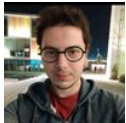


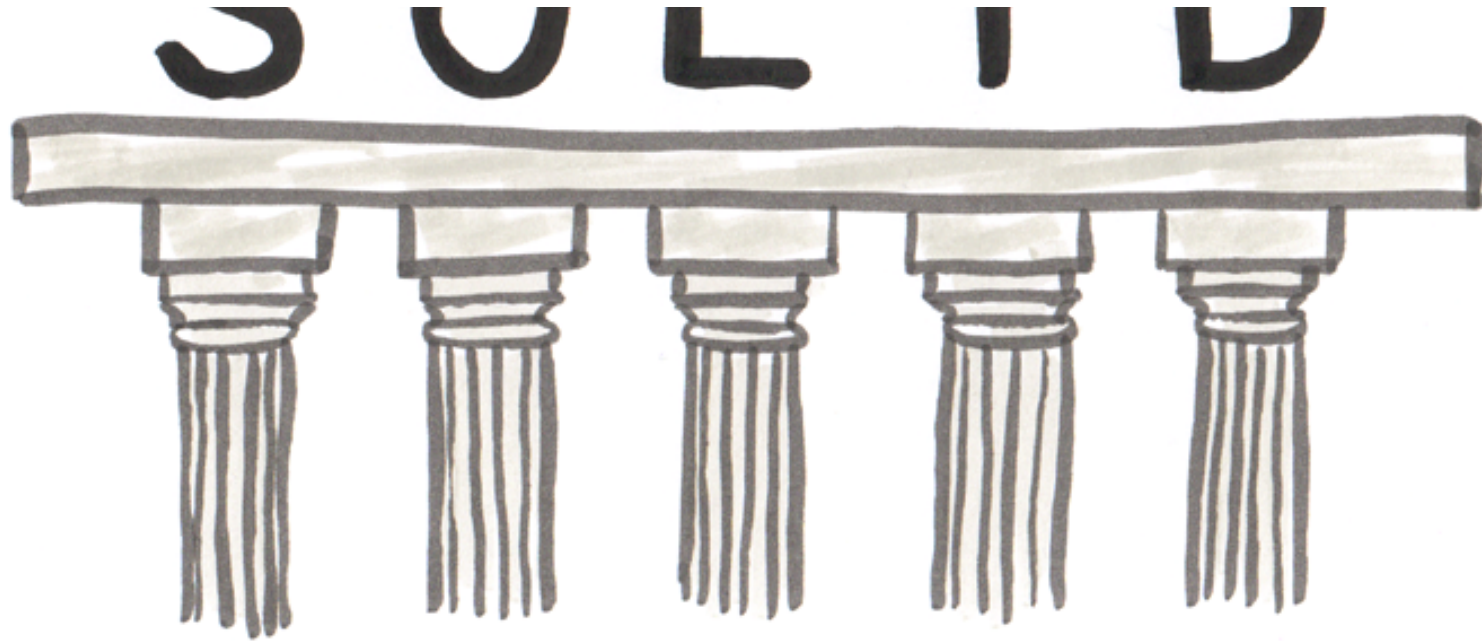
AUGUST 20, 2020 / [#OBJECT ORIENTED PROGRAMMING](#)

# The SOLID Principles of Object-Oriented Programming Explained in Plain English



Yiğit Kemal Erinç

Learn to code — free 3,000-hour curriculum



The SOLID Principles are five principles of Object-Oriented class design. They are a set of rules and best practices to follow while designing a class structure.

These five principles help us understand the need for certain design patterns and software architecture in general. So I believe that it is a topic that every developer should learn.

This article will teach you everything you need to know to apply SOLID principles to your

Learn to code — free 3,000-hour curriculum

nitty-gritty details – the why's and how's of each principle – by creating a class design and improving it step by step.

So grab a cup of coffee or tea and let's jump right in!

## Background

The SOLID principles were first introduced by the famous Computer Scientist Robert J. Martin (a.k.a Uncle Bob) in his paper in 2000. But the SOLID acronym was introduced later by Michael Feathers.

