Sidhant Puntambekar
Brian Noble
Isaac Pyle
1/26/2022

CSCI 4448: Project 2 Part 1

1. **What does "design by contract" mean in an OO paradigm? What is the difference between implicit and explicit contracts? Provide a text, pseudo code, or code example of each contract type**
    a. Design by contract in an OO paradigm refers to the process of keeping the integrity of the definitions for public methods and classes between superclasses and subclasses in an object oriented program. The design by contract principle particularly refers to method overloading where a subclass has the potential to change public method behavior and thereby violate the behavior established in the superclass that the subclass inherited from. This would be bad as it would destroy previously written robust and abstracted code. Design by contract also refers to the specifications and requirements on inputs and outputs to the program (such that unexpected behavior does not occur) when the program is running. (Lecture 5: OO Fundamentals, https://john.cs.olemiss.edu/~hcc/softArch/notes/iContract/OW_Berlin99_web.pdf)

    b. The main difference between implicit and explicit contracts is that implicit contract refers to indirectly protecting the schema defined by public methods and class/subclass designs in an object oriented program. In other words, this means in practice that the programmer must respect the structure and inheritance hierarchy of the object oriented program since changes in the design by contract are not actively monitored in code. This contrasts with explicit design by contract which directly defines and enforces the type specification for the public method actively in explicit code instructions. (Lecture 5: OO Fundamentals, https://john.cs.olemiss.edu/~hcc/softArch/notes/iContract/OW_Berlin99_web.pdf)

    c. Examples of implicit and explicit pseudo code are presented below:

        i. **Implicit Contract (https://john.cs.olemiss.edu/~hcc/softArch/notes/iContract/OW_Berlin99_web.pdf)**

```
1. class Animal {
        private age;

        /** The returned age should be greater than 0.  If age was
        * initialized to a negative value
        * then the age variable would not make sense because
        * age of an animal could never be negative
        * @post return > 0
```

```
        */
        public int getAge() {
            return age;
        }

        /** If we try to set the age of an animal, then newAge
        *should be greater than 0. Ages do not make sense if
        *they are negative so the behavior of this method should *
        return a greater than 0 age
        * @pre age > 0
        */
        public void setAge(int newAge) {
            age = newAge;
        }
    }
```

ii. Explicit Contract

1.
```
class Animal {
        private age;

        public int getAge() {
            if (age > 0) {
                return age;
            }
            else {
                throw new IllegalArgumentException("Age is
        not a positive number");
            }
        }

        public void setAge(int newAge) {
            if (newAge > 0) {
                age = newAge;
            }
            else {
                throw new
        IllegalArgumentException("newAge is not a positive
        number");
            }
        }
    }
```

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.**
    a. Since Java version 8, the language has provided support for default methods, static methods, and lambda expressions which differentiate post Java version 8 interfaces from the standard OO interface design.
        i. The **default** keyword for interface methods tells the implementing class that it is not a requirement to provide an implementation for that method (Lecture 6: Intro to Java, https://www.journaldev.com/2752/java-8-interface-changes-static-method-default-method). An example of the default keyword for interfaces can be seen below:

```
1. interface myInterface {
        public void add(int a, int b);

        default void show() {
            System.out.println("Default method executing");
        }
    }
    class myClass implements myInterface {
        public void add(int a, int b) {
            int sum = a + b;
            System.out.println(sum)
        }

        public static void main(String[] args) {
            myClass class1 = new myClass();
            class1.add(7, 11);

            class1.show();
        }
    }
```

        (https://www.geeksforgeeks.org/default-methods-java/)

        ii. The **static** keyword for interface methods allows the implementing class to call the interface method directly from the interface itself. It also allows the interface to use the static interface method for other default method implementations (Lecture 6: Intro to Java, https://www.geeksforgeeks.org/default-methods-java/). An example of the static keyword for interfaces can be seen below:

```
1. interface myInterface {
        static int doubleInt(int x) {
            return x*2;
        }
```

```
}

class Main{
    public static void main(String[] args) {
        System.out.println(myInterface.doubleInt(5));
    }
} (https://www.geeksforgeeks.org/default-methods-java/)
```

iii. **Lambda expressions** allow functions to be defined in-line without having to write separate function declarations and contents somewhere else in the program. The lambda expression remains local to the method it was defined in (Lecture 6: Intro to Java, https://www.tutorialspoint.com/java8/java8_lambda_expressions.html). An example of a lambda expression can be seen below:

1.
```
interface funcInterface {
    void fun(int x);
}

class Main{
    public static void main(String[] args){
        funcInterface fobj = (int x)->System.out.println(x*x);
        fobj.fun(25);
    }
} (Lecture 6: Intro to Java)
```

3. **Describe the differences and relationship between abstraction and encapsulation. Provide a Java code example that illustrates the difference.**
    a. Abstraction refers to the process of showing only essential attributes and hiding unnecessary information away from the user. Encapsulation is defined as the containment of data and methods into a single functional entity in order to prevent unauthorized access of code from external entities. (https://eng.libretexts.org/Courses/Delta_College/C_-_Data_Structures/06%3A_Abstraction_Encapsulation/1.01%3A_Difference_between_Abstraction_and_Encapsulation)

        i. One of the main differences between abstraction and encapsulation is that abstraction occurs at the design/interface phase while encapsulation occurs more so at the implementation level.

        ii. Another difference between abstraction and encapsulation is that abstraction is implemented in Java through abstract classes and interfaces (classes extend these entities) while encapsulation is implemented using access modifiers such as public, private, and protected.

iii. Abstraction is the process of gaining the information, while encapsulation is the process of containing the information.

iv. Objects that perform abstraction are encapsulated, but objects that are encapsulated are not necessarily abstract.

v. Abstraction is a more general design concept, encapsulation supports abstraction.

(referenced here: https://www.geeksforgeeks.org/difference-between-abstraction-and-encapsulation-in-java-with-examples/)

b. Abstraction vs. Encapsulation example

i. **Abstraction**:

```
abstract class Fruit {
        String color;
        public abstract String getColor();
        public abstract void setColor(String newColor)
        public abstract String ripen();
}

class abstract_banana extends Fruit {

        public String getColor() {
            return color;
          }


        public void setColor(String newColor) {
            color = newColor;
          }

        // This method abstracts the process of ripening a fruit by turning
        the color to yellow and returning it (thereby changing the state of
        the color variable and the banana).

        public String ripen() {
            color = yellow;
            return color;
        }
}
```

ii. **Encapsulation**:

```
class encapsulated_apple {
        // Here color is private such that it can't be changed by any outside
entity. It is hidden from the view of other parts of the code and can only be
changed by going through the mutator method associated with the
"encapsulated_apple" class.

        private String color;

        public String getColor() {
                return color;
        }

        public void setColor(String newColor) {
                color = newColor;
        }
}
```

4. **https://lucid.app/lucidchart/689a62ee-337a-4d88-ad4f-758c9e37d35e/edit?invitationId=inv_bddd9ab2-aa6f-45bd-861a-bc2f8d7714e8**