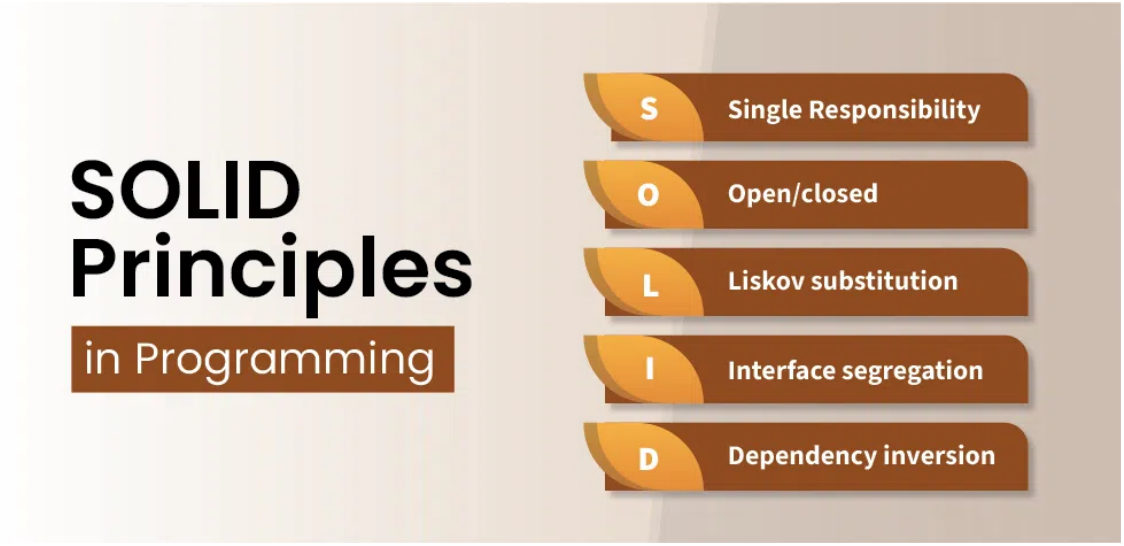
SOLID Principles in Programming: Understand With Real Life Examples

Last Updated : 03 Jan, 2025

The SOLID principles are five essential guidelines that enhance software design, making code more maintainable and scalable. They include Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, and Dependency Inversion. In this article, we'll explore each principle with real-life examples, highlighting their significance in creating robust and adaptable software systems. These five principles are:

1. **Single Responsibility Principle (SRP)**
2. **Open/Closed Principle**
3. **Liskov's Substitution Principle (LSP)**
4. **Interface Segregation Principle (ISP)**
5. **Dependency Inversion Principle (DIP)**



SOLID Principles in Programming

The SOLID principle helps in reducing tight coupling. Tight coupling means a group of classes are highly dependent on one another which you should avoid in your code.

- Opposite of tight coupling is loose coupling and your code is considered as a good code when it has loosely-coupled classes.
- Loosely coupled classes minimize changes in your code, helps in making code more reusable, maintainable, flexible and stable. Now let's discuss one by one these principles...

1. Single Responsibility Principle

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.

Let's understand Single Responsibility Principle using an example:

> *Imagine a baker who is responsible for baking bread. The baker's role is to focus on the task of baking bread, ensuring that the bread is of high quality, properly baked, and meets the bakery's standards.*

- However, if the baker is also responsible for managing the inventory, ordering supplies, serving customers, and cleaning the bakery, this would violate the SRP.
- Each of these tasks represents a separate responsibility, and by combining them, the baker's focus and effectiveness in baking bread could be compromised.
- To adhere to the SRP, the bakery could assign different roles to different individuals or teams. For example, there could be a separate person or team responsible for managing the inventory, another for ordering supplies, another for serving customers, and another for cleaning the bakery.

