6. SOLID Principles

6.1 Introduction to SOLID

SOLID is an acronym for five design principles that make software designs more understandable, flexible, and maintainable.

Letter	Principle	Core Idea
S	Single Responsibility	A class should have one reason to change
O	Open/Closed	Open for extension, closed for modification
L	Liskov Substitution	Derived classes must be substitutable for base classes
I	Interface Segregation	Many specific interfaces are better than one general interface
D	Dependency Inversion	Depend on abstractions, not concretions

6.2 S - Single Responsibility Principle (SRP)

Definition: A class should have only one reason to change, meaning it should have only one job or responsibility.

Bad Example (Multiple Responsibilities):

```
// This class has too many responsibilities!
class Employee {
private:
    string name;
    double salary;

public:
    // Responsibility 1: Employee data management
    void setName(string n) { name = n; }
    string getName() { return name; }
    void setSalary(double s) { salary = s; }
```

```
    double getSalary() { return salary; }

    // Responsibility 2: Salary calculation
    double calculateTax() {
        return salary * 0.2;
    }

    double calculateBonus() {
        return salary * 0.1;
    }

    // Responsibility 3: Database operations
    void saveToDatabase() {
        cout << "Saving employee to database..." << endl;
    }

    void loadFromDatabase() {
        cout << "Loading employee from database..." << endl;
    }

    // Responsibility 4: Report generation
    void printPaySlip() {
        cout << "Printing pay slip..." << endl;
    }
};
```

Good Example (Single Responsibility):

```
// Each class has ONE clear responsibility

// 1. Employee data management
class Employee {
private:
    string name;
    string id;
    double salary;

public:
    Employee(string n, string i, double s)
        : name(n), id(i), salary(s) {}
```

```
    string getName() const { return name; }
    string getId() const { return id; }
    double getSalary() const { return salary; }
    void setSalary(double s) { salary = s; }
};

// 2. Salary calculations
class SalaryCalculator {
public:
    double calculateTax(const Employee& emp) {
        return emp.getSalary() * 0.2;
    }

    double calculateBonus(const Employee& emp) {
        return emp.getSalary() * 0.1;
    }

    double calculateNetSalary(const Employee& emp) {
        return emp.getSalary() - calculateTax(emp) + calculateBonus(emp);
    }
};

// 3. Database operations
class EmployeeRepository {
public:
    void save(const Employee& emp) {
        cout << "Saving employee " << emp.getId() << " to database..." << endl;
    }

    Employee* load(string id) {
        cout << "Loading employee " << id << " from database..." << endl;
        return nullptr;  // Simplified
    }
};

// 4. Report generation
class PaySlipGenerator {
private:
    SalaryCalculator calculator;
```

```
public:
    void generatePaySlip(const Employee& emp) {
        cout << "\n===== PAY SLIP =====" << endl;
        cout << "Employee: " << emp.getName() << endl;
        cout << "ID: " << emp.getId() << endl;
        cout << "Gross Salary: $" << emp.getSalary() << endl;
        cout << "Tax: $" << calculator.calculateTax(emp) << endl;
        cout << "Bonus: $" << calculator.calculateBonus(emp) << endl;
        cout << "Net Salary: $" << calculator.calculateNetSalary(emp) << endl;
        cout << "====================" << endl;
    }
};

int main() {
    Employee emp("Alice Johnson", "E001", 5000.0);

    SalaryCalculator calc;
    EmployeeRepository repo;
    PaySlipGenerator payslip;

    payslip.generatePaySlip(emp);
    repo.save(emp);

    return 0;
}
```

Benefits:

- Easier to understand and maintain
- Changes in one responsibility don't affect others
- Easier to test individual components
- Better code organization

6.3 O - Open/Closed Principle (OCP)

Definition: Software entities should be **open for extension** but **closed for modification**. You should be able to add new functionality without changing existing code.

Bad Example (Violates OCP):

```
class Rectangle {
public:
    double width, height;
};

class Circle {
public:
    double radius;
};

// This class needs modification every time we add a new shape
class AreaCalculator {
public:
    double calculateArea(void* shape, string shapeType) {
        if (shapeType == "Rectangle") {
            Rectangle* rect = (Rectangle*)shape;
            return rect->width * rect->height;
        }
        else if (shapeType == "Circle") {
            Circle* circle = (Circle*)shape;
            return 3.14159 * circle->radius * circle->radius;
        }
        // Need to modify this function for every new shape!
        // else if (shapeType == "Triangle") { ... }
        return 0;
    }
};
```

