# SOLID Design Principles

Instructor: Mehroze Khan

# SOLID Principles

- Good software systems begin with clean code
- The SOLID principles tell us how to **arrange our functions and data structures** into classes, and how those classes should be **interconnected**
- The goal of the principles is the creation of mid-level software structures that:
  - Tolerate change
  - Are easy to understand
  - Are the basis of components that can be used in many software systems
- The term “mid-level” refers to the fact that these principles are applied by programmers working at the module level.

# SOLID Principles

- **SRP:** Single Responsibility Principle
- **OCP:** Open-Closed Principle
- **LSP:** Liskov Substitution Principle
- **ISP:** Interface Segregation Principle
- **DIP:** Dependency Inversion Principle

sub kamon ka separate interface bana do. free parking ka separate or paid ka separate. Means jo kam aik interface ka nahi he wo separate kardo or jese printer scan nahi karta to uska separate interface bana do or scanner ka separate

# ISP-Interface Segregation Principle

- Segregation means keeping things separated, and the Interface Segregation Principle is about **separating the interfaces**.
- The principle states that many client-specific interfaces are better than one general-purpose interface.
- Clients should not be forced to implement a function they do not need.

```
// Violating ISP
class Machine {
public:
    virtual void print() = 0;
    virtual void scan() = 0;
};

// Following ISP
class Printer {
public:
    virtual void print() = 0;
};

class Scanner {
public:
    virtual void scan() = 0;
};
```

```cpp
class Car {
    // Car class definition
};

class ParkingLot {
public:
    virtual void parkCar() = 0;       // Decrease empty spot count by 1
    virtual void unparkCar() = 0;     // Increase empty spots by 1
    virtual void getCapacity() = 0;   // Returns car capacity
    virtual double calculateFee(Car* car) = 0; // Returns price based on number of hours
    virtual void doPayment(Car* car) = 0;
};
```

- We modeled a very simplified parking lot. It is the type of parking lot where you pay an hourly fee.
- Now consider that we want to implement a parking lot that is free.

```cpp
class FreeParking : public ParkingLot {
public:
    void parkCar() override {
        // Implementation for parking a car
    }

    void unparkCar() override {
        // Implementation for unparking a car
    }

    void getCapacity() override {
        // Implementation for getting capacity
    }

    double calculateFee(Car* car) override {
        return 0;
    }

    void doPayment(Car* car) override {
        throw logic_error("Parking lot is free");
    }
};
```

```
int main() {
    Car myCar;
    FreeParking freeLot;
    try {
        freeLot.doPayment(&myCar);
    } catch (const logic_error& e) {
        std::cerr << "Error: " << e.what() << std::endl;
    }
    return 0;
}
```

- Our parking lot interface was composed of 2 things:
  - Parking related logic (park car, unpark car, get capacity)
  - Payment related logic (calculate fee, do payment).
- Because of class being too specific, our FreeParking class was forced to implement payment-related methods that are irrelevant to it.
- Let's separate or segregate the interfaces.

- To address the violation of the Interface Segregation Principle (ISP), we need to split the large ParkingLot interface into smaller, more focused interfaces, so classes that implement them are only required to define the methods they need.
- The FreeParking class doesn't need the calculateFee and doPayment methods since it's a free parking lot.
- To solve this, we can create separate interfaces for operations like parking/unparking, capacity and payment processing.

# How to fix?

```
class Car {
    // Car class definition
};

// Interface for basic parking operations
class ParkingOperations {
public:
    virtual void parkCar() = 0;
    virtual void unparkCar() = 0;
    virtual void getCapacity() = 0;
};

// Interface for payment-related operations
class PaymentProcessing {
public:
    virtual double calculateFee(Car* car) = 0;
    virtual void doPayment(Car* car) = 0;
};
```

# How to fix?

```
// FreeParking class only implements ParkingOperations
class FreeParking : public ParkingOperations {
public:
    void parkCar() override {
        // Implementation for parking a car
    }

    void unparkCar() override {
        // Implementation for unparking a car
    }

    void getCapacity() override {
        // Implementation for getting capacity
    }
};
```

# How to fix?

```cpp
// PaidParking class that needs both parking and payment-related methods
class PaidParking : public ParkingOperations, public PaymentProcessing {
public:
    void parkCar() override {
        // Implementation for parking a car
    }

    void unparkCar() override {
        // Implementation for unparking a car
    }

    void getCapacity() override {
        // Implementation for getting capacity
    }

    double calculateFee(Car* car) override {
        // Implementation for calculating fee
        return 10.0;  // Example fee calculation
    }

    void doPayment(Car* car) override {
        // Implementation for processing payment
        cout << "Payment processed for car." << std::endl;
    }
};
```