Code of above Example to understand Single Responsibility Principle:

C++

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

// Class for baking bread
class BreadBaker {
```

```
public:
    void bakeBread() {
        std::cout << "Baking high-quality bread..." << std::endl;
    }
};

// Class for managing inventory
class InventoryManager {
public:
    void manageInventory() {
        std::cout << "Managing inventory..." << std::endl;
    }
};

// Class for ordering supplies
class SupplyOrder {
public:
    void orderSupplies() {
        std::cout << "Ordering supplies..." << std::endl;
    }
};

// Class for serving customers
class CustomerService {
public:
    void serveCustomer() {
        std::cout << "Serving customers..." << std::endl;
    }
};

// Class for cleaning the bakery
class BakeryCleaner {
public:
    void cleanBakery() {
        std::cout << "Cleaning the bakery..." << std::endl;
    }
};

int main() {
    BreadBaker baker;
    InventoryManager inventoryManager;
```

```
    SupplyOrder supplyOrder;
    CustomerService customerService;
    BakeryCleaner cleaner;

    // Each class focuses on its specific responsibility
    baker.bakeBread();
    inventoryManager.manageInventory();
    supplyOrder.orderSupplies();
    customerService.serveCustomer();
    cleaner.cleanBakery();

    return 0;
}
```

In the above example:

- **BreadBaker Class**: Responsible solely for baking bread. This class focuses on ensuring the quality and standards of the bread without being burdened by other tasks.
- **InventoryManager Class**: Handles inventory management, ensuring that the bakery has the right ingredients and supplies available.
- **SupplyOrder Class**: Manages ordering supplies, ensuring that the bakery is stocked with necessary items.
- **CustomerService Class**: Takes care of serving customers, providing a focused approach to customer interactions.
- **BakeryCleaner Class**: Responsible for cleaning the bakery, ensuring a hygienic environment.

2. Open/Closed Principle

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.

Let's understand Open/Closed Principle using an example:

> *Imagine you have a class called `PaymentProcessor` that processes payments for an online store. Initially, the `PaymentProcessor` class only supports processing payments using credit cards. However, you want to extend its functionality to also support processing payments using PayPal.*

Instead of modifying the existing `PaymentProcessor` class to add PayPal support, you can create a new class called `PayPalPaymentProcessor` that extends the `PaymentProcessor` class. This way, the `PaymentProcessor` class remains closed for modification but open for extension, adhering to the Open-Closed Principle. Let's understand this through the code implementation.

Initial Implementation:

C++

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

// Base class for payment processing
class PaymentProcessor {
public:
    virtual void processPayment(double amount) = 0; // Pure virtual
function
};

// Credit card payment processor
class CreditCardPaymentProcessor : public PaymentProcessor {
public:
    void processPayment(double amount) override {
        std::cout << "Processing credit card payment of $" <<
amount << std::endl;
    }
};
```

Extended Functionality:

Now, to add support for PayPal payments, you create a new class PayPalPaymentProcessor that extends PaymentProcessor.

C++

```
// PayPal payment processor
class PayPalPaymentProcessor : public PaymentProcessor {
public:
    void processPayment(double amount) override {
        std::cout << "Processing PayPal payment of $" << amount <<
```

```
std::endl;
    }
};
```

Usage:

In your application, you can use either payment processor without modifying the existing code for PaymentProcessor or CreditCardPaymentProcessor.

C++

```
void processPayment(PaymentProcessor* processor, double amount) {
    processor->processPayment(amount);
}

int main() {
    CreditCardPaymentProcessor creditCardProcessor;
    PayPalPaymentProcessor payPalProcessor;

    processPayment(&creditCardProcessor, 100.00); // Processing
credit card payment
    processPayment(&payPalProcessor, 150.00);     // Processing
PayPal payment

    return 0;
}
```

Explanation of the above code:

- **Base Class (`PaymentProcessor`)**: This is an abstract base class with a pure virtual function `processPayment()`. It defines a common interface for all payment processors.
- **CreditCardPaymentProcessor**: This class implements the payment processing logic for credit card payments.
- **PayPalPaymentProcessor**: This new class extends the functionality by implementing the payment processing for PayPal payments.
- **Main Function**: The `processPayment` function takes a pointer to a `PaymentProcessor` and calls the `processPayment()` method. This allows you to use any processor that implements the `PaymentProcessor` interface without changing existing code.