Good Example (Follows OCP):

```
#include <iostream>
#include <vector>
#include <memory>
using namespace std;

// Abstract base class
class Shape {
public:
    virtual double calculateArea() = 0;
    virtual string getName() = 0;
```

```
    virtual ~Shape() {}
};

// Concrete shapes - extending without modifying existing code
class Rectangle : public Shape {
private:
    double width, height;

public:
    Rectangle(double w, double h) : width(w), height(h) {}

    double calculateArea() override {
        return width * height;
    }

    string getName() override {
        return "Rectangle";
    }
};

class Circle : public Shape {
private:
    double radius;

public:
    Circle(double r) : radius(r) {}

    double calculateArea() override {
        return 3.14159 * radius * radius;
    }

    string getName() override {
        return "Circle";
    }
};

// NEW shape - no modification to existing code needed!
class Triangle : public Shape {
private:
    double base, height;
```

```
public:
    Triangle(double b, double h) : base(b), height(h) {}

    double calculateArea() override {
        return 0.5 * base * height;
    }

    string getName() override {
        return "Triangle";
    }
};

// This class doesn't need modification when adding new shapes
class AreaCalculator {
public:
    void printArea(Shape* shape) {
        cout << shape->getName() << " area: "
             << shape->calculateArea() << endl;
    }

    double getTotalArea(vector<Shape*> shapes) {
        double total = 0;
        for (Shape* shape : shapes) {
            total += shape->calculateArea();
        }
        return total;
    }
};

int main() {
    Rectangle rect(5, 3);
    Circle circle(4);
    Triangle triangle(6, 8);

    AreaCalculator calculator;

    calculator.printArea(&rect);
    calculator.printArea(&circle);
    calculator.printArea(&triangle);
```

```
    vector<Shape*> shapes = {&rect, &circle, &triangle};
    cout << "Total area: " << calculator.getTotalArea(shapes) << endl;

    return 0;
}
```

Benefits:

- Add new features without breaking existing code
- Reduces risk of introducing bugs
- Promotes code reuse through inheritance
- Makes the system more maintainable

6.4 L - Liskov Substitution Principle (LSP)

Definition: Objects of a derived class should be able to replace objects of the base class without breaking the application. In other words, derived classes must be substitutable for their base classes.

Bad Example (Violates LSP):

```
class Bird {
public:
    virtual void fly() {
        cout << "Flying..." << endl;
    }
};

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

// Ostrich is a bird but can't fly!
class Ostrich : public Bird {
public:
    void fly() override {
        throw runtime_error("Ostrich can't fly!");
```

```
        // This breaks LSP - can't substitute Ostrich for Bird
    }
};

void makeBirdFly(Bird* bird) {
    bird->fly();  // Will crash if bird is an Ostrich!
}
```

Good Example (Follows LSP):

```
#include <iostream>
#include <string>
using namespace std;

// Better abstraction
class Bird {
protected:
    string name;

public:
    Bird(string n) : name(n) {}

    virtual void eat() {
        cout << name << " is eating..." << endl;
    }

    virtual void makeSound() = 0;

    string getName() { return name; }
};

// Separate interface for flying ability
class FlyingBird : public Bird {
public:
    FlyingBird(string n) : Bird(n) {}

    virtual void fly() {
        cout << name << " is flying..." << endl;
    }
};
```

```
// Flying birds
class Sparrow : public FlyingBird {
public:
    Sparrow() : FlyingBird("Sparrow") {}

    void makeSound() override {
        cout << "Chirp chirp!" << endl;
    }
};

class Eagle : public FlyingBird {
public:
    Eagle() : FlyingBird("Eagle") {}

    void makeSound() override {
        cout << "Screech!" << endl;
    }
};

// Non-flying bird - doesn't inherit fly()
class Ostrich : public Bird {
public:
    Ostrich() : Bird("Ostrich") {}

    void makeSound() override {
        cout << "Boom boom!" << endl;
    }

    void run() {
        cout << name << " is running fast..." << endl;
    }
};

class Penguin : public Bird {
public:
    Penguin() : Bird("Penguin") {}

    void makeSound() override {
        cout << "Honk honk!" << endl;
```

```
    }

    void swim() {
        cout << name << " is swimming..." << endl;
    }
};

int main() {
    Sparrow sparrow;
    Eagle eagle;
    Ostrich ostrich;
    Penguin penguin;

    // All birds can make sounds and eat
    Bird* birds[] = {&sparrow, &eagle, &ostrich, &penguin};

    cout << "=== All birds can do these ===" << endl;
    for (Bird* bird : birds) {
        bird->makeSound();
        bird->eat();
        cout << endl;
    }

    // Only flying birds can fly
    cout << "=== Only flying birds ===" << endl;
    FlyingBird* flyingBirds[] = {&sparrow, &eagle};

    for (FlyingBird* bird : flyingBirds) {
        bird->fly();
    }

    // Specialized behaviors
    cout << "\n=== Specialized behaviors ===" << endl;
    ostrich.run();
    penguin.swim();

    return 0;
}
```