# How to fix?

```
int main() {
    Car myCar;

    // Free parking lot does not support payment
    FreeParking freeLot;
    freeLot.parkCar();

    // Paid parking lot with payment processing
    PaidParking paidLot;
    paidLot.parkCar();
    double fee = paidLot.calculateFee(&myCar);
    paidLot.doPayment(&myCar);

    return 0;
}
```

# Example

- Let's assume that there is a Restaurant interface which contains methods for accepting orders from **online customers, telephone customers** and **walk-in customers**.
- It also contains methods for handling online payments (for online customers) and in-person payments.
- In-person payments deal with the walk-in customers as well as telephone customers.
- Telephone customers pay in-person at the time of order delivery.

```
class RestaurantService {
public:
    virtual void acceptOnlineOrder() = 0;
    virtual void acceptPhoneOrder() = 0;
    virtual void acceptWalkInOrder() = 0;
    virtual void processOnlinePayment() = 0;
    virtual void processInPersonPayment() = 0;
};
```

```cpp
class OnlineCustomer : public RestaurantService {
public:
    void acceptOnlineOrder() override {
        cout << "Online order placed successfully." << endl;
    }

    void acceptPhoneOrder() override {
        // Not applicable for online customer
        throw logic_error("Phone order not supported for online customer");
    }

    void acceptWalkInOrder() override {
        // Not applicable for online customer
        throw logic_error("Walk-in order not supported for online customer");
    }

    void processOnlinePayment() override {
        cout << "Online payment processed successfully." << endl;
    }

    void processInPersonPayment() override {
        // Not applicable for online customer
        throw logic_error("In-person payment not supported for online customer");
    }
};
```

```
int main() {
    OnlineCustomer onlineCustomer;
    try {
        onlineCustomer.acceptOnlineOrder();
        onlineCustomer.processOnlinePayment();
        onlineCustomer.acceptPhoneOrder(); // This will throw an exception
    } catch (const logic_error& e) {
        cerr << "Error: " << e.what() << endl;
    }
    return 0;
}
```

- Since OnlineCustomer class is for online customers, we will have to throw Exception for the methods which are not applicable for online customers.
- This is also termed as ***'Interface Pollution'***. Here we can observe clear violation of Interface Segregation Principle.

# How to fix?

```cpp
// OrderInterface for placing orders
class OrderInterface {
public:
    virtual void placeOrder() = 0;
};

// PaymentInterface for processing payments
class PaymentInterface {
public:
    virtual void payForOrder() = 0;
};

// OnlineCustomerImpl implements both OrderInterface and PaymentInterface
class OnlineCustomerImpl : public OrderInterface, public PaymentInterface {
public:
    void placeOrder() override {
        out << "Placing online order..." << endl;
        // logic to place online order
    }

    void payForOrder() override {
        cout << "Processing online payment..." << endl;
        // logic to do online payment
    }
};
```

# How to fix?

```
// WalkInCustomerImpl implements both OrderInterface and PaymentInterface
class WalkInCustomerImpl : public OrderInterface, public PaymentInterface {
public:
    void placeOrder() override {
        cout << "Placing walk-in order..." << endl;
        // logic to place in-person order
    }

    void payForOrder() override {
        cout << "Processing in-person payment..." << endl;
        // logic to do in-person payment
    }
};

int main() {
    // Online customer example
    OnlineCustomerImpl onlineCustomer;
    onlineCustomer.placeOrder();
    onlineCustomer.payForOrder();

    // Walk-in customer example
    WalkInCustomerImpl walkInCustomer;
    walkInCustomer.placeOrder();
    walkInCustomer.payForOrder();

    return 0;
}
```

high level class or low level class ke b/w aik interface bana do or us interface ke sath low level classes ko attach kardo

# DIP-Dependency Inversion Principle

- The Dependency Inversion principle states that our **classes should depend upon interfaces or abstract classes instead of concrete classes and functions**.
- **High-level modules** should not depend on **low-level modules**. Both should depend on **abstractions** (e.g., interfaces).
- We want our classes to be **open to extension**, so we have reorganized our dependencies to depend on interfaces instead of concrete classes.
- This helps in achieving **loosely coupled systems**, making the code more modular, easier to maintain, and flexible for change.

