Solid Principles

SOLID is an acronym for the first five object-oriented design (OOD) principles by Robert C. Martin

1. ## Single Responsibility principle

    - There should never be more than one reason for a class to change.
      A class should only provide focused, specific, single functionality.

```
// Assume there is a user controller which is response for creating/updating/deleting a user

public class UserController {
  public boolean createUser(User user) {
    // 1. Validate user info like if user email, contact number is correct or not.
    // 2. If USER is valid then Save to DB (persist this info) else return false.
  }
}

// In above example UserController has two responsibilities - validation and persisiting user info.
// This UserController class can change whenever we add a new field in user (example - address). Because
we'll need to validate this new field as well.
// Therefore, it is not following single responsibility prince rule.

// This version of UserController is following single responsibility principle
public class UserController {
  private UserPersistenceService persistenceService;

  public UserController() {
   UserPersistenceService persistenceService = new UserPersistenceService();
  }

  public boolean createUser(User user) {
   UserValidator validator = new UserValidator();
   if(validator.isValid(user)) {
       persistenceService.saveUser(user);
   }else{
      return false;
   }
  }
}

public class UserValidator {
  public boolean isValid(User user) {
    // Validate User
  }
}

public class UserPersistenceService {
  public boolean saveUser(User user) {
    // Save User to DB
  }
}
```

2. **Open Closed Principle**

    - Sofware entities (Classes, Modules, methods) should be open for extension (extend existence behaviour) but closed for modification (existing code remains unchanged)

```
// Assume we are writing code for a telephone company that provides two types for services - phone and
ISP
public class PhoneSubscriber {
    private Long id;
    private Long phoneNumber;
    private int baseRate;
    private String address;

    public double calculatePhoneBill() {
        // Calculate phone usage and using that calculate the bill
    }
}

public class ISPSubscriber {
  private Long id;
  private Long phoneNumber;
  private int baseRate;
  private String address;
  private Long freeUsage;

  public double calculateInternetBill() {
    // Calculate internet usage and using (actual usage - free usage), calculate the bill
  }
}
// We can see that there is some common code which can be reused by both subscribers

// Base Class - closed for modification
public class Subscriber {
    protected Long id;
    protected Long phoneNumber;
    protected int baseRate;
    protected String address;

    // calculateBill is open for extension
    public abstract double calculateBill();
}

public class PhoneSubscriber extends Subscriber {
    @Override
    public abstract double calculateBill() {
      // Calculate phone usage and using that calculate the bill
    }
}

public class ISPSubscriber extends  Subscriber {
    private Long freeUsage;
    @Override
    public abstract double calculateBill() {
      // Calculate internet usage and using (actual usage - free usage), calculate the bill
    }
}

// Now above example follows open closed principle.
```

3. Liskov Substitution principle

- This states that "We should be able to substitute base class object with child class objects without altering the behaviour/characteristics of the program".
  Note: The behaviour of the program should also remain same. (This is not language specific)
- This is violated when child class completely modifies the contract/behaviour of base class method by overriding it.

```
// Base/Parent Class
public class Rectangle {

    private int width;

    private int height;

    public Rectangle(int width, int height) {
        this.width = width;
        this.height = height;
    }
    // getter and setters
    public int computeArea() {
        return width * height;
    }
}

// Dervied/Child Class
public class Square extends Rectangle {

  public Square(int side) {
    super(side, side);
  }

  @Override
  public void setWidth(int width) {
    setSide(width);
  }

  @Override
  public void setHeight(int height) {
    setSide(height);
  }

  public void setSide(int side) {
    super.setWidth(side);
    super.setHeight(side);
  }

}

public class Main {
  public static void main(String[] args) {

    Rectangle rectangle = new Rectangle(10, 20);
    System.out.println(rectangle.computeArea());

    Square square = new Square(10);
    System.out.println(square.computeArea());

    useRectangle(rectangle);
    // Acc to liskov principle, we can use child class object for base class object.
    // Therefore, we should be able to pass square but that is not the case.
    // So, it is violating Liskov substitution rule
    useRectangle(square);

  }

  private static void useRectangle(Rectangle rectangle) {
    rectangle.setHeight(20);
    rectangle.setWidth(30);
    assert rectangle.getHeight() == 20 : "Height Not equal to 20";
    assert rectangle.getWidth() == 30 : "Width Not equal to 30";
  }
}

// Let's modify the above example to follow Liskov Substitution rule.
public interface Shape {
  public int computeArea();
}

public class Rectangle implements Shape {
  private int width;
  private int height;

  public Rectangle(int width, int height) {
    this.width = width;
    this.height = height;
  }
  // getter and setters
  @Override
  public int computeArea() {
    return width * height;
  }
}

public class Square implements Shape {
    private int side;

    public Square(int side) {
        this.side = side;
    }

    public int getSide() {
        return this.side;
    }

    public int setSide(int side) {
        this.side = side;
    }
}

// now that square and rectangle does not have a direct relationship, the above example follows Liskov Substitution principle.
```

