SOLID Principles Cheat Sheet

A concise guide to the SOLID principles of object-oriented design, aimed at creating maintainable, flexible, and robust software.

SOLID Principles Overview

Introduction to SOLID

SOLID is an acronym representing five key principles of object-oriented design. These principles aim to reduce dependencies, increase code reusability, and improve overall software maintainability.

Adhering to SOLID principles leads to code that is easier to understand, test, and modify, reducing the likelihood of introducing bugs during development or maintenance.

The SOLID Acronym

S	Single Responsibility Principle
O	Open/Closed Principle
L	Liskov Substitution Principle
I	Interface Segregation Principle
D	Dependency Inversion Principle

Single Responsibility Principle (SRP)

SRP Definition

A class should have one, and only one, reason to change. In other words, a class should have only one job or responsibility.

This principle aims to avoid creating 'God Classes' that handle too many unrelated tasks, making them difficult to maintain and understand.

SRP Example

Consider a class that handles both user authentication and logging. According to SRP, these should be separated into distinct classes (e.g., `Authenticator` and `Logger`).

```
// Before SRP
class UserManagement {
    public void authenticateUser(String username, String password) { /*
... */ }
    public void logActivity(String activity) { /* ... */ }
}

// After SRP
class Authenticator {
    public void authenticateUser(String username, String password) { /*
... */ }
}

class Logger {
    public void logActivity(String activity) { /* ... */ }
}
```

Open/Closed Principle (OCP)

OCP Definition

Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.

This means you should be able to add new functionality without changing existing code. Achieved through abstraction and polymorphism.

OCP Example

Instead of modifying a class to support new payment methods, create an abstract `PaymentMethod` class with concrete subclasses for each method (e.g., `CreditCardPayment`, `PayPalPayment`).

```
// Before OCP
class PaymentProcessor {
    public void processPayment(String method, double amount) {
        if (method.equals("credit_card")) { /* ... */ }
        else if (method.equals("paypal")) { /* ... */ }
    }
}

// After OCP
interface PaymentMethod {
    void processPayment(double amount);
}

class CreditCardPayment implements PaymentMethod {
    public void processPayment(double amount) { /* ... */ }
}

class PayPalPayment implements PaymentMethod {
    public void processPayment(double amount) { /* ... */ }
}
```

Liskov Substitution Principle (LSP)

LSP Definition

Subtypes must be substitutable for their base types without altering the correctness of the program.

In simpler terms, if you have a base class and a derived class, you should be able to use the derived class wherever the base class is expected without causing unexpected behavior.

LSP Example

A classic violation is the 'square/rectangle' problem. If `Square` inherits from `Rectangle`, and `setWidth` and `setHeight` are defined in `Rectangle`, `Square` cannot maintain the invariant that width always equals height when setting width/height independently.

```
// LSP Violation
class Rectangle {
    protected int width, height;

    public void setWidth(int width) { this.width = width; }
    public void setHeight(int height) { this.height = height; }

    public int getArea() { return width * height; }
}

class Square extends Rectangle {
    @Override
    public void setWidth(int width) {
        super.setWidth(width);
        super.setHeight(width); // Breaks LSP
    }

    @Override
    public void setHeight(int height) {
        super.setWidth(height);
        super.setHeight(height); // Breaks LSP
    }
}
```

Interface Segregation Principle (ISP)

ISP Definition

Clients should not be forced to depend upon interfaces that they do not use.

Instead of creating large, monolithic interfaces, it's better to split them into smaller, more specific interfaces, so that clients only need to implement the methods they actually use.

ISP Example

Instead of having one `Worker` interface with methods like `work()`, `eat()`, and `sleep()`, create separate interfaces like `IWorkable`, `IEatable`, and `ISleepable`. This prevents classes that only need to work from being forced to implement eat and sleep methods.

```
// Before ISP
interface Worker {
    void work();
    void eat();
    void sleep();
}

class Human implements Worker {
    public void work() { /* ... */ }
    public void eat() { /* ... */ }
    public void sleep() { /* ... */ }
}

class Robot implements Worker { //ISP Violation
    public void work() { /* ... */ }
    public void eat() { /* Not applicable */ }
    public void sleep() { /* Not applicable */ }
}

// After ISP
interface IWorkable { void work(); }
interface IEatable { void eat(); }
interface ISleepable { void sleep(); }

class Human implements IWorkable, IEatable, ISleepable {
    public void work() { /* ... */ }
    public void eat() { /* ... */ }
    public void sleep() { /* ... */ }
}

class Robot implements IWorkable {
    public void work() { /* ... */ }
}
```

Dependency Inversion Principle (DIP)

DIP Definition

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.

This principle promotes loose coupling by introducing abstraction layers between high-level policies and low-level implementation details.

DIP Example

Instead of a high-level module directly using a low-level module, both should depend on an interface. For example, a `PasswordReminder` class shouldn't depend directly on a `MySQLConnection` class, but rather on a `DatabaseConnection` interface.

```
// Before DIP
class PasswordReminder {
    private MySQLConnection dbConnection;

    public PasswordReminder(MySQLConnection dbConnection) {
        this.dbConnection = dbConnection;
    }
}

class MySQLConnection {
    public void connect() { /* ... */ }
}

// After DIP
interface DBConnection {
    void connect();
}

class MySQLConnection implements DBConnection {
    public void connect() { /* ... */ }
}

class PasswordReminder {
    private DBConnection dbConnection;

    public PasswordReminder(DBConnection dbConnection) {
        this.dbConnection = dbConnection;
    }
}
```