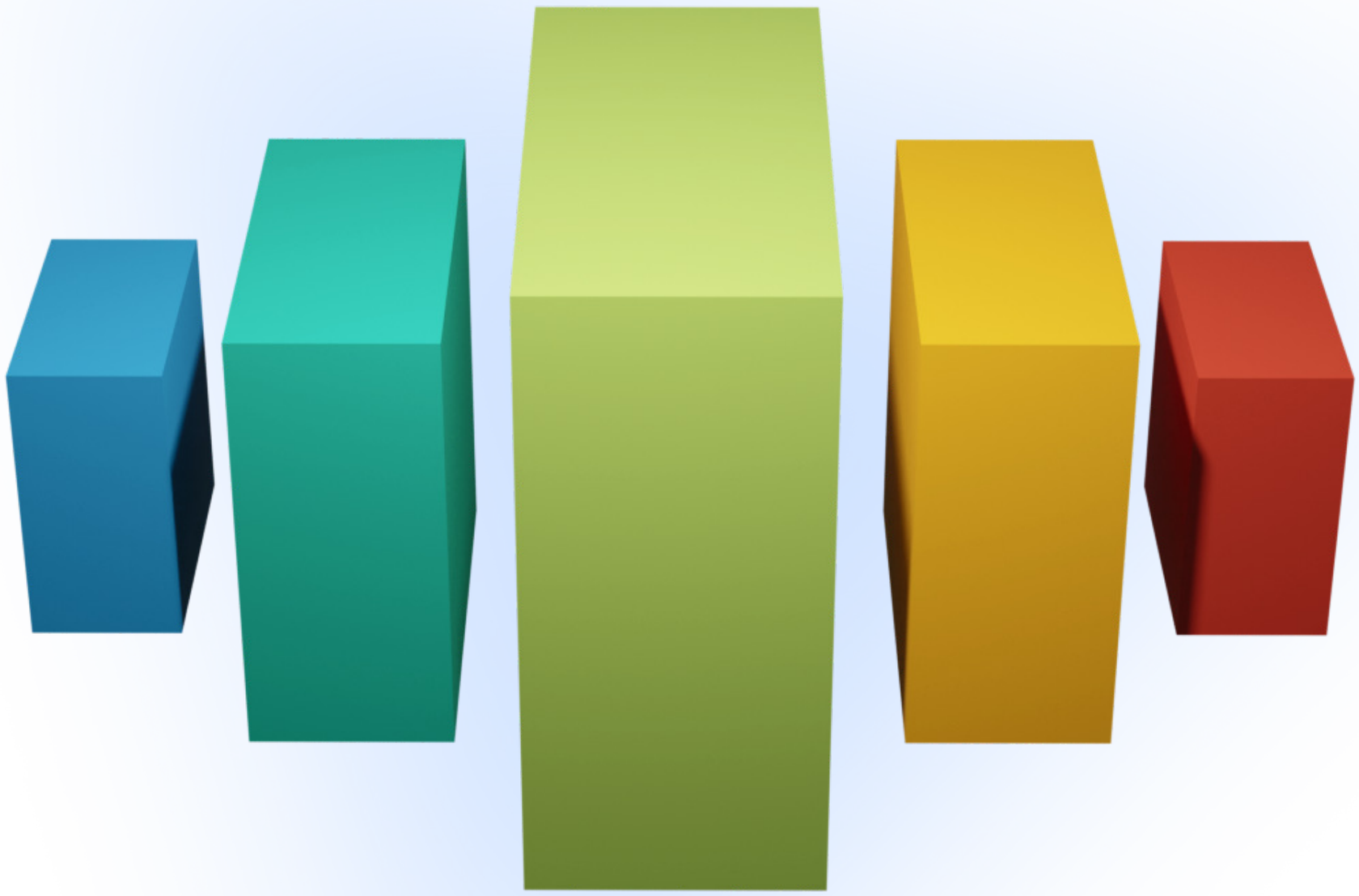
BOSSCODER ACADEMY

The Ultimate Guide to

S.O.L.I.D.

PRINCIPLES



SOLID Principles in Software Development

SOLID principles for software design were given by Robert J. Martin (Uncle Bob) and Michael Feathers.

Promote clean, maintainable, and testable code.

🎯 Purpose

- Create understandable, readable, and testable code.
- Facilitate collaboration among developers.

SOLID Principles

Single Responsibility (SRP)

- Classes have one responsibility.
- Encourages focused classes for easier maintenance.