3. Liskov's Substitution Principle

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.

Let's understand Liskov's Substitution Principle using an example:

> *One of the classic examples of this principle is a rectangle having four sides. A rectangle's height can be any value and width can be any value. A square is a rectangle with equal width and height. So we can say that we can extend the properties of the rectangle class into square class.*

In order to do that you need to swap the child (square) class with parent (rectangle) class to fit the definition of a square having four equal sides but a derived class does not affect the behavior of the parent class so if you will do that it will violate the Liskov Substitution Principle.

Code of above example to completely understand Liskov's Substitution Principle:

C++

```
#include <iostream>

// Base class for shapes
class Rectangle {
protected:
    double width;
    double height;

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

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

    double getWidth() const {
        return width;
    }
```

```
    double getHeight() const {
        return height;
    }

    void setWidth(double w) {
        width = w;
    }

    void setHeight(double h) {
        height = h;
    }
};

// Derived class for squares
class Square : public Rectangle {
public:
    Square(double size) : Rectangle(size, size) {}

    void setWidth(double w) override {
        width = height = w; // Ensure both width and height remain
the same
    }
```

Explanation of the above code:

- **Rectangle Class**: This is the base class that has properties for width and height. It has methods for calculating the area and for setting width and height.
- **Square Class**: This class inherits from `Rectangle` but overrides the `setWidth` and `setHeight` methods to ensure that changing one dimension affects the other, maintaining the property that all sides are equal.

LSP Violation Example:

- To see a potential violation of LSP, consider what would happen if you were to use the `Square` class in a context expecting a `Rectangle`:
- If you substitute a `Square` where a `Rectangle` is expected, changing just the

width or height would lead to unexpected results because it will change both dimensions.

4. Interface Segregation Principle

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**". Here your main goal is to focus on avoiding fat interface and give preference to many small client-specific interfaces. You should prefer many client interfaces rather than one general interface and each interface should have a specific responsibility.

Let's understand Interface Segregation Principle using an example:

> *Suppose if you enter a restaurant and you are pure vegetarian. The waiter in that restaurant gave you the menu card which includes vegetarian items, non-vegetarian items, drinks, and sweets.*

- In this case, as a customer, you should have a menu card which includes only vegetarian items, not everything which you don't eat in your food. Here the menu should be different for different types of customers.
- The common or general menu card for everyone can be divided into multiple cards instead of just one. Using this principle helps in reducing the side effects and frequency of required changes.

Code of above example to understand Interface Segregation Principle:

C++

```
#include <iostream>
#include <vector>
#include <string>

// Interface for vegetarian menu
class IVegetarianMenu {
public:
    virtual std::vector<std::string> getVegetarianItems() = 0;
};
```

```
// Interface for non-vegetarian menu
class INonVegetarianMenu {
public:
    virtual std::vector<std::string> getNonVegetarianItems() = 0;
};

// Interface for drinks menu
class IDrinkMenu {
public:
    virtual std::vector<std::string> getDrinkItems() = 0;
};

// Class for vegetarian menu
class VegetarianMenu : public IVegetarianMenu {
public:
    std::vector<std::string> getVegetarianItems() override {
        return {"Vegetable Curry", "Paneer Tikka", "Salad"};
    }
};

// Class for non-vegetarian menu
class NonVegetarianMenu : public INonVegetarianMenu {
public:
    std::vector<std::string> getNonVegetarianItems() override {
        return {"Chicken Curry", "Fish Fry", "Mutton Biryani"};
    }
};

// Class for drinks menu
class DrinkMenu : public IDrinkMenu {
public:
    std::vector<std::string> getDrinkItems() override {
        return {"Water", "Soda", "Juice"};
    }
};

// Function to display menu items for a vegetarian customer
void displayVegetarianMenu(IVegetarianMenu* menu) {
    std::cout << "Vegetarian Menu:\n";
    for (const auto& item : menu->getVegetarianItems()) {
        std::cout << "- " << item << std::endl;
```

```
    }
}

// Function to display menu items for a non-vegetarian customer
void displayNonVegetarianMenu(INonVegetarianMenu* menu) {
    std::cout << "Non-Vegetarian Menu:\n";
    for (const auto& item : menu->getNonVegetarianItems()) {
        std::cout << "- " << item << std::endl;
    }
}

int main() {
    VegetarianMenu vegMenu;
    NonVegetarianMenu nonVegMenu;
    DrinkMenu drinkMenu;

    displayVegetarianMenu(&vegMenu);
    displayNonVegetarianMenu(&nonVegMenu);

    return 0;
}
```

