SOLID PRINCIPLES ASSIGNMENT

Solid principles are an approach applied in the object oriented programming to structure the software design.

These principles are used to make the code easier for the people working in the same environment to understand it and make necessary modification or extend the code while debug or maintenance so it is a game changer.

The Solid word can be split into:

1. Single Responsibility Principle (SRP): SRP states that a java class must have only one functionality. Because having multiple functionality can make the code complex and difficult for the next operator/programmer to understand.

   Example:

```
public class Student {


        private String name;

        private int age;

        private int roll_no;

        private String address;

        private int phone_no;

}
```

   Here you can observe that only the details pertaining to the student are stored in the class Student and nothing additional or out of the scope is included. This perfectly explains the concept of SRP.

2. Open-Closed Principle (OCP): A class contains many methods, variables and other data in them. They shouldn't be changed/altered which might cause damage to the original code. So, this principle states that the module should be open to extend but it's closed to modification.

   Example:

```
public class Computer{
        private String ram;
        private String rom;
        private String hdd;
        private String processor;
        private String motherboard;
        private String keyboard;
        private String display;
}

public class Features extends Computer{
        private String graphic_card;
        private String speaker;
}
```

   In this example it can be observed that there are two classes Computer and Features where Computer is parent class and Features is child class. Here, our goal is to build a computer for which all the necessary variables are added in the Computer class. But suppose a person wants to make modifications it has to be done by using extends to the child class so that the existing Computer doesn't suffer from failure.

3. Liskov Substitution Principle (LSP): This principle states that we should be able to replace a superclass with a derived class without interrupting the program or causing any erratic behaviour. The subclass should be able to hold the property.

   Example:

   ```
   public class Cricketer{
      public void bat(){}
   }
   public class Batsman extends Cricketer{}
   ```

   In this case the Cricketer contains bat function and the Batsman extends Cricketer class so, Batsman can bat.

   ```
   public class Bowler extends Cricketer{}
   ```

   But here the bowler extends Cricketer so he must be able to bat which is incorrect. We are not following LSP in this case.

   ```
   public class Cricketer{}
   public class Batsman extends Cricketer{
      public void bat(){}
   }
   public class Bowler extends Cricketer{
      public void bowl(){}
   }
   ```

   This example makes sense in every direction because the Batsman can bat, the Bowler can bowl.

4. Interface Segregation Principle (ISP): Multiple methods in the same interface often lead to unnecessary implementation of

the methods. So, segregating them into different interfaces leads to the use of only required ones.

Example:

```
public interface Football {
	void pass();
	void shoot();
}

public class Player implements Football{

	@Override
	public void pass() {
	}

	@Override
	public void shoot() {
	}

	}
```

Here we have created an instance with name Football. This has been implemented by the class Player. Because there are unimplemented methods in the Football interface they are supposed to be implemented in the Player class. In a few cases this might not be desirable where the player both passes and shoots the ball. So, to counter this we can create two different interfaces for pass and shoot and need to be implemented only when required. Note: Player to be created in a separate class file [Player.java] i.e the highlighted text and the interfaces also in separate files.

```
public interface Pass {
	void pass();
}
```

```
public interface Shoot {
        void shoot();
}

public class Player implements Pass{
        @Override
        public void pass() {
        }
        }
```

5. Dependency Inversion Principle (DIP): This principle states that the use of abstraction can reduce the risk of the code breaking down due to the dependence of the higher modules on the lower modules. The details should depend on abstraction and not vice versa.

   Example:

```
public class Phone {

   private final Display display;
   private final Camera camera;

   public Phone () {
      display = new Display();
      camera = new Camera ();
   }
}
```

   In this example we are creating a classes Phone, Display and Camera so every Phone has a Display and Camera. We can use them in the program. But the concern with it is that the three classes (Display, Camera and Phone) are tightly coupled because of the new keyword used in declaration. To overcome

the problem we can replace the Display class with Display interface and also use the keyword ‘this’.

```
public interface Display { }

public class Phone {

   private final Display display;
   private final Camera camera;

   public Phone (Display display, Camera camera) {
      this.display = display;
      this.camera = camera;
   }
}
```

So, the classes are decoupled and Display can communicate through abstraction. This also allows us to switch the Display with a different implementation of the interface. The same can be followed for the Camera class.