachandar

Project 2 Q4 - UML