Open-Closed (OCP)

- Software entities open for extension, closed for modification.
- Facilitates code extension without altering existing code.

Liskov Substitution (LSP)

- Subtypes are substitutable for base types.
- Ensures expected behavior in derived classes.

Interface Segregation (ISP)

- Clients don't depend on unused interfaces.
- Promotes smaller, specialized interfaces to avoid unnecessary dependencies.

Dependency Inversion (DIP)

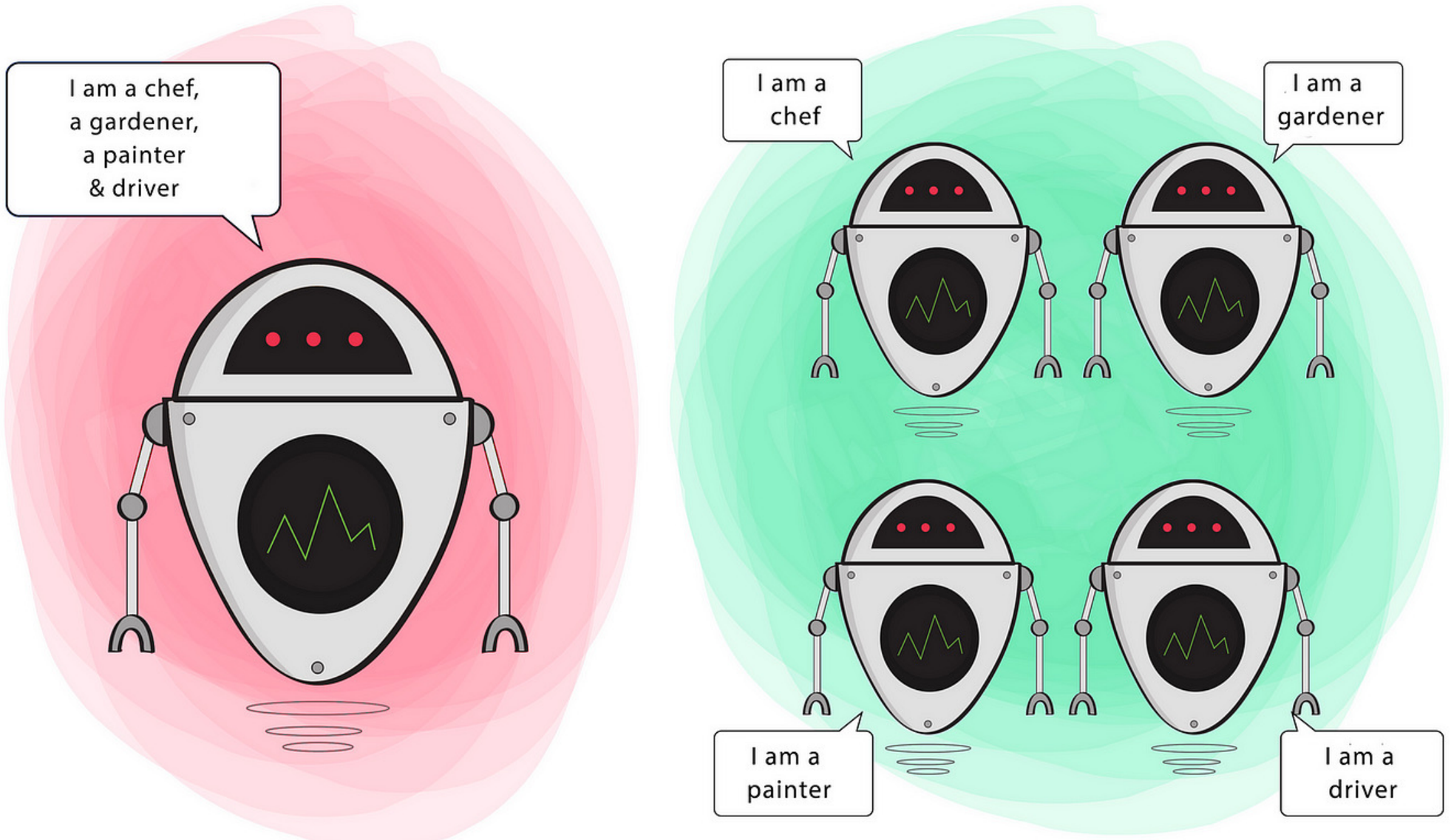
- High-level modules depend on abstractions.
- Encourages flexibility and easy testing.

Single Responsibility Principle (SRP)

It states that a class should have only one reason to change, meaning it should have a single responsibility.

