Principles of Software Development

SOLID

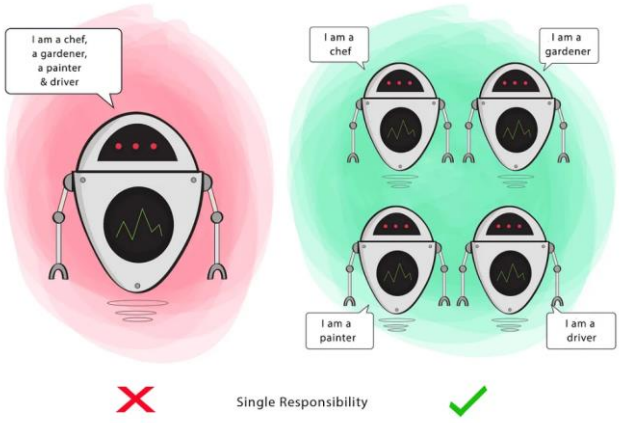
They promote the design of robust and scalable code. It was introduced by Robert C. Martin (Uncle Bob) in the early 2000s. The five SOLID principles and their respective advantages:

S - Single Responsibility Principle (SRP)

The Single Responsibility Principle (SRP) states that a class should have only one well-defined responsibility OR A class should have one, and only one, reason to change. In other words, a class should be responsible for only one task or one aspect of the system. This facilitates code understanding, maintenance, and reusability.

Goal: This principle aims to separate behaviours so that if bugs arise as a result of your change, it won't affect other unrelated behaviours.

The benefits of applying SRP are numerous. First, it makes the code more modular, making it easier to make modifications and additions later on. Additionally, troubleshooting and issue resolution are simplified as each class focuses on a single responsibility. Finally, code reusability is promoted, as specialized classes can be used in different parts of the system.



Code Example: Report Generation

1. Without SRP (Bad Design)

```
class Report {
    private String content;

    public Report(String content) {
        this.content = content;
    }

    public String getContent() {
        return content;
    }

    // This method mixes formatting responsibility
    public void printReport() {
        System.out.println("Report Content: " + content);
    }
}

public class Main {
    public static void main(String[] args) {
        Report report = new Report("This is the report content.");
        report.printReport(); // Prints the report
    }
}
```

Problem:

- The Report class has two responsibilities:
    1. Managing the report data.
    2. Printing the report.
- Any change in formatting or printing would require modifying the Report class.

2. With SRP (Good Design)

```
class Report {
    private String content;

    public Report(String content) {
        this.content = content;
    }
```

```
        public String getContent() {
            return content;
        }
    }

    class ReportPrinter {
        public void printReport(Report report) {
            System.out.println("Report Content: " + report.getContent());
        }
    }

    public class Main {
        public static void main(String[] args) {
            Report report = new Report("This is the report content.");
            ReportPrinter printer = new ReportPrinter();
            printer.printReport(report); // Prints the report
        }
}
```

Improvements:

1. Separate Responsibilities:
    - The Report class handles only report data.
    - The ReportPrinter class handles printing.
2. Easier Maintenance:
    - Changes to the printing logic (e.g., formatting) are confined to ReportPrinter, leaving Report unaffected.

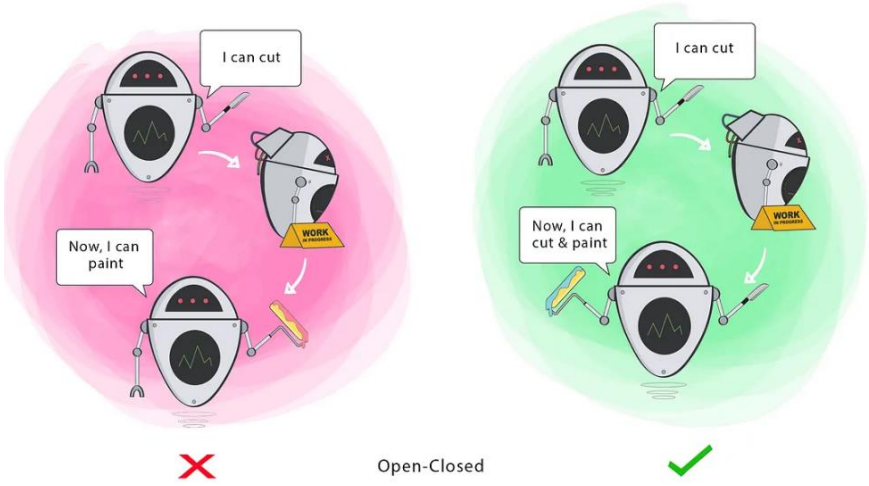
O — Open/Closed Principle (OCP)