# DIP-Dependency Inversion Principle

- In the context of the **DIP**, classes are often categorized as **high-level** and **low-level** based on their responsibilities and level of abstraction:
- **High-Level Class:** Deals with the **core business logic** or the functionality of a system. It focuses on the **bigger picture** and often relies on lower-level classes to perform specific tasks.
- **Characteristics**:
  - It is more abstract and less concerned with the technical details of how things are done.
  - It often orchestrates and coordinates the flow of operations but leaves specific details to other classes.
  - It represents the **overall behavior** or the **higher-level functionality** of the application.

# DIP-Dependency Inversion Principle

- **Low-Level Class:** Responsible for more **detailed, specific functionality**. It deals with the **implementation details** of a particular task or component of the system.
- **Characteristics**:
  - It is more concrete and focuses on **how something is done** (e.g., handling specific input/output operations).
  - It often performs one specific task and is called upon by high-level classes to fulfill part of the overall functionality.
  - Low-level classes contain the **specific implementations** of operations that the high-level classes rely on.

# Example

- Suppose *ClassA* depends on *ClassB*. At runtime, an instance of *ClassB* will be created or injected into *ClassA*.

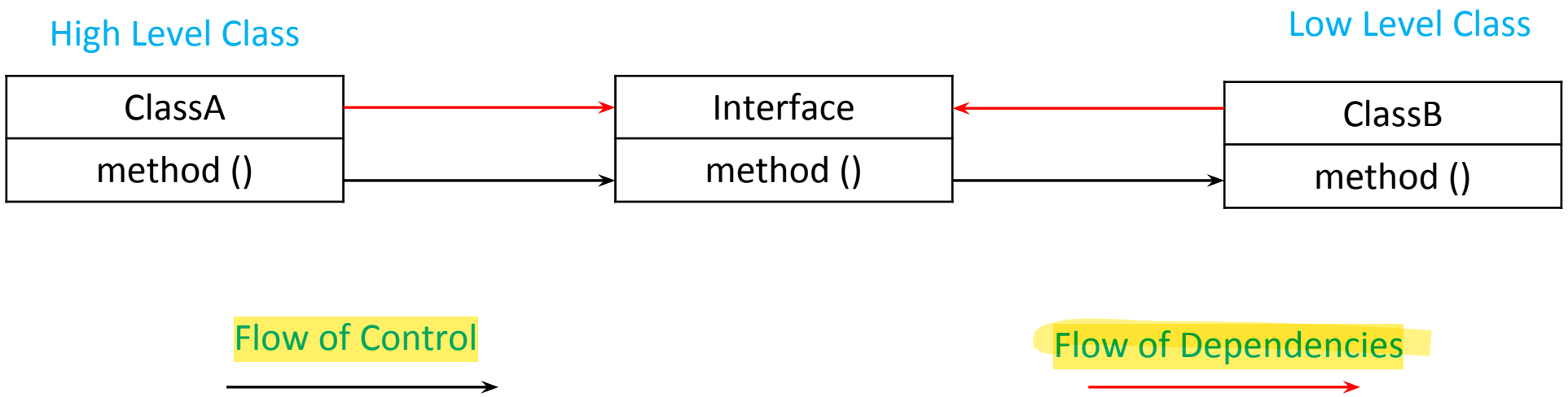
```
class ClassB {
    // fields, constructor and methods
}
class ClassA {
    ClassB objectB;

    ClassA(ClassB objectB) {
        this.objectB = objectB;
    }
    // invoke classB methods
}
```

# Example (Code Violating DIP)

```cpp
// Low-level class
class WiredMouse {
public:
    void connect() {
        cout << "Wired Mouse connected." << endl;
    }
};

// High-level class
class Computer {
    WiredMouse mouse; // Direct dependency on the low-level class

public:
    void start() {
        mouse.connect(); // Directly depends on a specific implementation of the mouse
        cout << "Computer started." << endl;
    }
};

int main() {
    Computer computer;
    computer.start(); // WiredMouse is hard-coded, no flexibility
    return 0;
}
```

# Example

- DIP is telling us is to invert the dependency.
- The flow of control will still follow the same path. However, now both our objects will depend on the abstraction level of the interface.
- Thus, *ClassB* inverts its dependency on *ClassA*.