Uncle Bob is also the author of bestselling books *Clean Code* and *Clean Architecture*, and is one of the participants of the "Agile Alliance".

Therefore, it is not a surprise that all these concepts of clean coding, object-oriented architecture, and design patterns are somehow connected and complementary to each other.

They all serve the same purpose:

"To create understandable, readable, and testable code that many developers can collaboratively work on."

Learn to code — free 3,000-hour curriculum

- The **O**pen-Closed Principle
- The **L**iskov Substitution Principle
- The **I**nterface Segregation Principle
- The **D**ependency Inversion Principle

## The Single Responsibility Principle

The Single Responsibility Principle states that a class should do one thing and therefore it should have only a single reason to change.

To state this principle more technically: Only one potential change (database logic, logging logic, and so on.) in the software's specification should be able to affect the specification of the class.

This means that if a class is a data container, like a Book class or a Student class, and it has some fields regarding that entity, it should change only when we change the data model.

Following the Single Responsibility Principle is important. First of all, because many different teams can work on the same project and edit the same class for different reasons, this could lead to incompatible modules.

Learn to code — free 3,000-hour curriculum

following the SRP, we will know that it is related to storage or database-related stuff.

Merge conflicts are another example. They appear when different teams change the same file. But if the SRP is followed, fewer conflicts will appear – files will have a single reason to change, and conflicts that do exist will be easier to resolve.

## Common Pitfalls and Anti-patterns

In this section we will look at some common mistakes that violate the Single Responsibility Principle. Then we will talk about some ways to fix them.

We will look at the code for a simple bookstore invoice program as an example. Let's start by defining a book class to use in our invoice.

```
class Book {
    String name;
    String authorName;
    int year;
    int price;
    String isbn;

    public Book(String name, String authorName, int year, int price, String isbn) {
        this.name = name;
        this.authorName = authorName;
    }
}
```

## Learn to code — free 3,000-hour curriculum

```
}
```

This is a simple book class with some fields. Nothing fancy. I am not making fields private so that we don't need to deal with getters and setters and can focus on the logic instead.

Now let's create the invoice class which will contain the logic for creating the invoice and calculating the total price. For now, assume that our bookstore only sells books and nothing else.

```
public class Invoice {  
  
    private Book book;  
    private int quantity;  
    private double discountRate;  
    private double taxRate;  
    private double total;  
  
    public Invoice(Book book, int quantity, double discountRate, double taxRate) {  
        this.book = book;  
        this.quantity = quantity;  
        this.discountRate = discountRate;  
        this.taxRate = taxRate;  
        this.total = this.calculateTotal();  
    }  
}
```

## Learn to code — free 3,000-hour curriculum

```
double priceWithTaxes = price * (1 + taxRate);

return priceWithTaxes;
}

public void printInvoice() {
    System.out.println(quantity + "x " + book.name + " " + book.price + "$");
    System.out.println("Discount Rate: " + discountRate);
    System.out.println("Tax Rate: " + taxRate);
    System.out.println("Total: " + total);
}

    public void saveToFile(String filename) {
// Creates a file with given name and writes the invoice
    }
}
```

Here is our invoice class. It also contains some fields about invoicing and 3 methods:

- **calculateTotal** method, which calculates the total price,
- **printInvoice** method, that should print the invoice to console, and
- **saveToFile** method, responsible for writing the invoice to a file.

You should give yourself a second to think about what is wrong with this class design before

Learn to code — free 3,000-hour curriculum

OK so what's going on here: Our class violates the single responsibility principle in multiple ways.

The first violation is the **printInvoice** method, which contains our printing logic. The SRP states that our class should only have a single reason to change, and that reason should be a change in the invoice calculation for our class.

But in this architecture, if we wanted to change the printing format, we would need to change the class. This is why we should not have printing logic mixed with business logic in the same class.

There is another method that violates the SRP in our class: the **saveToFile** method. It is also an extremely common mistake to mix persistence logic with business logic.

Don't just think in terms of writing to a file – it could be saving to a database, making an API call, or other stuff related to persistence.