Classes (modules, functions, etc.) should be open for extension, but closed for modification. Changing the current behaviour of a Class will affect all the systems using that Class.

If you want the Class to perform more functions, the ideal approach is to add to the functions that already exist NOT change them.

Goal: This principle aims to extend a Class's behaviour without changing the existing behaviour of that Class. This is to avoid causing bugs wherever the Class is being used.

The key advantage of applying OCP lies in its ability to make the code more flexible and extensible. By using mechanisms such as inheritance, polymorphism, and inversion of control, we can add new features without impacting the existing code. It also facilitates unit testing, as existing features are not altered when introducing new ones.



Code Example: Payment Processing System

1. Without OCP (Bad Design)

```
class PaymentProcessor {
    public void processPayment(String paymentType) {
        if (paymentType.equals("CreditCard")) {
            System.out.println("Processing credit card payment...");
        } else if (paymentType.equals("PayPal")) {
            System.out.println("Processing PayPal payment...");
        }
        // Adding new payment types requires modifying this method
    }
}

public class Main {
    public static void main(String[] args) {
        PaymentProcessor processor = new PaymentProcessor();
        processor.processPayment("CreditCard");
    }
}
```

Problem:

- Adding a new payment type (e.g., "Bitcoin") requires modifying the processPayment method, violating OCP.
- The class grows in complexity as new payment methods are added.

2. With OCP (Good Design):

```
interface Payment {
    void process();
}

class CreditCardPayment implements Payment {
    @Override
    public void process() {
        System.out.println("Processing credit card payment...");
    }
}

class PayPalPayment implements Payment {
    @Override
    public void process() {
        System.out.println("Processing PayPal payment...");
    }
}

// Adding a new payment method
class BitcoinPayment implements Payment {
    @Override
    public void process() {
        System.out.println("Processing Bitcoin payment...");
    }
}

class PaymentProcessor {
    public void processPayment(Payment payment) {
        payment.process();
    }
}

public class Main {
    public static void main(String[] args) {
        PaymentProcessor processor = new PaymentProcessor();

        Payment creditCardPayment = new CreditCardPayment();
        Payment payPalPayment = new PayPalPayment();
        Payment bitcoinPayment = new BitcoinPayment(); // New payment
method

        processor.processPayment(creditCardPayment);
        processor.processPayment(payPalPayment);
```

```
        processor.processPayment(bitcoinPayment);
    }
}
```

Improvements:

1. Extensibility:
    - To add a new payment type, simply create a new class that implements the Payment interface.
    - No changes are required to the PaymentProcessor class.
2. Closed for Modification:
    - The PaymentProcessor class remains untouched even when new payment types are introduced.

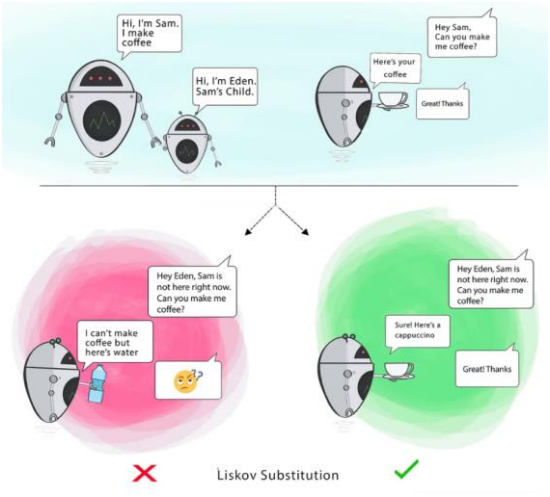
L - Liskov Substitution Principle (LSP)

If S is a subtype of T, then objects of type T in a program may be replaced with objects of type S without altering any of the desirable properties of that program.