Real-World Example:

```
// Bad: Square inheriting from Rectangle violates LSP
class Rectangle {
protected:
    double width, height;

public:
    virtual void setWidth(double w) { width = w; }
    virtual void setHeight(double h) { height = h; }
    double getArea() { return width * height; }
};

class Square : public Rectangle {
public:
    void setWidth(double w) override {
        width = height = w;  // Problem: setting width also sets height!
    }

    void setHeight(double h) override {
        width = height = h;  // Problem: setting height also sets width!
    }
};

// This function expects Rectangle behavior
void processRectangle(Rectangle* rect) {
    rect->setWidth(5);
    rect->setHeight(4);
    // Expected area: 20
    // But if rect is a Square, area will be 16!
    cout << "Area: " << rect->getArea() << endl;
}
```

Better Design:

```
class Shape {
public:
    virtual double getArea() = 0;
    virtual ~Shape() {}
};

class Rectangle : public Shape {
```

```
private:
    double width, height;

public:
    Rectangle(double w, double h) : width(w), height(h) {}

    void setWidth(double w) { width = w; }
    void setHeight(double h) { height = h; }

    double getArea() override {
        return width * height;
    }
};

class Square : public Shape {
private:
    double side;

public:
    Square(double s) : side(s) {}

    void setSide(double s) { side = s; }

    double getArea() override {
        return side * side;
    }
};
```

Benefits:

- Ensures polymorphism works correctly
- Prevents unexpected behavior in derived classes
- Makes code more reliable and predictable

6.5 I - Interface Segregation Principle (ISP)

Definition: No client should be forced to depend on methods it does not use. It's better to have many specific interfaces than one general-purpose interface.

Bad Example (Violates ISP):

```
// Fat interface - forces classes to implement methods they don't need
class Worker {
public:
    virtual void work() = 0;
    virtual void eat() = 0;
    virtual void sleep() = 0;
    virtual void attendMeeting() = 0;
    virtual void writeCode() = 0;
    virtual void designUI() = 0;
};

// Robot worker doesn't eat or sleep!
class RobotWorker : public Worker {
public:
    void work() override {
        cout << "Robot working..." << endl;
    }

    void eat() override {
        // Robots don't eat! But forced to implement
        throw runtime_error("Robots don't eat!");
    }

    void sleep() override {
        // Robots don't sleep! But forced to implement
        throw runtime_error("Robots don't sleep!");
    }

    void attendMeeting() override {
        throw runtime_error("Robots don't attend meetings!");
    }

    void writeCode() override {
        cout << "Robot writing code..." << endl;
    }

    void designUI() override {
        throw runtime_error("Robots don't design UI!");
    }
};
```

```
// Manager doesn't write code!
class Manager : public Worker {
public:
    void work() override {
        cout << "Manager working..." << endl;
    }

    void eat() override {
        cout << "Manager eating..." << endl;
    }

    void sleep() override {
        cout << "Manager sleeping..." << endl;
    }

    void attendMeeting() override {
        cout << "Manager attending meeting..." << endl;
    }

    void writeCode() override {
        // Managers don't write code! But forced to implement
        throw runtime_error("Managers don't write code!");
    }

    void designUI() override {
        throw runtime_error("Managers don't design UI!");
    }
};
```