So how can we fix this print function, you may ask.

We can create new classes for our printing and persistence logic so we will no longer need to modify the invoice class for those purposes.

We create 2 classes, **InvoicePrinter** and **InvoicePersistence**, and move the methods.



## Learn to code — free 3,000-hour curriculum

```
public class InvoicePrinter {
    private Invoice invoice;

    public InvoicePrinter(Invoice invoice) {
        this.invoice = invoice;
    }

    public void print() {
        System.out.println(invoice.quantity + "x " + invoice.book.name + " " + invoice.book.price
        System.out.println("Discount Rate: " + invoice.discountRate);
        System.out.println("Tax Rate: " + invoice.taxRate);
        System.out.println("Total: " + invoice.total + " $");
    }
}
```

```
public class InvoicePersistence {
    Invoice invoice;

    public InvoicePersistence(Invoice invoice) {
        this.invoice = invoice;
    }

    public void saveToFile(String filename) {
        // Creates a file with given name and writes the invoice
    }
}
```

Learn to code — free 3,000-hour curriculum

responsible for one aspect of our application. Great!

## Open-Closed Principle

The Open-Closed Principle requires that **classes should be open for extension and closed to modification**.

Modification means changing the code of an existing class, and extension means adding new functionality.

So what this principle wants to say is: We should be able to add new functionality without touching the existing code for the class. This is because whenever we modify the existing code, we are taking the risk of creating potential bugs. So we should avoid touching the tested and reliable (mostly) production code if possible.

But how are we going to add new functionality without touching the class, you may ask. It is usually done with the help of interfaces and abstract classes.

Now that we have covered the basics of the principle, let's apply it to our Invoice application.

Let's say our boss came to us and said that they want invoices to be saved to a database so that we can search them easily. We think okay, this is easy peasy boss, just give me a second!

## Learn to code — free 3,000-hour curriculum

```
public class InvoicePersistence {  
    Invoice invoice;  
  
    public InvoicePersistence(Invoice invoice) {  
        this.invoice = invoice;  
    }  
  
    public void saveToFile(String filename) {  
        // Creates a file with given name and writes the invoice  
    }  
  
    public void saveToDatabase() {  
        // Saves the invoice to database  
    }  
}
```

Unfortunately we, as the lazy developer for the book store, did not design the classes to be easily extendable in the future. So in order to add this feature, we have modified the **InvoicePersistence** class.

If our class design obeyed the Open-Closed principle we would not need to change this class.

So, as the lazy but clever developer for the book store, we see the design problem and decide