When a child Class cannot perform the same actions as its parent Class, this can cause bugs. The child Class should be able to process the same requests and deliver the same result as the parent Class or it could deliver a result that is of the same type. If the child Class doesn't meet these requirements, it means the child Class is changed completely and violates this principle.

Goal: This principle aims to enforce consistency so that the parent Class or its child Class can be used in the same way without any errors.

The main advantage of applying LSP this promotes modularity and code reusability, as new subclasses can be added without disrupting existing parts of the system.



Code Example: Shape Area Calculation

1. Bad Design: Violating LSP

```
class Rectangle {
    protected int width;
    protected int height;

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

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

    public int getWidth() {
        return width;
    }

    public int getHeight() {
        return height;
    }

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

class Square extends Rectangle {
    @Override
    public void setWidth(int width) {
        this.width = width;
        this.height = width; // Ensuring both sides are equal
    }

    @Override
    public void setHeight(int height) {
        this.height = height;
        this.width = height; // Ensuring both sides are equal
    }
}

public class Main {
    public static void main(String[] args) {
        Rectangle rectangle = new Rectangle();
```

```
        rectangle.setWidth(5);
        rectangle.setHeight(10);
        System.out.println("Rectangle area: " + rectangle.getArea()); // Output: 50

        Rectangle square = new Square(); // Using Square as a Rectangle
        square.setWidth(5);
        square.setHeight(10);
        System.out.println("Square area: " + square.getArea()); // Output: 100
(Violates expectations)
    }
}
```

Problem:

- Substituting a Square object for a Rectangle breaks the program's logic.
- A Rectangle expects width and height to be independent, but the Square changes this behavior, violating LSP.

2. Good Design: Complying with LSP

```
interface Shape {
    int getArea();
}

class Rectangle implements Shape {
    protected int width;
    protected int height;

    public Rectangle(int width, int height) {
        this.width = width;
        this.height = height;
    }

    public int getWidth() {
        return width;
    }

    public int getHeight() {
        return height;
    }

    @Override
    public int getArea() {
```

```
        return width * height;
    }
}

class Square implements Shape {
    private int side;

    public Square(int side) {
        this.side = side;
    }

    public int getSide() {
        return side;
    }

    @Override
    public int getArea() {
        return side * side;
    }
}

public class Main {
    public static void main(String[] args) {
        Shape rectangle = new Rectangle(5, 10);
        System.out.println("Rectangle area: " + rectangle.getArea()); //
Output: 50

        Shape square = new Square(5);
        System.out.println("Square area: " + square.getArea()); //
Output: 25
    }
}
```

Improvements:

1. Separate Responsibilities:
    - Rectangle and Square now implement the Shape interface instead of inheriting from one another.
2. No Unintended Behavior:
    - Square doesn't override or modify Rectangle's behavior, ensuring substitution doesn't break the program.
3. LSP Compliance:

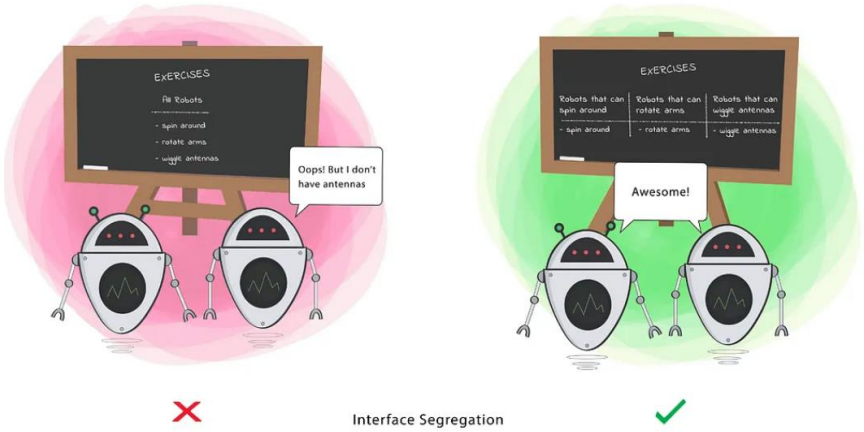
- Both Rectangle and Square can be used interchangeably as Shape without altering the program's correctness.

