Talon Knowlton

CSCI 4448

June 8, 2022

OOAD Project 1 Definitions

Abstraction: This is the idea of only showing essential details and hiding everything else. It's like having a microwave and only knowing how to operate it. You may know how to use the microwave through interfacing with the buttons, but it doesn't mean you need know how individual parts/code of it work to use it. You simply have this abstract view that when you put some buttons it will add some time to the clock and start cooking your food.

In programming, one piece of code will interact with other pieces of code through an interface without knowledge of the algorithms and implementations used by the other code. Information between two programs will be exchanged and neither will know any of the inner workings of each other.

For example, in part two of this assignment in my Analyzer class I have a method called analyze that calls all of the methods within the class to perform the various calculations required in the assignment (sum(), median(), etc.). The analyze() method has no idea how sum(), median(), or how any of the other methods work. It has no need to know the algorithms used within each method. All it knows is that if you call that function with the necessary parameters, you will get the correct result.

```
class Analyzer{

    public void analyze(ArrayList<Integer> mylist)
    {
        mylist.sort(Comparator.naturalOrder());//sort Array

        //call all functions
        double list_sum=(sum(mylist));
        double median=(median(mylist));
        double mean=(mean(mylist));
        double minimum=(minimum(mylist));
        double maximum=(maximum(mylist));
        double standard_deviation=(standard_deviation(mylist));

        //output the resulting calculations
        System.out.println("The list sum is " + list_sum);
        System.out.println("The list median is " + median);
        System.out.print(s: "The list mean is ");
        System.out.format(format: "%.2f",  mean);
        System.out.println(x: " ");
        System.out.println("The list standard deviation is " + standard_deviation);
        System.out.println("The list minimum is " + minimum);
        System.out.println("The list maximum is " + maximum);

    }
```

Encapsulation: This is the idea of hiding data within a class and preventing anything outside that class from directly interacting with it. It "encapsulates" the data of a specific object. It differs from abstraction in that encapsulation is *data hiding* whereas abstraction is *implemenation hiding*. In order

for members of class 1 to interact with private attributes of class 2, class 1 must use the public getting and setting methods within class 2. This allows programers to control the access of information.

This can be seen in the simple example below from GeeskforGeeks. Within the “Name” class you have a private integer called “age” that can only be accessed from within the Name class. But since the “Name” class has a public getter method called “setAge()”, the “GFG” class is able to access the private integer and assign it a new value. The “GFG” class is also able to again access the private integer “Age” by using a getter method called “getAge().” Without the getter or setter methods within the “Name” class there is no way for other classes to access the “Age” variable.

```
class Name {

    private int age; // Private is using to hide the data

    public int getAge() { return age; } // getter

    public void setAge(int age)
    {
        this.age = age;
    } // setter
}

class GFG {
    Run | Debug
    public static void main(String[] args)
    {

        Name n1 = new Name();

        n1.setAge(age: 19);

        System.out.println("The age of the person is: "
                           + n1.getAge());
    }
}
```

Example Code From Geeks for Geeks

Polymorphism: Polymorphism means having many forms. In programming this means that methods are able to take on many forms. There is dynamic and static polymorphism.

- Dynamic polymorphism occurs during run-time. It describes when a a method is in both a subclass and a superclass. The methods share the same name, but have different implementations. The implementation of the subclass that the object is an instance of overrides that of the superclass.

  In the example below you have two classes, a base class and a derived class, that both have a method called “void fun().” If you create an instance of the base class and call “fun()” it will call the “fun()” function within the base class (printing out “Base class”). But if you instead create an instance of the derived class and call “fun()” it will override the base class “void fun()” and instead call “void fun ()” within the Derived class (printing out “Derived class”).

```java
//Parent class
class Base {

    // Method of Base class
    void fun()
    {
        System.out.println(x: "Base class");
    }
}

// Subclass
class Derived extends Base {

    void fun()
    {
        System.out.println(x: "Derived class");
    }
}
```

- Static polymorphism (aka method overloading) occurs during compile-time rather than during runtime. This refers to when multiple methods with the same name, but different argurments/number of arguments are defined within the same class.

  For example lets say that we have a class called “multiply_class” that has three methods all called “Multiply”. The first method takes in two integers, the second takes in two doubles, and the third takes in three double. Depending on how you make a call to the “Multiply” function in terms of arguments given will determine exactly which method is called within the class.

```java
class multiply_class {

    // Method with 2 integer parameters
    static int Multiply(int a, int b)
    {
        return a * b;
    }

    // Method 2 with same name but with 2 double parameters
    static double Multiply(double a, double b)
    {
        return a * b;
    }

    // Method 3 with same name but with 3 double parameters
    static double Multiply(double a, double b, double c)
    {
        return a * b*  c;
    }
}
```

- Multiply(3, 4) will call the first method
- Multiply(3.0, 4.0) will call the second method
- Multiply(3.0, 4.0, 5.0) will call the third method

Coupling:

Coupling is the strength of the connection between *two classes*. It's a measure of how much of a change in class A creates changes in class B. Ideally you want weak coupling (code is not dependent on other code) so that a single change will not ripple through a system. Within classes this is the dependency of one object on another ojbect.