Learn to code — free 3,000-hour curriculum

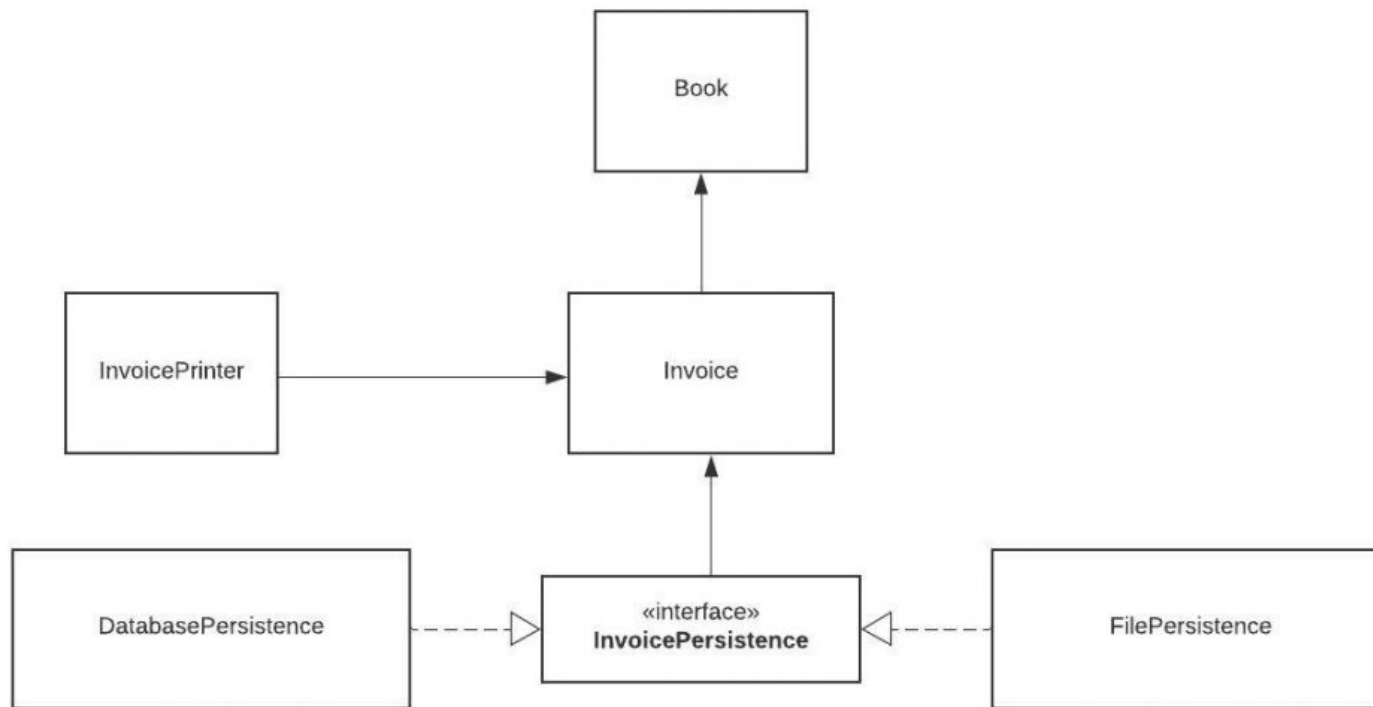
```
interface InvoicePersistence {  
  
    public void save(Invoice invoice);  
}
```

We change the type of **InvoicePersistence** to Interface and add a save method. Each persistence class will implement this save method.

```
public class DatabasePersistence implements InvoicePersistence {  
  
    @Override  
    public void save(Invoice invoice) {  
        // Save to DB  
    }  
}
```

```
public class FilePersistence implements InvoicePersistence {  
  
    @Override  
    public void save(Invoice invoice) {  
        // Save to file  
    }  
}
```

So our class structure now looks like this:



Now our persistence logic is easily extendable. If our boss asks us to add another database and have 2 different types of databases like MySQL and MongoDB, we can easily do that.

Learn to code — free 3,000-hour curriculum

method to all of them.

But let's say that we extend our app and have multiple persistence classes like **InvoicePersistence**, **BookPersistence** and we create a **PersistenceManager** class that manages all persistence classes:

```
public class PersistenceManager {  
    InvoicePersistence invoicePersistence;  
    BookPersistence bookPersistence;  
  
    public PersistenceManager(InvoicePersistence invoicePersistence,  
                             BookPersistence bookPersistence) {  
        this.invoicePersistence = invoicePersistence;  
        this.bookPersistence = bookPersistence;  
    }  
}
```

We can now pass any class that implements the **InvoicePersistence** interface to this class with the help of polymorphism. This is the flexibility that interfaces provide.

## Liskov Substitution Principle

The Liskov Substitution Principle states that subclasses should be substitutable for their base classes.

Learn to code — free 3,000-hour curriculum

of class B to any method that expects an object of class A and the method should not give any weird output in that case.

This is the expected behavior, because when we use inheritance we assume that the child class inherits everything that the superclass has. The child class extends the behavior but never narrows it down.

Therefore, when a class does not obey this principle, it leads to some nasty bugs that are hard to detect.