I - Interface Segregation Principle (ISP)

The Interface Segregation Principle (ISP) advocates for defining specific interfaces for clients rather than having a monolithic interface. In other words, clients should not be forced to implement methods they don't use.

By respecting ISP, we create more concise interfaces tailored to the specific needs of clients, making our code more flexible and extensible.When a Class is required to perform actions that are not useful, it is wasteful and may produce unexpected bugs if the Class does not have the ability to perform those actions.

Goal: This principle aims at splitting a set of actions into smaller sets so that a Class executes ONLY the set of actions it requires.



Code Example: ISP in Action

Scenario: Printing and Scanning Devices

1. Without ISP (Bad Design):

```
interface Machine {
    void print(String document);
    void scan(String document);
    void fax(String document);
}

class AllInOnePrinter implements Machine {
    @Override
    public void print(String document) {
```

```
        System.out.println("Printing: " + document);
    }

    @Override
    public void scan(String document) {
        System.out.println("Scanning: " + document);
    }

    @Override
    public void fax(String document) {
        System.out.println("Faxing: " + document);
    }
}

class BasicPrinter implements Machine {
    @Override
    public void print(String document) {
        System.out.println("Printing: " + document);
    }

    @Override
    public void scan(String document) {
        throw new UnsupportedOperationException("Scan not supported!");
    }

    @Override
    public void fax(String document) {
        throw new UnsupportedOperationException("Fax not supported!");
    }
}

public class Main {
    public static void main(String[] args) {
        Machine printer = new BasicPrinter();
        printer.print("Document 1");
        printer.scan("Document 1"); // Throws exception
    }
}
```

Problem:

- The BasicPrinter class is forced to implement methods (scan, fax) it doesn't need.

- Results in methods throwing UnsupportedOperationException, which violates ISP.

2. With ISP (Good Design):

```
interface Printer {
    void print(String document);
}

interface Scanner {
    void scan(String document);
}

interface Fax {
    void fax(String document);
}

class AllInOnePrinter implements Printer, Scanner, Fax {
    @Override
    public void print(String document) {
        System.out.println("Printing: " + document);
    }

    @Override
    public void scan(String document) {
        System.out.println("Scanning: " + document);
    }

    @Override
    public void fax(String document) {
        System.out.println("Faxing: " + document);
    }
}

class BasicPrinter implements Printer {
    @Override
    public void print(String document) {
        System.out.println("Printing: " + document);
    }
}

public class Main {
    public static void main(String[] args) {
        Printer printer = new BasicPrinter();
```

```
        printer.print("Document 1"); // Works fine
    }
}
```

Improvements with ISP:

1. Specific Interfaces:
    - Instead of a large Machine interface, we now have Printer, Scanner, and Fax interfaces, each focused on a specific responsibility.
2. Flexibility:
    - Classes only implement interfaces they need. For example, BasicPrinter implements only Printer, avoiding unnecessary or unsupported methods.
3. No Useless Methods:
    - No more UnsupportedOperationException. Each class uses only the functionality it actually supports.