Explanation of the above code:

- **IVegetarianMenu Interface**: This interface defines a method to get vegetarian items. It ensures that only classes implementing vegetarian menus will need to provide this functionality.
- **INonVegetarianMenu Interface**: Similar to the vegetarian interface, this one defines a method for getting non-vegetarian items.
- **IDrinkMenu Interface**: This interface defines a method for getting drink items, keeping it separate from food items.
- **VegetarianMenu Class**: Implements the `IVegetarianMenu` interface and provides a list of vegetarian items.
- **NonVegetarianMenu Class**: Implements the `INonVegetarianMenu` interface and provides a list of non-vegetarian items.
- **DrinkMenu Class**: Implements the `IDrinkMenu` interface and provides a list of drink items.

5. Dependency Inversion Principle

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.

- In simpler terms, the DIP suggests that classes should rely on abstractions (e.g., interfaces or abstract classes) rather than concrete implementations.
- This allows for more flexible and decoupled code, making it easier to change implementations without affecting other parts of the codebase.

Let's understand Dependency Inversion Principle using an example:

> *In a software development team, developers depend on an abstract version control system (e.g., Git) to manage and track changes to the codebase. They don't depend on specific details of how Git works internally.*

This allows developers to focus on writing code without needing to understand the intricacies of version control implementation. Below is the code of above example to understand Dependency Inversion Principle:

C++

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

// Interface for version control system
class IVersionControl {
public:
    virtual void commit(const std::string& message) = 0;
    virtual void push() = 0;
    virtual void pull() = 0;
};

// Git version control implementation
class GitVersionControl : public IVersionControl {
public:
    void commit(const std::string& message) override {
        std::cout << "Committing changes to Git with message: " <<
message << std::endl;
```

```
    }

    void push() override {
        std::cout << "Pushing changes to remote Git repository." <<
std::endl;
    }

    void pull() override {
        std::cout << "Pulling changes from remote Git repository."
<< std::endl;
    }
};

// Team class that relies on version control
class DevelopmentTeam {
private:
    IVersionControl* versionControl;

public:
    DevelopmentTeam(IVersionControl* vc) : versionControl(vc) {}

    void makeCommit(const std::string& message) {
        versionControl->commit(message);
    }

    void performPush() {
        versionControl->push();
    }

    void performPull() {
        versionControl->pull();
    }
};

int main() {
    GitVersionControl git;
    DevelopmentTeam team(&git);

    team.makeCommit("Initial commit");
    team.performPush();
    team.performPull();
```

```
    return 0;
}
```

Explanation of the above code:

- **IVersionControl Interface**: This defines the operations that any version control system should support, like `commit`, `push`, and `pull`. It serves as an abstraction that decouples high-level code from low-level implementations.
- **GitVersionControl Class**: This class implements the `IVersionControl` interface, providing specific functionality for managing version control using Git.
- **DevelopmentTeam Class**: This class relies on the `IVersionControl` interface, meaning it can work with any version control implementation that adheres to the interface. It does not need to know the details of how Git works internally.

Need for SOLID Principles in Object-Oriented Design

Below are some of the main reasons why solid principles are important in object oriented design:

- SOLID principles make code easier to maintain. When each class has a clear responsibility, it's simpler to find where to make changes without affecting unrelated parts of the code.
- These principles support growth in software. For example, the Open/Closed Principle allows developers to add new features without changing existing code, making it easier to adapt to new requirements.
- SOLID encourages flexibility. By depending on abstractions rather than specific implementations (as in the Dependency Inversion Principle), developers can change components without disrupting the entire system.

Next Article

Creational Design Patterns