# Example

```
interface InterfaceB {
    method()
}

class ClassB implements InterfaceB {
    // fields, constructor and methods
}

class ObjectA {
    InterfaceB objectB;

    ObjectA(InterfaceB objectB) {
        this.objectB = objectB;
    }
    ...
}
```

```
// Violating DIP
class MySQLConnection {
public:
    void connect() {}
};

class PasswordReminder {
private:
    MySQLConnection dbConnection; // High-level module depends on low-level module
directly

public:
    PasswordReminder(MySQLConnection conn) : dbConnection(conn) {}
};

// Following DIP
class DatabaseConnection {
public:
    virtual void connect() = 0; // Abstraction
};

class MySQLConnection : public DatabaseConnection {
public:
    void connect() override {}
};

class PasswordReminder {
private:
    DatabaseConnection* dbConnection; // Depends on abstraction

public:
    PasswordReminder(DatabaseConnection* conn) : dbConnection(conn) {}
};
```

# How to fix?

- To follow DIP, we can introduce an abstraction (interface) for the Mouse, and the Computer will depend on this abstraction instead of the concrete class.
- The actual implementation (wired or wireless mouse) can be provided at runtime.

# Example (Code Fix)

```cpp
// Abstract interface for Mouse
class Mouse {
public:
    virtual void connect() = 0;
};

// Low-level class 1: WiredMouse
class WiredMouse : public Mouse {
public:
    void connect() override {
        cout << "Wired Mouse connected." << endl;
    }
};

// Low-level class 2: WirelessMouse
class WirelessMouse : public Mouse {
public:
    void connect() override {
        cout << "Wireless Mouse connected." << endl;
    }
};
```

# Example (Code Fix)

```
// High-level class: Computer, depending on the abstraction (Mouse interface)
class Computer {
    Mouse* mouse; // Dependency is on the abstraction (Mouse)

public:
    // Constructor injection (DIP: Dependency is passed in, not hardcoded)
    Computer(Mouse* m) {
        mouse = m;
    }

    void start() {
        mouse->connect(); // Computer only depends on the Mouse interface
        cout << "Computer started." << endl;
    }
};

int main() {
    WiredMouse wiredMouse;
    WirelessMouse wirelessMouse;

    // Computer can work with any type of mouse
    Computer computer1(&wiredMouse);
    computer1.start();

    Computer computer2(&wirelessMouse);
    computer2.start();

    return 0;
}
```

# Example

- Suppose a bookstore asked us to build a feature that enables customers to put their favorite books on a shelf.

# Example (Book Store)

```cpp
// Book class
class Book {
public:
    void seeReviews() {
        cout << "Seeing book reviews..." << endl;
    }

    void readSample() {
        cout << "Reading book sample..." << endl;
    }
};

// Shelf class that is tightly coupled to Book
class Shelf {
private:
    Book* book;

public:
    void addBook(Book* b) {
        book = b;
        cout << "Book added to shelf." << endl;
    }

    void customizeShelf() {
        cout << "Customizing shelf..." << endl;
    }
};
```

# Example Book Store

- Store asks us to enable customers to add DVDs to their shelves too.

```
// DVD class
class DVD {
public:
    void seeReviews() {
        cout << "Seeing DVD reviews..." << endl;
    }

    void watchSample() {
        cout << "Watching DVD sample..." << endl;
    }
};
```

# Example (Book Store) Fix

```cpp
// Abstract base class (Product interface)
class Product {
public:
    virtual void seeReviews() = 0; // Pure virtual function
    virtual void getSample() = 0;  // Pure virtual function
};

// Book class inheriting from Product
class Book : public Product {
public:
    void seeReviews() override {
        cout << "Seeing book reviews..." << endl;
    }

    void getSample() override {
        cout << "Reading book sample..." << endl;
    }
};
```

# Example (Book Store) Fix

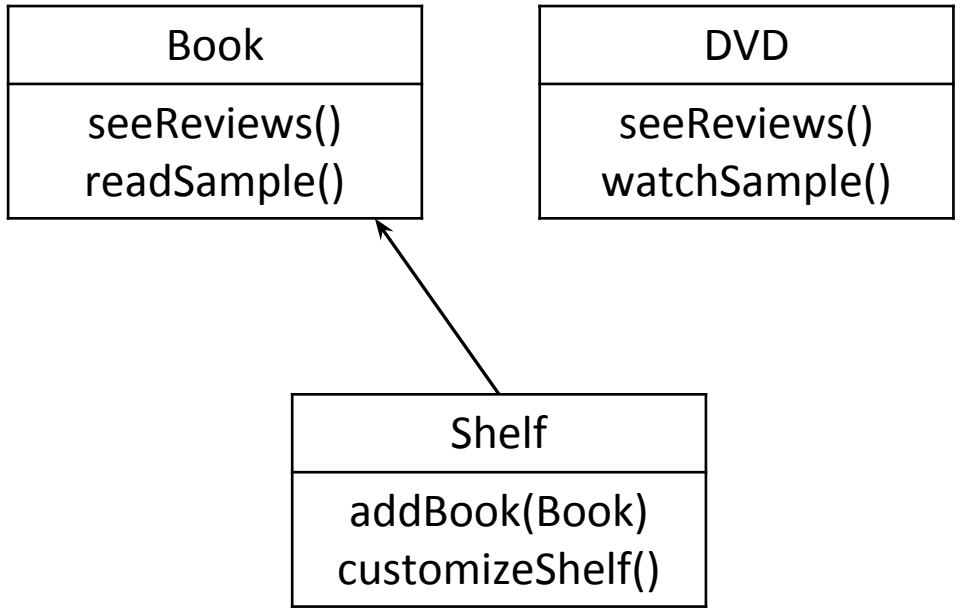
```cpp
// DVD class inheriting from Product
class DVD : public Product {
public:
    void seeReviews() override {
        cout << "Seeing DVD reviews..." << endl;
    }

    void getSample() override {
        cout << "Watching DVD sample..." << endl;
    }
};

// Shelf class to handle products
class Shelf {
private:
    Product* product; // Pointer to the abstract Product

public:
    void addProduct(Product* p) {
        product = p;
        cout << "Product added to shelf." << endl;
    }

    void customizeShelf() {
        cout << "Customizing shelf..." << endl;
    }
};
```