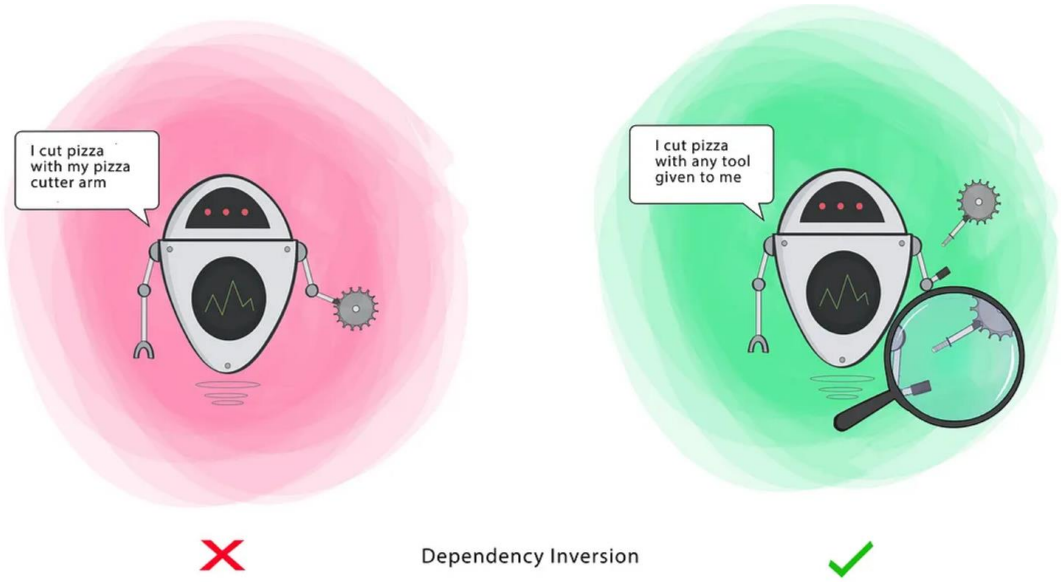
D — Dependency Inversion Principle

The Dependency Inversion Principle (DIP) encourages the use of abstract dependencies rather than relying on concrete classes. In other words, high-level modules should not depend directly on low-level modules but on common abstractions OR No client should be forced to depend on interfaces they don't use.

This principle says a Class should not be fused with the tool it uses to execute an action. Rather, it should be fused to the interface that will allow the tool to connect to the Class.

It also says that both the Class and the interface should not know how the tool works. However, the tool needs to meet the specification of the interface.

Goal: This principle aims at reducing the dependency of a high-level Class on the low-level Class by introducing an interface.



Code Example: Notification Service

1. Without DIP (Bad Design):

```
class EmailService {
    public void sendEmail(String message) {
        System.out.println("Email sent: " + message);
    }
}

class Notification {
    private EmailService emailService;

    public Notification() {
        this.emailService = new EmailService(); // Tight coupling
    }

    public void send(String message) {
        emailService.sendEmail(message);
    }
}

public class Main {
    public static void main(String[] args) {
        Notification notification = new Notification();
        notification.send("Hello, Dependency Inversion!");
    }
}
```

Problem:

- Notification depends directly on the EmailService class, creating a tight coupling.
- Changing the notification method (e.g., adding SMS or push notifications) requires modifying the Notification class.

2. With DIP (Good Design):

```
interface MessageService {
    void sendMessage(String message);
}

class EmailService implements MessageService {
    @Override
    public void sendMessage(String message) {
        System.out.println("Email sent: " + message);
    }
}

class SMSService implements MessageService {
    @Override
    public void sendMessage(String message) {
        System.out.println("SMS sent: " + message);
    }
}

class Notification {
    private MessageService messageService;

    // Depend on abstraction (interface)
    public Notification(MessageService messageService) {
        this.messageService = messageService;
    }

    public void send(String message) {
        messageService.sendMessage(message);
    }
}

public class Main {
    public static void main(String[] args) {
        MessageService emailService = new EmailService();
        MessageService smsService = new SMSService();
```

```
        Notification emailNotification = new Notification(emailService);
        emailNotification.send("Hello via Email!");

        Notification smsNotification = new Notification(smsService);
        smsNotification.send("Hello via SMS!");
    }
}
```

Improvements with DIP:

- Abstraction: Notification depends on the MessageService interface, not concrete implementations.
- Flexibility: You can easily add new message services (e.g., PushNotificationService) without changing the Notification class.
- Loose Coupling: High-level and low-level modules depend on abstractions, making the system more maintainable and testable.

DRY (Don't Repeat Yourself)

The DRY (Don't Repeat Yourself) principle emphasizes the elimination of unnecessary code duplication in a software development project. According to this principle, each piece of knowledge or logic should have a single canonical representation within the system.

Let's explore the benefits offered by the DRY principle.