Why SRP Matters?

- Focusing on a single responsibility makes code easier to maintain, understand, and test.
- Prevents a class from becoming overly complex or bloated with unrelated functionality.



Single Responsibility

Source: Medium.com

Example : Email Sender Class

Suppose we have an `EmailSender` class responsible for sending emails.

Before SRP

The `EmailSender` class handles email composition, user authentication, and email sending.

After Applying SRP

We refactor the class to split responsibilities:

a. **`EmailComposer`:** Responsible for composing email content.
b. **`UserAuthenticator`:** Handles user authentication.
c. **`EmailSender`:** Focuses on sending emails.

Benefits

- **Improved Maintainability:** Changes to one responsibility won't affect the others.
- **Better Readability:** Easier to understand each component's role.
- **Efficient Testing:** Isolating components simplifies unit testing.

Open-Closed Principle (OCP)

It suggests that software entities (classes, modules) should be open for extension but closed for modification.

New functionality should be incorporable without requiring modifications to the existing code within a class.

Modifying existing, well-tested code introduces the risk of introducing bugs, which should be avoided whenever possible.

Why OCP Matters?

- OCP promotes software that can be easily extended with new features without changing existing code.
- It reduces the risk of introducing new bugs when making modifications.

Example : Shape Drawing Application

Imagine you have a system that calculates the area of various shapes, such as rectangles and circles.

You want to follow the OCP to allow for easy extension with new shapes without modifying the existing code.

Start with an abstract **Shape** class representing the common properties and methods of all shapes:

```
abstract class Shape {
    public abstract double area();
}
```

Implement concrete shapes like **`Rectangle`** and **`Circle`** by extending the **Shape** class:

```
class Rectangle extends Shape {
    private double width;
    private double height;
    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

    @Override
    public double area() {
        return width * height;
    }
}
class Circle extends Shape {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    @Override
    public double area() {
        return Math.PI * radius * radius;
    }
}
```

Now, if you want to add a new shape, such as a triangle, you can do so without modifying the existing code. You create a new **`Triangle`** class that extends **`Shape`**:

```
class Triangle extends Shape {
    private double base;
    private double height;

    public Triangle(double base, double height) {
        this.base = base;
        this.height = height;
    }

    @Override
    public double area() {
        return 0.5 * base * height;
    }
}
```

By following the Open-Closed Principle, you can easily add new shapes to your system without altering the existing code that deals with calculating areas.

Benefits

- **Extensibility:** Easily add new features without altering existing code.
- **Reduced Risk:** Modifying existing code can introduce bugs, and OCP helps minimize this risk.
- **Maintainability:** Code remains stable and easier to manage over time.

Liskov Substitution Principle (LSP)

It emphasizes that objects of derived classes must be substitutable for objects of their base classes without affecting the program's correctness.

When class B inherits from class A as a subclass, it should seamlessly work as a substitute for class A in any method that expects an object of class A.

Why LSP Matters

- LSP ensures that a derived class doesn't violate the expected behavior of the base class.
- It maintains consistency in polymorphic behavior, improving code reliability.

Example:

Let's consider an example using a classic geometric shape hierarchy:

java

```
class Shape {
    public double getArea() {
        return 0.0;
    }
}
```

```
class Circle extends Shape {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    @Override
    public double getArea() {
        return Math.PI * radius * radius;
    }
}

class Square extends Shape {
    private double side;

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

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

In this example:

1. The **`Shape`** class is the base class, defining a **`getArea`** method that returns 0 as a default implementation.
2. **`Circle`** and **`Square`** are subclasses that extend **`Shape`** and provide their own implementations of **`getArea`**. These implementations calculate the area of a circle and a square, respectively.

Here's how LSP is applied:

```
Here's how LSP is applied:
Shape circleShape = new Circle(5.0);
Shape squareShape = new Square(4.0);

double circleArea = circleShape.getArea();
double squareArea = squareShape.getArea();

System.out.println("Circle Area: " + circleArea);
System.out.println("Square Area: " + squareArea);
```

In this code, we create instances of **`Circle`** and **`Square`** but store them in variables of type **`Shape`**, the base class. The Liskov Substitution Principle ensures that we can use these subclasses wherever a **`Shape`** is expected.

Benefits

- **Polymorphic Behavior:** LSP ensures that derived classes behave as expected when used polymorphically.
- **Consistency:** Code relying on base classes can work with derived classes seamlessly.
- **Reduced Bugs:** Improved program correctness leads to fewer unexpected issues.

Interface Segregation Principle (ISP)

It emphasizes that clients should not be forced to depend on interfaces they do not use.

Why ISP Matters

- ISP reduces unnecessary dependencies, leading to more cohesive and maintainable code.
- It ensures that clients only implement the methods they require, improving code clarity.

"Clients should not be forced to depend upon interfaces that they don't use"

Good

"Segregate your interfaces"

Example:

Let's consider an example in the context of a software application for a document management system. Suppose we have an interface called **`Document`** that initially includes methods for creating, editing, and sharing documents:

```
public interface Document {
    void create();
    void edit();
    void share();
}
```

However, in our application, we have two types of documents: **`TextDocument`** and **`SpreadsheetDocument`**. While the **`TextDocument`** class can implement all the methods in the **`Document`** interface, the **`SpreadsheetDocument`** class doesn't need the **`edit`** method since spreadsheets don't have a typical "editing" action. This violates the ISP because it forces the **`SpreadsheetDocument`** class to implement methods it doesn't need.

To adhere to the ISP, we should refactor the interface to make it more granular, like this:

```
public interface Createable {
    void create();
}

public interface Editable {
    void edit();
}
```

```
public interface Shareable {
    void share();
}
```

Now, our classes can implement only the interfaces that are relevant to them:

```
public class TextDocument implements Createable,
Editable, Shareable {
    // Implement the methods for text documents.
}
public class SpreadsheetDocument implements Createable,
Shareable {
    // Implement the methods for spreadsheet documents.
}
```

By adhering to the ISP, we've created smaller, more focused interfaces that allow classes to implement only what they need, reducing unnecessary dependencies and making the code more maintainable and flexible.

Benefits

- **Reduced Dependencies:** Clients depend only on the interfaces they need, reducing unnecessary coupling.
- **Improved Clarity:** Code becomes more self-explanatory as clients implement interfaces that align with their specific functionality.
- **Easier Maintenance:** Changes or additions to interfaces impact only relevant clients, reducing ripple effects

Dependency Inversion Principle (DIP)

It states that high-level modules should not depend on low-level modules, but both should depend on abstractions. In other words, the details should depend on abstractions, not the other way around.

Why DIP Matters

DIP promotes loose coupling between components, making software more flexible and maintainable. It enables the creation of interchangeable, pluggable components that can be easily substituted without affecting the overall system.

"Higher modules should not depend on lower modules. In such cases invert the dependency"

Example: Messaging System

Consider a messaging system that sends notifications through various channels: email, SMS, and push notifications.

Violation of DIP Without following DIP, the high-level NotificationService class directly depends on low-level implementations:

```
class NotificationService {
    private EmailSender emailSender;
    private SMSSender smsSender;
    private PushNotificationSender pushSender;

    public NotificationService() {
        emailSender = new EmailSender();
        smsSender = new SMSSender();
        pushSender = new PushNotificationSender();
    }

    public void sendEmail() {
        emailSender.send();
    }

    public void sendSMS() {
        smsSender.send();
    }

    public void sendPushNotification() {
        pushSender.send();
    }
}
```

Applying DIP Following DIP, we use abstractions and interfaces to invert the dependencies:

```
interface MessageSender {
    void send();
}

class EmailSender implements MessageSender {
    @Override
    public void send() {
        // Send email
    }
}
class SMSSender implements MessageSender {
    @Override
    public void send() {
        // Send SMS
    }
}
class PushNotificationSender implements MessageSender {
    @Override
    public void send() {
        // Send push notification
    }
}
class NotificationService {
    private MessageSender sender;

    public NotificationService(MessageSender sender) {
        this.sender = sender;
    }

    public void send() {
        sender.send();
    }
}
```

By following the Dependency Inversion Principle, the NotificationService is no longer tightly coupled to specific sender implementations.

You can easily add new sender types without modifying the service, promoting flexibility and maintainability.

Benefits

- **Flexibility:** Easily swap or extend components without affecting high-level modules.
- **Scalability:** Allows for the addition of new features or integrations with minimal code changes.
- **Testability:** Simplifies unit testing by enabling the use of mock or stub implementations for testing.

BOSSCODER ACADEMY

WHY BOSSCODER?

1000+ Alumni placed at Top Product-based companies.

More than **136% hike** for every **2 out of 3** working professional.

Average package of **24LPA.**

The syllabus is most **up-to-date** and the list of problems provided covers **all important topics.**

Lavanya
Meta

Course is very well **structured** and **streamlined** to crack any **MAANG** company

Rahul .
Google

EXPLORE MORE