Name : Atharva Dhage
Reg num: 231070017
Batch: A

DAA Lab 06

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

1. Single Responsibility Principle (SRP)

Each class should have only one reason to change, meaning it should have only one responsibility or task.
Imagine a User class that handles both data storage (saving to a database) and email communication (sending a welcome email). According to SRP, these are two distinct responsibilities: data handling and email communication. By splitting them into separate classes
(e.g., UserDatabase for storage and EmailService for communication), each class now has a
single, clear responsibility, which makes them easier to maintain and reuse

Code:

```
#include <iostream>
#include <string>
using namespace std;
// Violates SRP
class User
{
public:
    string name;
    string email;
    User(string n, string e) : name(n), email(e) {}
    void saveToDatabase()
    {
        cout << "Saving " << name << " to the database...\n";
    }
    void sendWelcomeEmail()
    {
        cout << "Sending welcome email to " << email << "...\n";
    }
};
// Applies SRP
class User
```

```
{
public:
    string name;
    string email;
    User(string n, string e) : name(n), email(e) {}
};
class UserDatabase
{
public:
    void save(User user)
    {
        cout << "Saving " << user.name << " to the database...\n";
    }
};
class EmailService
{
public:
    void sendWelcomeEmail(User user)
    {
        cout << "Sending welcome email to " << user.email << "...\n";
    }
};
```

2. Open/Closed Principle (OCP)

Software entities (classes, modules, functions) should be open for extension but closed for modification.
Suppose we have a notification system with a Notification base class. We want the system to be open to new notification types like SMS, push notifications, or emails without modifying the original Notification class. By defining a base Notification class and implementing subclasses like ‘EmailNotification’ and ‘SMSNotification’, we add new types of notifications without altering the existing ones. This design is scalable and minimises code alterations, thereby maintaining stability.

Code:

```
#include <iostream>
#include <string>
using namespace std;
// Base class for Notification
class Notification
{
public:
    virtual void send(string message) = 0; // Pure virtual function
};
class EmailNotification : public Notification
{
public:
    void send(string message) override
    {
        cout << "Sending Email with message: " << message << endl;
    }
```

```
};
class SMSNotification : public Notification
{
public:
    void send(string message) override
    {
        cout << "Sending SMS with message: " << message << endl;
    }
};
// Client code
void notify(Notification *notification, string message)
{
    notification->send(message);
}
int main()
{
    EmailNotification email;
    SMSNotification sms;
    notify(&email, "Hello via Email!");
    notify(&sms, "Hello via SMS!");
    return 0;
}
```

3. Liskov Substitution Principle (LSP)

Objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program. Imagine we have a Bird class with a fly method. If we create a Sparrow class that can fly, it’s valid to replace Bird with Sparrow. However, if we add an Ostrich class that cannot fly but still inherits from Bird, it breaks LSP because it changes the expected behaviour of fly. By introducing a FlyingBird subclass for birds that can fly, we can separate the behaviours, maintaining LSP compliance since each subclass now has a behaviour that is predictable and consistent with its type.

Code:

```
#include <iostream>
using namespace std;
// Base class
class Bird
{
public:
    virtual void fly()
    {
        cout << "I can fly!" << endl;
    }
};
```

```cpp
// Subclass which can fly
class Sparrow : public Bird
{
};
// Subclass which cannot fly (breaks LSP if used with Bird pointer)
class Ostrich : public Bird
{
public:
    void fly() override
    {
        throw "Cannot fly!";
    }
};
// Correct implementation (LSP compliant)
class Bird
{
public:
    virtual void fly() = 0; // Pure virtual
};
class FlyingBird : public Bird
{
public:
    void fly() override
    {
        cout << "I can fly!" << endl;
    }
};
class Sparrow : public FlyingBird
{
};
class Ostrich : public Bird
{
public:
    void fly() override
    {
        cout << "I cannot fly!" << endl;
    }
};
int main()
{
    Sparrow sparrow;
    Ostrich ostrich;
    Bird *bird1 = &sparrow;
    Bird *bird2 = &ostrich;
    bird1->fly(); // Works fine
    bird2->fly(); // Works fine, no exception
    return 0;
}
```

4. Interface Segregation Principle (ISP)

Clients should not be forced to depend on interfaces they do not use.
Consider a Machine interface that includes print, scan, and fax methods. A class that only needs print capabilities would still be forced to implement scan and fax methods, violating ISP. By breaking down Machine into separate Printer, Scanner, and Fax interfaces, classes can now implement only the interfaces they require, avoiding irrelevant dependencies and simplifying the design.

Code:

```
#include <iostream>
#include <string>
using namespace std;
// Violates ISP - a single interface with unrelated methods
class Machine
{
public:
    virtual void print(string document) = 0;
    virtual void scan(string document) = 0;
    virtual void fax(string document) = 0;
};
// Correct implementation with separate interfaces
class Printer
{
public:
    virtual void print(string document) = 0;
};
class Scanner
{
public:
    virtual void scan(string document) = 0;
};
class Fax
{
public:
    virtual void fax(string document) = 0;
};
// Implementing specific functionalities
class MultiFunctionPrinter : public Printer, public Scanner, public Fax
{
public:
    void print(string document) override
    {
        cout << "Printing: " << document << endl;
    }
    void scan(string document) override
```

```
    {
        cout << "Scanning: " << document << endl;
    }
    void fax(string document) override
    {
        cout << "Faxing: " << document << endl;
    }
};
class SimplePrinter : public Printer
{
public:
    void print(string document) override
    {
        cout << "Printing: " << document << endl;
    }
};
```

5. Dependency Inversion Principle (DIP)

High-level modules should not depend on low-level modules; both should depend on abstractions. Abstractions should not depend on details; details should depend on Abstractions. Let's say we have a DataAccess class responsible for retrieving data. Instead of directly depending on a specific database implementation like MySQLDatabase, which is a low-level detail, we introduce an abstract Database interface that both MySQLDatabase and PostgreSQLDatabase can implement. The DataAccess class then depends on Database, an abstraction, instead of a concrete database class. This makes it easy to swap out or add different database implementations without modifying DataAccess, maintaining a flexible and extensible Design.

Code:

```
#include <iostream>
using namespace std;
// Abstraction
class Database
{
public:
    virtual void connect() = 0;
};
// Low-level module
class MySQLDatabase : public Database
{
public:
    void connect() override
    {
        cout << "Connecting to MySQL Database" << endl;
    }
```

```
};
class PostgreSQLDatabase : public Database
{
public:
    void connect() override
    {
        cout << "Connecting to PostgreSQL Database" << endl;
    }
};
// High-level module
class DataAccess
{
private:
    Database *database;

public:
    DataAccess(Database *db) : database(db) {}
    void getData()
    {
        database->connect();
        cout << "Fetching data" << endl;
    }
};
int main()
{
    MySQLDatabase mysqlDb;
    PostgreSQLDatabase postgresDb;
    DataAccess dataAccess1(&mysqlDb);
    DataAccess dataAccess2(&postgresDb);
    dataAccess1.getData();
    dataAccess2.getData();
    return 0;
}
```