Good Example (Follows ISP):

```
#include <iostream>
#include <string>
using namespace std;

// Segregated interfaces - small, specific interfaces

interface Workable {
public:
```

```
    virtual void work() = 0;
    virtual ~Workable() {}
};

class Eatable {
public:
    virtual void eat() = 0;
    virtual ~Eatable() {}
};

class Sleepable {
public:
    virtual void sleep() = 0;
    virtual ~Sleepable() {}
};

class Codable {
public:
    virtual void writeCode() = 0;
    virtual ~Codable() {}
};

class Designable {
public:
    virtual void designUI() = 0;
    virtual ~Designable() {}
};

class Manageable {
public:
    virtual void attendMeeting() = 0;
    virtual void delegateTasks() = 0;
    virtual ~Manageable() {}
};

// Now classes only implement interfaces they need

class Developer : public Workable, public Eatable, public Sleepable, public Codable {
private:
    string name;
```

```
public:
    Developer(string n) : name(n) {}

    void work() override {
        cout << name << " (Developer) is working..." << endl;
    }

    void eat() override {
        cout << name << " is eating..." << endl;
    }

    void sleep() override {
        cout << name << " is sleeping..." << endl;
    }

    void writeCode() override {
        cout << name << " is writing code..." << endl;
    }
};

class Designer : public Workable, public Eatable, public Sleepable, public Designable {
private:
    string name;

public:
    Designer(string n) : name(n) {}

    void work() override {
        cout << name << " (Designer) is working..." << endl;
    }

    void eat() override {
        cout << name << " is eating..." << endl;
    }

    void sleep() override {
        cout << name << " is sleeping..." << endl;
    }
```

```
    void designUI() override {
        cout << name << " is designing UI..." << endl;
    }
};

class Manager : public Workable, public Eatable, public Sleepable, public Manageable {
private:
    string name;

public:
    Manager(string n) : name(n) {}

    void work() override {
        cout << name << " (Manager) is working..." << endl;
    }

    void eat() override {
        cout << name << " is eating..." << endl;
    }

    void sleep() override {
        cout << name << " is sleeping..." << endl;
    }

    void attendMeeting() override {
        cout << name << " is attending meeting..." << endl;
    }

    void delegateTasks() override {
        cout << name << " is delegating tasks..." << endl;
    }
};

class RobotWorker : public Workable, public Codable {
private:
    string model;

public:
    RobotWorker(string m) : model(m) {}
```

```
    void work() override {
        cout << model << " (Robot) is working 24/7..." << endl;
    }

    void writeCode() override {
        cout << model << " is writing code..." << endl;
    }
    // No eat() or sleep() - robots don't need these!
};

int main() {
    Developer dev("Alice");
    Designer des("Bob");
    Manager mgr("Carol");
    RobotWorker robot("R2D2");

    cout << "=== Developer ===" << endl;
    dev.work();
    dev.writeCode();
    dev.eat();

    cout << "\n=== Designer ===" << endl;
    des.work();
    des.designUI();
    des.sleep();

    cout << "\n=== Manager ===" << endl;
    mgr.work();
    mgr.attendMeeting();
    mgr.delegateTasks();

    cout << "\n=== Robot ===" << endl;
    robot.work();
    robot.writeCode();
    // robot.eat();  // Doesn't exist - good!

    return 0;
}
```

Benefits:

- Classes only depend on methods they actually use
- More flexible and maintainable code
- Easier to understand class responsibilities
- Reduces coupling between classes

6.6 D - Dependency Inversion Principle (DIP)

Definition:

1. High-level modules should not depend on low-level modules. Both should depend on abstractions.
2. Abstractions should not depend on details. Details should depend on abstractions.

Bad Example (Violates DIP):

```
// Low-level modules (concrete implementations)
class MySQLDatabase {
public:
    void connect() {
        cout << "Connecting to MySQL..." << endl;
    }

    void executeQuery(string query) {
        cout << "Executing MySQL query: " << query << endl;
    }
};

// High-level module directly depends on low-level module
class UserService {
private:
    MySQLDatabase database;  // Direct dependency on concrete class!

public:
    void getUser(int id) {
        database.connect();
        database.executeQuery("SELECT * FROM users WHERE id = " + to_string(id));
    }

    // If we want to switch to PostgreSQL, we must modify this class!
};
```