Liskov's principle is easy to understand but hard to detect in code. So let's look at an example.

```
class Rectangle {  
    protected int width, height;  
  
    public Rectangle() {  
    }  
  
    public Rectangle(int width, int height) {  
        this.width = width;  
        this.height = height;  
    }  
  
    public int getWidth() {  
        return width;  
    }  
}
```

Learn to code — free 3,000-hour curriculum

```
        this.width = width;
    }

    public int getHeight() {
        return height;
    }

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

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

We have a simple Rectangle class, and a **getArea** function which returns the area of the rectangle.

Now we decide to create another class for Squares. As you might know, a square is just a special type of rectangle where the width is equal to the height.

```
class Square extends Rectangle {
    public Square() {}
}
```



Learn to code — free 3,000-hour curriculum

```
@Override
public void setWidth(int width) {
    super.setWidth(width);
    super.setHeight(width);
}

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

Our Square class extends the Rectangle class. We set height and width to the same value in the constructor, but we do not want any client (someone who uses our class in their code) to change height or weight in a way that can violate the square property.

Therefore we override the setters to set both properties whenever one of them is changed. But by doing that we have just violated the Liskov substitution principle.

Let's create a main class to perform tests on the **getArea** function.

```
class Test {
```

Learn to code — free 3,000-hour curriculum

```
r.setHeight(10);
System.out.println("Expected area of " + (width * 10) + ", got " + r.getArea());
}

public static void main(String[] args) {
    Rectangle rc = new Rectangle(2, 3);
    getAreaTest(rc);

    Rectangle sq = new Square();
    sq.setWidth(5);
    getAreaTest(sq);
}
```

Your team's tester just came up with the testing function **getAreaTest** and tells you that your **getArea** function fails to pass the test for square objects.

In the first test, we create a rectangle where the width is 2 and the height is 3 and call **getAreaTest**. The output is 20 as expected, but things go wrong when we pass in the square. This is because the call to **setHeight** function in the test is setting the width as well and results in an unexpected output.

## Interface Segregation Principle

Segregation means keeping things separated and the Interface Segregation Principle is

## Learn to code — free 3,000-hour curriculum

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.

This is a simple principle to understand and apply, so let's see an example.

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

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.

```
public class FreeParking implements ParkingLot {
```

## Learn to code — free 3,000-hour curriculum

```
}

@Override
public void unparkCar() {

}

@Override
public void getCapacity() {

}

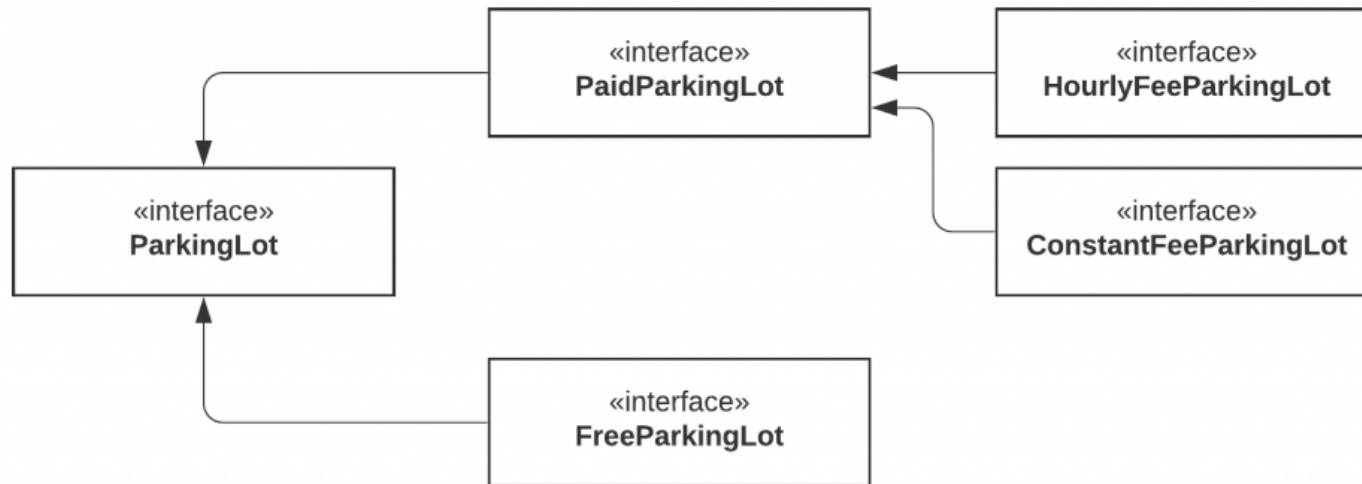
@Override
public double calculateFee(Car car) {
    return 0;
}

@Override
public void doPayment(Car car) {
    throw new Exception("Parking lot is free");
}
}
```

Our parking lot interface was composed of 2 things: Parking related logic (park car, unpark car, get capacity) and payment related logic.