# Example (Book Store) Fix

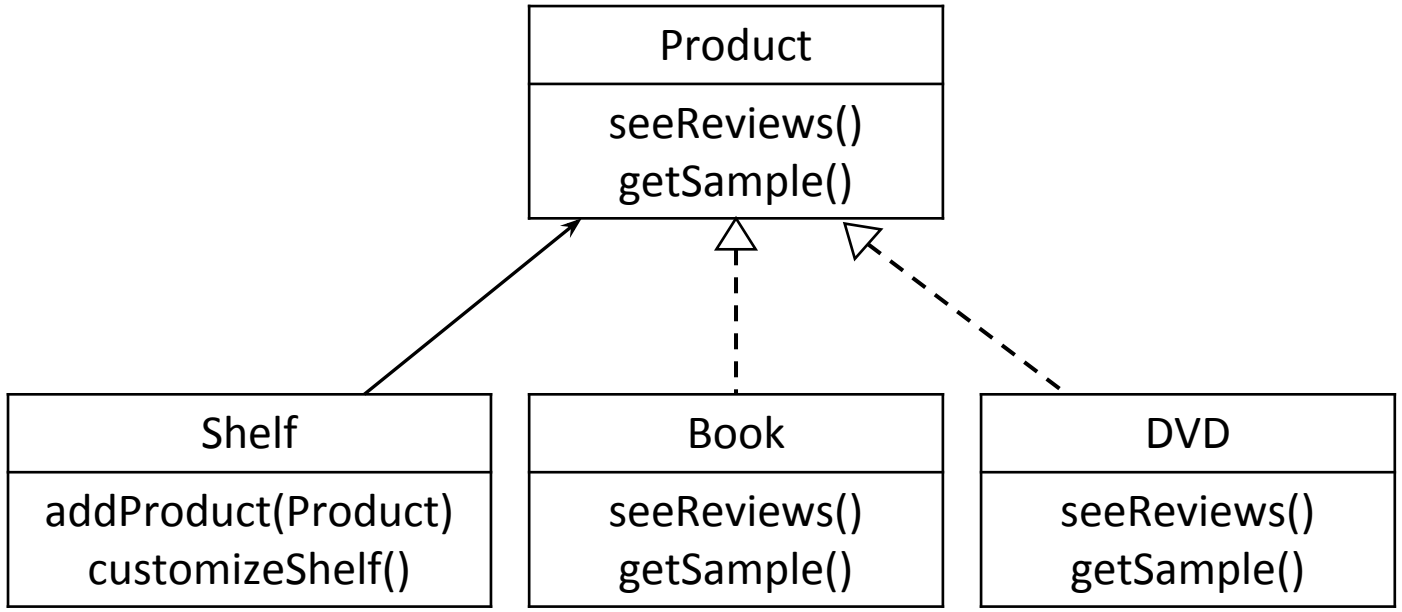
```
int main() {
    // Create book and DVD objects
    Book myBook;
    DVD myDVD;

    // Create shelf
    Shelf myShelf;

    // Add book to shelf
    myShelf.addProduct(&myBook);

    // Add DVD to shelf
    myShelf.addProduct(&myDVD);

    return 0;
}
```

# Example

Before




After



# References


- Clean Architecture: A Craftsman's Guide to Software Structure and Design, 1st Edition, Robert C. Martin, Pearson, 2017.