4. Interface Segregation Principle

- Clients should not be forces to depend upon interfaces that they do not use.
- **Interface Pollution** : We should not have large interfaces that have unrelated methods.
- Signs of Interface Pollution, because the class might not need these actual methods, you'll see these common patterns which violates interface segregation principle:

    1. Classes have empty method implementation
    2. Methods implementations throw unsupported operation exception
    3. Mehtods implementations returns null or dummy value.

- Write highly cohesive interfaces.

```
public abstract class Entity {
    private string id;
    // getter and setters
}

public interface PersistenceService<T extends Entity> {
    public void save(T entity);
    public void findByName(string name);
}

public interface UserPersistenceService implements PersistenceService<User>{
    @Override
    public void save(User entity){
        // save user
    }

    @Override
    public void findByName(string name){
        // get user by name
    }
}

public interface OrderPersistenceService implements PersistenceService<Order>{
    @Override
    public void save(Order entity){
      // save Order
    }
    // It does not make sense that any order will have a name. Therefore, we'll need to throw unsupported operation exception
    // Now this becomes classic case of violation of interface segregation principle
    @Override
    public void findByName(string name){
      throw UnsupportedOperationException("find by name method is not supported");
    }
}

// To fix this simply remove findByName method from PersistenceService interface as not all class will have find by name method.
public interface PersistenceService<T extends Entity> {
  public void save(T entity);
}

public interface UserPersistenceService implements PersistenceService<User>{
  @Override
  public void save(User entity){
    // save user
  }

  public void findByName(string name){
    // get user by name
  }
}

public interface OrderPersistenceService implements PersistenceService<Order>{
  @Override
  public void save(Order entity){
    // save Order
  }
}
```

5. **Dependency Inversion Principle**

    - High Level modules (a module that provides or implements business logic) should not depend on low level modules (a functionality/module that can be used anywhere, this is independent from business logc). Both should depend upon abstraction.
    - Abstraction should not depend upon details. Details should depend upon abstraction

```
import java.io.FileWriter;

public class Main {
  public static void main(String[] args) throws IOException {
    Message msg = new Message("This is a message again");
    MessagePrinter printer = new MessagePrinter();
    printer.writeMessage(msg, "test_msg.txt");
  }
}

public class MessagePrinter {
  //Writes message to a file
  public void writeMessage(Message msg, String fileName) throws IOException {
    Formatter formatter = new JSONFormatter();//creates formatter
    try (PrintWriter writer = new PrintWriter(new FileWriter(fileName))) { //creates print writer
      writer.println(formatter.format(msg)); //formats and writes message
      writer.flush();
    }
  }
}

//Common interface for classes formatting Message object
public interface Formatter {
  public String format(Message message) throws FormatException;
}

//Formats message to JSON format
public class JSONFormatter implements Formatter {
  public String format(Message message) throws FormatException {
    ObjectMapper mapper = new ObjectMapper();
    try {
      return mapper.writeValueAsString(message);
    } catch (JsonProcessingException e) {
      e.printStackTrace();
      throw new FormatException(e);
    }
  }
}

// Now, if we want to write message to json file or to print in console, we'll need to change this method again
// Here, we can clearly see that MessagePrinter (High level class - which has business logic) is dependent
on JSONFormatter and FileWriter(low level module/class)

// Instead we can modify this to follow dependency inversion principle.
public class Main {
  public static void main(String[] args) throws IOException {
    Message msg = new Message("This is a new message");
    MessagePrinter printer = new MessagePrinter();
    try (PrintWriter writer = new PrintWriter(System.out)) {
      printer.writeMessage(msg, new JSONFormatter(), writer);
    }
    try (PrintWriter writer = new PrintWriter(new FileWriter("test_msg.txt"))) {
      printer.writeMessage(msg, new JSONFormatter(), writer);
    }
  }
}

public class MessagePrinter {
  //Writes message to a file
  public void writeMessage(Message msg, Formatter formatter, PrintWriter writer) throws IOException {
    writer.println(formatter.format(msg)); //formats and writes message
    writer.flush();
  }
}
```

```
// This example follows dependency inversion i.e., we are nor creating dependencies. Our MessagePrinter is
not dependent on anything.
// Instead, someone else is creating those dependencies and passing them to out MessagePrinter
```