Reduction of Complexity:

- It reduces code complexity by avoiding unnecessary repetitions.
- It simplifies code maintenance, as modifications and fixes only need to be made in one place rather than in multiple parts of the code.
- It promotes code reuse, as common functionalities or logics can be encapsulated into functions, classes, or modules that can be used in multiple places within the system.

Elimination of Duplicate Code:

- extracting functions or methods allows grouping similar and repetitive code blocks into a reusable function. This way, the same code can be called in multiple places without needing to rewrite it.

Grouping by Functionality:

- the use of classes and inheritance can help encapsulate common functionalities and reuse them in specific subclasses. This way, common functionalities can be defined once in a parent class and inherited in child classes.

Code Reusability

- Finally, the use of libraries, modules, or frameworks can aid in reusing code that has already been written and tested by other developers, avoiding the need to reinvent the wheel.

KISS (Keep It Simple, Stupid)

According to this principle, it's better to maintain simple solutions rather than making them complex. Simplicity promotes understanding, maintenance, and problem-solving.

Applying the KISS principle brings several advantages:

- Better Code Understanding: It facilitates code understanding for developers as simple solutions are clearer and more intuitive.
- Reduced Errors: It also reduces the risk of errors and bugs since simple solutions are easier to test and verify.
- More Scalable Code: Simplicity makes code more flexible and scalable as it's easier to make modifications or add new features to simple code rather than complex code.

Other Important Principles:

YAGNI (You Ain't Gonna Need It)

The YAGNI (You Ain't Gonna Need It) principle emphasizes not implementing features or code that are not immediately necessary. According to this principle, it's better to focus on essential features and avoid anticipating hypothetical future needs.

Advantages:

- It avoid over-engineering, reduces code complexity by avoiding the addition of unnecessary features.
- This makes the code clearer, lighter, and easier to maintain.
- It saves time and resources by avoiding the development and testing of features that might never be used.
- It promotes an iterative approach to development by focusing on the immediate needs of users and allowing the addition of additional features as they become genuinely necessary.

Let's take a concrete example to illustrate the application of the YAGNI principle.

Suppose we are developing a task management application. Instead of implementing an advanced scheduling feature with customizable reminders right from the start, we could begin with a basic functionality of task creation and tracking. By focusing on essential features, we can quickly deliver an initial version of the application, gather user feedback, and iterate by adding additional features like advanced scheduling if it proves to be a genuine user demand.

Convention over Configuration (CoC)

The Convention over Configuration (CoC) principle promotes the use of predetermined conventions rather than explicit configurations. By following these conventions, developers can reduce the amount of necessary configuration and automatically benefit from functionality, simplifying the development process and improving code readability.

This principle is widely applied in many tools and frameworks, and developers often benefit from it without even realizing it.

For example, the structure of a Java project with directories like src/main/java, src/main/resources, and src/test/java follows the CoC principle. By placing test files in the src/test/java directory, the tests are automatically executed when running the tests. Similarly, the “Test” suffix in JUnit file names also follows the Convention over Configuration principle.

Applying the CoC principle also facilitates collaboration among team members as they share a common understanding of conventions and can focus on business logic rather than configuration details.

Law of Demeter (LoD)

The Law of Demeter (LoD), also known as the “Principle of Only Talking to Your Closest Friends,” is a software design principle that promotes decoupling and reducing dependencies between classes. According to this principle, a class should only interact with its immediate collaborators and not directly access members of objects it interacts with indirectly.

Advantages:

- It promotes decoupling between classes, making the code more modular, flexible, and easier to maintain.
- When a class depends only on its immediate collaborators, it becomes less sensitive to internal changes in the objects it interacts with indirectly. This helps reduce the risks of unintended side effects and facilitates the localization and correction of errors.

Suppose we have a “Client” class that interacts with a “Bank” class to perform financial transactions. Instead of directly accessing members of the “Bank” class such

as bank accounts, the “Client” class can use methods provided by the “Bank” class that supply the necessary information. This way, the “Client” class depends only on the interface provided by the “Bank” class and doesn’t need to know the internal details of that class.

Resource: [Principles of Software Development: SOLID, DRY, KISS, and more | Scalastic](#)