Solid Principles

a. Single Responsibility Principle (SRP):

This principle states that "**A class should have only one reason to change**" which means every class should have a single responsibility or single job or single purpose. In other words, a class should have only one job or purpose within the software system.

o Example: Class DataContainer handles data storage, while DataProcessor manages data processing. This structure ensures each class has a single responsibility.

Example Code:

```
#include <iostream>
#include <string>
using namespace std;
// Class representing user data
class User {
public:
string name;
string email;
User(string n, string e) : name(n), email(e) {}
};
// Class responsible for database operations related to User
class UserDatabase {
public:
void save(const User& user) {
cout << "Saving " << user.name << " to the database...\n";
}
};
// Class responsible for email operations related to User
class EmailService {
public:
void sendWelcomeEmail(const User& user) {
cout << "Sending welcome email to " << user.email <<
"...\n";
}
};
// Main function to demonstrate SRP-compliant design
int main() {
```

```
// Create a new user
User user("dhruv", "dhruvp2005@gmail.com");
// Instantiate services
UserDatabase;
EmailService;
// Save user to the database and send a welcome email
userDatabase.save(user);
emailService.sendWelcomeEmail(user);
return 0;
}
```

b. Open/Closed Principle (OCP):

This principle states that "**Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification**" which means you should be able to extend a class behavior, without modifying it

o Example: CoreFeature provides the essential framework, while ExtendedFeature1 and ExtendedFeature2 add new functionalities by inheriting CoreFeature. This enables feature additions without changing existing structures.

Example 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;
}
```

c. Liskov Substitution Principle (LSP):

The principle was introduced by Barbara Liskov in 1987 and according to this principle “**Derived or child classes must be substitutable for their base or parent classes**“. This principle ensures that any class that is the child of a parent class should be usable in place of its parent without any unexpected behavior.

o Example: GeneralShape defines common methods, and SpecificShape1 and SpecificShape2 extend it. This allows these subclasses to replace GeneralShape without compatibility issues.\

Example Code:

```
#include <iostream>
using namespace std;
// Base class
class Bird {
public:
virtual void fly() {
cout << "I can fly!" << endl;
}
};
// 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;
Ostrich;
Bird* bird1 = &sparrow;
Bird* bird2 = &ostrich;
bird1->fly(); // Works fine
bird2->fly(); // Works fine, no exception
return 0;
}
```

d. Interface Segregation Principle (ISP):

This principle is the first principle that applies to Interfaces instead of classes in SOLID and it is similar to the single responsibility principle. It states that "**do not force any client to implement an interface which is irrelevant to them**". You should prefer many client interfaces rather than one general interface and each interface should have a specific responsibility.

o Example: Instead of a broad interface containing many methods, creating smaller, targeted interfaces keeps dependencies relevant and manageable.

Example Code:

```
#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;
}
};
```

e. Dependency Inversion Principle (DIP):

The Dependency Inversion Principle (DIP) is a principle in object-oriented design that states that "**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.

o Example: MainModule interacts with AbstractionLayer, which is an interface, avoiding a dependency on specific implementations like ConcreteType1. This flexibility allows for easy implementation changes.

Example 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;
}
```