Good Example (Follows DIP):

```
#include <iostream>
#include <string>
#include <memory>
using namespace std;

// Abstraction (high-level interface)
class Database {
public:
    virtual void connect() = 0;
    virtual void executeQuery(string query) = 0;
    virtual ~Database() {}
};

// Low-level modules implement the abstraction
class MySQLDatabase : public Database {
public:
    void connect() override {
        cout << "Connecting to MySQL database..." << endl;
    }

    void executeQuery(string query) override {
        cout << "MySQL executing: " << query << endl;
    }
};

class PostgreSQLDatabase : public Database {
public:
    void connect() override {
        cout << "Connecting to PostgreSQL database..." << endl;
    }

    void executeQuery(string query) override {
        cout << "PostgreSQL executing: " << query << endl;
    }
};

class MongoDatabase : public Database {
public:
```

```
    void connect() override {
        cout << "Connecting to MongoDB..." << endl;
    }

    void executeQuery(string query) override {
        cout << "MongoDB executing: " << query << endl;
    }
};

// High-level module depends on abstraction, not concrete class
class UserService {
private:
    Database* database;  // Depends on abstraction!

public:
    // Dependency injection through constructor
    UserService(Database* db) : database(db) {}

    void getUser(int id) {
        database->connect();
        database->executeQuery("SELECT * FROM users WHERE id = " + to_string(id));
    }

    void saveUser(string name, string email) {
        database->connect();
        database->executeQuery("INSERT INTO users (name, email) VALUES ('" +
                               name + "', '" + email + "')");
    }

    // Can easily switch database implementation!
    void setDatabase(Database* db) {
        database = db;
    }
};

int main() {
    // Create different database implementations
    MySQLDatabase mysql;
    PostgreSQLDatabase postgres;
    MongoDatabase mongo;
```

```
    // Inject dependency (database) into high-level module
    cout << "=== Using MySQL ===" << endl;
    UserService userService1(&mysql);
    userService1.getUser(1);
    userService1.saveUser("Alice", "alice@example.com");

    cout << "\n=== Switching to PostgreSQL ===" << endl;
    UserService userService2(&postgres);
    userService2.getUser(2);

    cout << "\n=== Switching to MongoDB ===" << endl;
    UserService userService3(&mongo);
    userService3.getUser(3);

    return 0;
}
```

Another Example: Payment Processing

```
#include <iostream>
#include <string>
using namespace std;

// Abstraction
class PaymentProcessor {
public:
    virtual bool processPayment(double amount) = 0;
    virtual string getProcessorName() = 0;
    virtual ~PaymentProcessor() {}
};

// Concrete implementations
class CreditCardProcessor : public PaymentProcessor {
public:
    bool processPayment(double amount) override {
        cout << "Processing $" << amount << " via Credit Card..." << endl;
        return true;
    }
```

```
    string getProcessorName() override {
        return "Credit Card";
    }
};

class PayPalProcessor : public PaymentProcessor {
public:
    bool processPayment(double amount) override {
        cout << "Processing $" << amount << " via PayPal..." << endl;
        return true;
    }

    string getProcessorName() override {
        return "PayPal";
    }
};

class BitcoinProcessor : public PaymentProcessor {
public:
    bool processPayment(double amount) override {
        cout << "Processing $" << amount << " via Bitcoin..." << endl;
        return true;
    }

    string getProcessorName() override {
        return "Bitcoin";
    }
};

// High-level module depends on abstraction
class ShoppingCart {
private:
    double total;
    PaymentProcessor* paymentProcessor;

public:
    ShoppingCart() : total(0), paymentProcessor(nullptr) {}

    void addItem(double price) {
        total += price;
```

```
        cout << "Item added. Current total: $" << total << endl;
    }

    void setPaymentMethod(PaymentProcessor* processor) {
        paymentProcessor = processor;
        cout << "Payment method set to: " << processor->getProcessorName() << endl;
    }

    void checkout() {
        if (paymentProcessor == nullptr) {
            cout << "Error: No payment method selected!" << endl;
            return;
        }

        cout << "\n=== Checkout ===" << endl;
        cout << "Total amount: $" << total << endl;

        if (paymentProcessor->processPayment(total)) {
            cout << "Payment successful!" << endl;
            total = 0;
        } else {
            cout << "Payment failed!" << endl;
        }
    }
};

int main() {
    CreditCardProcessor creditCard;
    PayPalProcessor paypal;
    BitcoinProcessor bitcoin;

    ShoppingCart cart;

    cart.addItem(29.99);
    cart.addItem(49.99);
    cart.addItem(15.50);

    cout << "\n--- Paying with Credit Card ---" << endl;
    cart.setPaymentMethod(&creditCard);
    cart.checkout();
```

```
    cout << "\n--- New purchase ---" << endl;
    cart.addItem(99.99);
    cout << "\n--- Paying with PayPal ---" << endl;
    cart.setPaymentMethod(&paypal);
    cart.checkout();

    cout << "\n--- Another purchase ---" << endl;
    cart.addItem(199.99);
    cout << "\n--- Paying with Bitcoin ---" << endl;
    cart.setPaymentMethod(&bitcoin);
    cart.checkout();

    return 0;
}
```

Benefits:

- Easy to switch implementations
- Reduces coupling between modules
- More testable code (can inject mock dependencies)
- Promotes code reuse and flexibility

Revision #2
Created 2025-11-09 13:19:33 UTC by DS
Updated 2025-11-09 13:21:20 UTC by DS