LAB 9:

Designing Classes - I

Introduction:

In this lab, we look at some of the factors that influence the design of a class. What makes a class design either good or bad? Writing good classes can take more effort in the short term than writing bad classes, but in the long term that extra effort will often be justified. To help us write good classes, there are some principles that we can follow. We introduce the view that class design should be responsibility-driven, and that classes should encapsulate their data.

The problems typically surface when a maintenance programmer wants to make some changes to an existing application. If, for example, a programmer attempts to fix a bug, or wants to add new functionality to an existing program, a task that might be easy and obvious with well-designed classes, may well be very hard and involve a great deal of work if the classes are badly designed.

Two terms are central when talking about the quality of a class design:

- *Coupling*
- *Cohesion*

COUPLING

The term *coupling* refers to the interconnectedness of classes. The degree of coupling indicates how tightly these classes are connected. We strive for a low degree of coupling, or *loose coupling*.
The degree of coupling determines how hard it is to make changes in an application.

TIGHT COUPLING

In a tightly-coupled class structure, a change in one class can make it necessary to change several other classes as well. This is what we try to avoid because the effect of making one small change can quickly ripple through a complete application. In addition, finding all the places where changes are necessary and actually making the changes can be difficult and time consuming.

***IPods** are a good example of tight coupling: once the battery dies you might as well buy a new iPod because the battery is soldered fixed and won't come loose, thus making replacing very expensive. A loosely coupled player would allow effortlessly changing the battery.*

LOOSE COUPLING

In a loosely coupled system, on the other hand, we can often change one class without making any changes to other classes, and the application will still work.

COHESION

The term *cohesion* relates to the number and diversity of tasks for which a single unit of an application is responsible. Cohesion is relevant for units of a single class and an individual method.

A class with high cohesion would be one where all the methods and class level variables are used together to accomplish a specific task. On the other end, a class with low cohesion is one where functions are randomly inserted into a class and used to accomplish a variety of different tasks.
IMPORTANT

Generally tight coupling gives low cohesion and loose coupling gives high cohesion.

CODE # 1

```
class TightCoupling
{
  public void ShowWelcomeMsg(string type)
  {
    switch (type)
    {
      case “GM”:
        Console.WriteLine(“Good Morning”);
        break;
      case “GE”:
        Console.WriteLine(“Good Evening”);
        break;
      case “GN”:
        Console.WriteLine(“Good Night”);
        break;
    }
  }
}
class TightCoupling2
{
  public TightCoupling2()
  {
    TightCoupling example = new TightCoupling();
    example.ShowWelcomeMsg(“GE”);
  }
}
```

In the above example, the ShowWelcomeMsg function cannot be called without knowing the inner workings of that function making it nearly useless for other systems. Secondly, if another developer in the future decides to change the switch statement in the ShowWelcomeMsg function, it could inadvertently effect other classes that call that function.

CODE # 2

The code below is of an EmailMessage class that has high cohesion. All the methods and class level variables are very closely related and work together to accomplish a single task.

```
class EmailMessage
{
  private string sendTo;
  private string subject;
  private string message;
  public EmailMessage(string to, string subject, string message)
  {
    this.sendTo = to;
    this.subject = subject;
    this.message = message;
  }
  public void SendMessage()
  {
    // send message using sendTo, subject and message
  }
```

```
}
```

Now here is an example of the same class but this time as a low cohesive class. This class was originally designed to send an email message but sometime in the future the user needed to be logged in to send an email so the Login method was added to the EmailMessage class.

```
class EmailMessage
{
  private string sendTo;
  private string subject;
  private string message;
  private string username;
  public EmailMessage(string to, string subject, string message)
  {
    this.sendTo = to;
    this.subject = subject;
    this.message = message;
  }
  public void SendMessage()
  {
    // send message using sendTo, subject and message
  }
  public void Login(string username, string password)
  {
    this.username = username;
    // code to login
  }
}
```

The Login method and username class variable really have nothing to do with the EmailMessage class and its main purpose. This class now has low cohesion and is probably not a good example to follow.

Lab Tasks:

Task 1:

Discuss at least one scenario that clearly explains the concept of cohesion and coupling with reference to java. Explain the example with code.

- You can take of any real time scenario as a case study.
- You can also use your already developed projects (make some modifications through cohesion and coupling).

Task 2:

Is the code below tightly coupled if yes provide a reason and implement a loosely coupled working version of it?

```
// Java program to illustrate
// tight coupling concept
class Volume
{
    public static void main(String args[])
    {
        Box b = new Box(5,5,5);
        System.out.println(b.volume);
    }
```

```
}
class Box
{
    public int volume;
    Box(int length, int width, int height)
    {
        this.volume = length * width * height;
    }
}
```

Task 3:

Is the code below highly cohesive is yes kindly provide a reason?

```
class Name {
        String name;
        public String getName(String name)
        {
            this.name = name;
            return name;
        }
}

class Age {
        int age;
        public int getAge(int age)
        {
            this.age = age;
            return age;
        }
}

class Number {
        int mobileno;
        public int getNumber(int mobileno)
        {
            this.mobileno = mobileno;
            return mobileno;
        }
}

class Display {
        public static void main(String[] args)
        {
            Name n = new Name();
            System.out.println(n.getName("Geeksforgeeks"));
            Age a = new Age();
            System.out.println(a.getAge(10));
            Number no = new Number();
            System.out.println(no.getNumber(1234567891));
        }
}
```

```
class Name {
        String name;
        public String getName(String name)
```

```
                {
                    this.name = name;
                    return name;
                }
}

class Age {
                int age;
                public int getAge(int age)
                {
                    this.age = age;
                    return age;
                }
}

class Number {
                int mobileno;
                public int getNumber(int mobileno)
                {
                    this.mobileno = mobileno;
                    return mobileno;
                }
}

class Display {
                public static void main(String[] args)
                {
                    Name n = new Name();
                    System.out.println(n.getName("Geeksforgeeks"));
                    Age a = new Age();
                    System.out.println(a.getAge(10));
                    Number no = new Number();
                    System.out.println(no.getNumber(1234567891));
                }
}
```

Conclusion:

After today's labs, these questions regarding classes and their design can can now be answered in terms of cohesion and coupling. A method is too long if it does more than one logical task. A class is too complex if it represents more than one logical entity.