But it is too specific. Because of that, our FreeParking class was forced to implement payment-related methods that are irrelevant. Let's separate or segregate the interfaces.

Learn to code — free 3,000-hour curriculum



We've now separated the parking lot. With this new model, we can even go further and split the **PaidParkingLot** to support different types of payment.

Now our model is much more flexible, extendable, and the clients do not need to implement any irrelevant logic because we provide only parking-related functionality in the parking lot interface.

## Dependency Inversion Principle

The Dependency Inversion principle states that our classes should depend upon interfaces or

Learn to code — free 3,000-hour curriculum

In his [article](#) (2000), Uncle Bob summarizes this principle as follows:

"If the OCP states the goal of OO architecture, the DIP states the primary mechanism".

These two principles are indeed related and we have applied this pattern before while we were discussing the Open-Closed Principle.

We want our classes to be open to extension, so we have reorganized our dependencies to depend on interfaces instead of concrete classes. Our PersistenceManager class depends on InvoicePersistence instead of the classes that implement that interface.

## Conclusion

In this article, we started with the history of SOLID principles, and then we tried to acquire a clear understanding of the why's and how's of each principle. We even refactored a simple Invoice application to obey SOLID principles.

I want to thank you for taking the time to read the whole article and I hope that the above concepts are clear.

I suggest keeping these principles in mind while designing, writing, and refactoring your code so that your code will be much more clean, extendable, and testable.

Learn to code — free 3,000-hour curriculum

list to get notified when I publish a new article.



### Yiğit Kemal Erinoç

I am an MSc. student at TU Munich and a Fullstack engineer with industry experience in backend development. I like helping and motivating other people so feel free to reach me out :)

---

If you read this far, tweet to the author to show them you care.

[Tweet a thanks](#)

Learn to code for free. freeCodeCamp's open source curriculum has helped more than 40,000 people get jobs as developers.

[Get started](#)

freeCodeCamp is a donor-supported tax-exempt 501(c)(3) nonprofit organization (United States Federal Tax Identification Number: 82-0779546)

Our mission: to help people learn to code for free. We accomplish this by creating thousands of videos, articles, and interactive coding lessons - all freely available to the public. We also have thousands of freeCodeCamp study groups around the world.

Learn to code — free 3,000-hour curriculum

You can [make a tax-deductible donation here](#).

### Trending Guides

[What is Docker?](#)

[TCP/IP Model](#)

[RTF File](#)

[CSS Transition](#)

[How to Use Instagram?](#)

[MBR VS GPT](#)

[FAT32 Format](#)

[Error 503 Code](#)

[Windows Hosts File](#)

[Mobi to PDF](#)

[WordPress for Beginners](#)

[Qualitative VS Quantitative](#)

[JavaScript Split String](#)

[Accented Letters on Mac](#)

[Windows 10 Product Key](#)

[What is STEM?](#)

[JavaScript Void 0](#)

[SQL Delete Row](#)

[JavaScript Replace](#)

[Python JSON Parser](#)

[cmd Delete Folder](#)

[What is NFC?](#)

[Content Type JSON](#)

[Convert HEIC to JPG](#)

[Math Random Java](#)

[Google Docs Landscape](#)

[Antimalware Executable](#)

[Windows 10 Start Menu](#)

[Windows 10 Command Line](#)

[Google Account Recovery](#)



[Forum](#)

[Donate](#)

[Learn to code — free 3,000-hour curriculum](#)

[Terms of Service](#) [Copyright Policy](#)