In the example below (from GeeksforGeeks) we can see tight and loose coupling. In the tight coupling on the left there is an interdependency between the classes where a change in the Box class will cause a change in Volume class. The variable "volume" is public and open to access by any other class (there is a strong connection between classes).

On the right we have loose coupling where two clases are instead less connect. Instead of relying on the variable "volume" to be public and open to anything (in tight coupling), we can instead use the "getVolume()" function to act as a getter that returns the now private variable "volume." Other classes are able to interact with the Box class less and have to do it through its designated getter function.

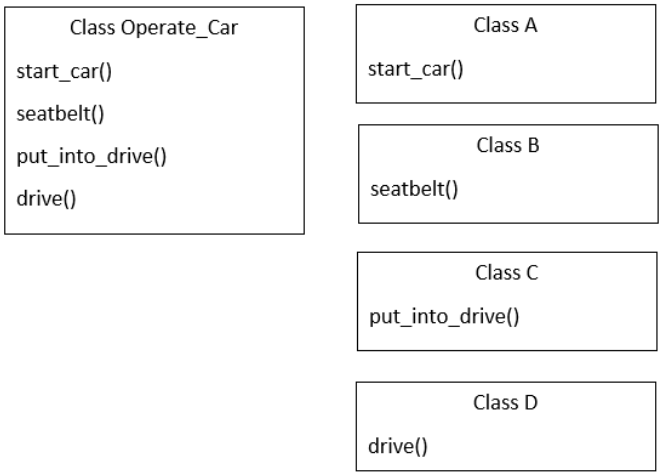
```java
// tight coupling concept
class Volume
{
    Run | Debug
    public static void main(String args[])
    {
        Box b = new Box(length: 5,width: 5,height: 5);
        System.out.println(b.volume);
    }
}
class Box
{
    public int volume;
    Box(int length, int width, int height)
    {
        this.volume = length * width * height;
    }
}
```

```java
// loose coupling concept
class Volume
{
    Run | Debug
    public static void main(String args[])
    {
        Box b = new Box(5,5,5);
        System.out.println(b.getVolume());
    }
}
final class Box
{
    private int volume;
    Box(int length, int width, int height)
    {
        this.volume = length * width * height;
    }
    public int getVolume()
    {
        return volume;

    }
}
```

Cohesion:

Cohesion is how closely the operations within a *single class* are related. It's about how much a single part of a program is able to do.



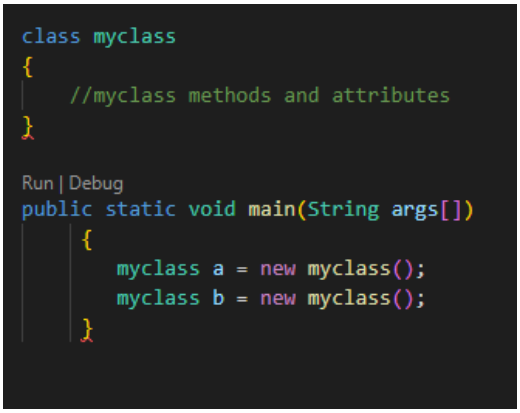
Low Cohesion on the left, high cohesion on the right

In the example above we have a class on the left called "Operate_Car" which has low cohesion because one single class has to do multiple things. On the right side of the image we have an example of high cohesion where there are multiple classes that focus on their own specific functionality.

What We Want:

Weak cohesion implies that there is strong coupling. Strong cohesion implies that there is loose coupling. Ideally in programming we want there to be strong cohesion (where various parts of the code focus on one specific function/specialization) and weak coupling (code is not dependent on other code). If we have strong coupling, a single change in one method or data structure will cause unwanted ripple effects through the system.

Identity: They are unique names given to an instance of an object. While two different indentites may be an instance of the same class, they still are their own unique instance/identity of a class. In the example below we have a class called "myclass." In the main we declare two new identities of "myclass" called "a" and "b." These two identities are both objects of the same class, but they are their own unique declaration of "myclass."


class myclass
{
//myclass methods and attributes
}
Run | Debug
public static void main(String args[])
{
myclass a = new myclass();
myclass b = new myclass();
}

Works Cited

"Intro to Object Oriented Programming - Crash Course." *YouTube*, YouTube, 30 Sept. 2020, https://www.youtube.com/watch?v=SiBw7os-_zI. Accessed 7 June 2022.

"Object Oriented Programming (OOPS) Concept in Java." *GeeksforGeeks*, 13 May 2022, https://www.geeksforgeeks.org/object-oriented-programming-oops-concept-in-java/.