Name: Kaustubh Dhongade

Reg no.: 231070018

Batch: A

Lab-6

Aim:

Task -1 : Read and understand SOLID principles of software development. Write a sample code for each principle.

- **Single Responsibility Principle (SRP)** is one of the core principles of object-oriented design. It states that a class should have only one responsibility, meaning it should focus on a single task or functionality. If a class has multiple responsibilities, changes in one responsibility might impact the other, leading to tightly coupled code that is harder to understand, maintain, and test.

Code:

```cpp
 #include <iostream>
#include <string>

// Class responsible for handling student grades
class Grade {
public:
    std::string grade;
    Grade(std::string g) : grade(g) {}
};

// Class responsible for displaying information (separate responsibility)
class GradeDisplay {
public:
    void displayGrade(const Grade& g) {
        std::cout << "Student grade: " << g.grade << std::endl;
```

```
    }
};

int main() {
    Grade studentGrade("AA");
    GradeDisplay display;
    display.displayGrade(studentGrade);
    return 0;
}
```

- The **Open/Closed Principle (OCP)** states that software entities (classes, modules, functions) should be **open for extension but closed for modification**. This means you can add new functionality without changing existing code, promoting flexibility and reducing the risk of breaking existing features.

Code:

```
#include <iostream>

// Base class with a virtual method
class Shape {
public:
    virtual double area() const = 0; // pure virtual function
    virtual ~Shape() = default;
};

// Circle class extending Shape
class Circle : public Shape {
private:
    double radius;
public:
    Circle(double r) : radius(r) {}
    double area() const override {
        return 3.14 * radius * radius;
    }
};

// Rectangle class extending Shape
class Rectangle : public Shape {
private:
    double width, height;
public:
    Rectangle(double w, double h) : width(w), height(h) {}
```

```
    double area() const override {
        return width * height;
    }
};

int main() {
    Circle c(5);
    Rectangle r(3, 4);

    std::cout << "Area of Circle: " << c.area() << std::endl;
    std::cout << "Area of Rectangle: " << r.area() << std::endl;

    return 0;
}
```

- **Liskov Substitution Principle (LSP)**

Objects of a subclass should be able to replace objects of the parent class without altering the correctness of the program.

```
#include <iostream>

// Base class
class Bird {
public:
    virtual void fly() {
        std::cout << "Flying" << std::endl;
    }
};

// Derived class
class Sparrow : public Bird {
public:
    void fly() override {
        std::cout << "Sparrow flying" << std::endl;
    }
};

// Derived class
class Penguin : public Bird {
public:
    void fly() override {
        // Penguins can't fly, so we might break LSP
        std::cout << "Penguin can't fly" << std::endl;
    }
};
```

```
int main() {
    Bird* b1 = new Sparrow();
    b1->fly();  // Sparrow flying

    Bird* b2 = new Penguin();
    b2->fly();  // Penguin can't fly

    delete b1;
    delete b2;

    return 0;
}
```

- **Interface Segregation Principle (ISP)** Clients should not be forced to depend on interfaces they do not use. It encourages the creation of small, focused interfaces.

```
#include <iostream>

// Interface for flying animals
class IFlyable {
public:
    virtual void fly() = 0;
};

// Interface for swimming animals
class ISwimmable {
public:
    virtual void swim() = 0;
};

// Bird class implementing IFlyable
class Bird : public IFlyable {
public:
    void fly() override {
        std::cout << "Bird is flying" << std::endl;
    }
};

// Fish class implementing ISwimmable
class Fish : public ISwimmable {
public:
    void swim() override {
        std::cout << "Fish is swimming" << std::endl;
    }
};
```

```
•
•  int main() {
•      Bird bird;
•      Fish fish;
•
•      bird.fly();  // Bird is flying
•      fish.swim(); // Fish is swimming
•
•      return 0;
•  }
```

- **Dependency Inversion Principle (DIP)**

High-level modules should not depend on low-level modules. Both should depend on abstractions. Additionally, abstractions should not depend on details; details should depend on abstractions.

```cpp
#include <iostream>

// Abstraction (interface)
class IPrinter {
public:
    virtual void print() = 0;
};

// Low-level module
class LaserPrinter : public IPrinter {
public:
    void print() override {
        std::cout << "Printing with Laser Printer" << std::endl;
    }
};

// High-level module
class Document {
private:
    IPrinter* printer;
public:
    Document(IPrinter* p) : printer(p) {}
    void print() {
        printer->print();
    }
};

int main() {
    LaserPrinter laserPrinter;
    Document doc(&laserPrinter);
```

```
doc.print();  // Printing with Laser Printer
return 0;
}
```