BUILDING BETTER CODE WITH

SOLID PRINCIPLES

By, Kanishkumar

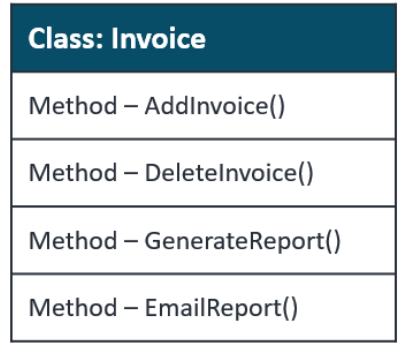
SOLID PRINCIPLES

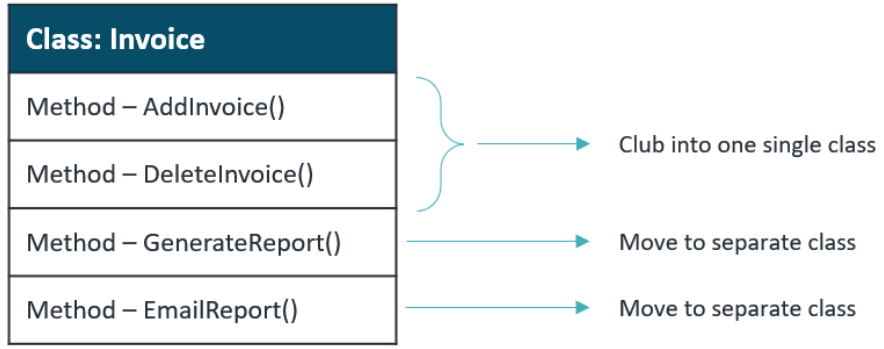
- The SOLID principles are a set of design principles that aim at improving the quality and maintainability of the software.
- Although originally formulated for object-oriented programming, many of these principles can efficiently be applied to JavaScript as well.
- The SOLID JavaScript principles provide developers with a set of guidelines to write software that is Modular, easy to understand and easy to modify.
- By adhering to these SOLID principles, you can create a software that is more **flexible and scalable**.

SINGLE RESPONSIBILITY PRINCIPLE

SINGLE RESPONSIBILITY PRINCIPLE

- The Single Responsibility Principle emphasizes that each class or module should have only one responsibility.
- It should only have one reason to change. This means it should handle one specific aspect of the functionality, and any change to that aspect should only impact that class or module.
- In JavaScript, this can be achieved by breaking down complex functions into smaller ones that perform a single task.





```
class Library {
        constructor() {
            this.books = [];
        }

        addBook(title, author) {
            this.books.push({ title, author });
        }

        getBooks() {
            return this.books;
        }

        printBooks(books) {
            this.books.forEach(book => {
                console.log(`Title: ${book.title}, Author: ${book.author}`);
            });
        }
    }

const library = new Library();
library.addBook('Python', 'XXXX');
library.addBook('Java', 'YYYY');
library.printBooks();
```

Pros Of The Single Responsibility Principle:

- Easier to maintain and modify the code
- Enhanced modular code, reusable code
- Easier to read and test the code
- Less prone to errors

Cons Of Single Responsibility Principle:

- Increases the number of modules in the codebase
- Can require more time to implement.

OPEN CLOSED PRINCIPLE

OPEN CLOSED PRINCIPLE

- The Open Closed Principle states that the classes or modules should be open for an extension but closed for modification.

- This means that you should be able to add new functionality to a class or module without modifying its existing code.

- In JavaScript, this can be achieved by using interfaces, abstract classes, and inheritance.

```
// src/BookPrinter.js

class BookPrinter {
        static printBooks(books, format = 'text') {
        switch (format) {
            case 'html':
                let html = '<ul>';
                books.forEach(book => {
                    html += `<li>Title: ${book.title}, Author: ${book.author}</li>`;
                });
                html += '</ul>';
                console.log(html);
                break;

            case 'csv':
                let csv = 'Title,Author\n';
                books.forEach(book => {
                    csv += `${book.title},${book.author}\n`;
                });
                console.log(csv);
                break;

            case 'text':
            default:
                books.forEach(book => {
                    console.log(`Title: ${book.title}, Author: ${book.author}`);
                });
                break;
    }
  }}
```

Pros Of Open-closed principle:

- More flexible code
- Easier to modify and extend code
- Less prone to errors

Cons Of Open-closed principle:

- Can require more time to implement
- Can increase the complexity of the codebase

LISCOV

SUBSTITUTION PRINCIPLE

LISCOV SUBSTITUTION PRINCIPLE

- The Liskov Substitution Principle states **that the object of a superclass shall be replaceable with objects of its subclasses without breaking the application**.

- In JavaScript, this can be achieved by ensuring that the subclasses inherit the same properties and methods as their super classes.

EXPECTATIONS OF THE SUPERCLASS FROM THE SUBCLASS

- The subclass should maintain the same behavior as the superclass and not introduce inconsistencies.
- Preconditions: Do not require stricter conditions than the superclass.
- Maintain the guarantees provided by the superclass.
- Ensure the subclass does not compromise the correctness of the superclass functionality.

```
class Rectangle {
    constructor(width, height) {
        this.width = width;
        this.height = height;
    }

    getArea() {
        return this.width * this.height;
    }
}
```

```
class Square extends Rectangle {
    constructor(side) {
        super(side, side);
    }

    set width(value) {
        this._width = value;
        this._height = value;
    }

    set height(value) {
        this._height = value;
        this._width = value;
    }
}
```

Behavioral Inconsistency (where width and height are independent.)
Preconditions: The `Square` imposes a stricter condition (width must equal height),

```
class Shape {
    getArea() {
        throw new Error("Method 'getArea()' must be
implemented.");
    }
}
```

```
class Rectangle extends Shape {
    constructor(width, height) {
        super();
        this.width = width;
        this.height = height;
    }

    getArea() {
        return this.width * this.height;
    }
}
```

```
class Square extends Shape {
    constructor(side) {
        super();
        this.side = side;
    }

    getArea() {
        return this.side * this.side;
    }
}
```

```
class PrintStrategy {
    print(books) {
        throw new Error("Method 'print()' must be implemented.");
    }
```

```
class HTMLPrintStrategy extends PrintStrategy {
    print(books) {
        let html = '<ul>';
        books.forEach(book => {
            html += `<li>${book.title} by ${book.author}</li>`;
        });
        html += '</ul>';
        console.log(html);
    }
}
```

```
class CSVPrintStrategy extends PrintStrategy {
    print(books) {
        let csv = 'Title, Author\n';
        books.forEach(book => {
            csv += `${book.title}, ${book.author}\n`;
        });
        console.log(csv);
    }
}
```

Pros Of Liskov Substitution Principle:

- More predictable code
- Easier to test code
- Less prone to errors

Cons Of Liskov Substitution Principle:

- Can increase the complexity of the codebase
- Might require more time to implement

INTERFACE SEGREGATION PRINCIPLE

INTERFACE SEGREGATION PRINCIPLE

- The Interface Segregation Principle states that clients should not be forced to depend on interfaces that they do not use.

- In JavaScript, **this can be achieved by creating smaller interfaces with fewer methods instead of larger interfaces with numerous methods**.

SupportAgent

1. Add
2. Update

```
class TaskManager {

    addTask(task) { ... }

    updateTask(id, task) {...}

    deleteTask(id) { ...}
}
```

Admin

1. Add
2. Update
3. Delete

Pros Of Interface Segregation Principle:

- More modular code
- Easier to maintain and modify code
- Less prone to errors

Cons Of Interface Segregation Principle:

- Can increase the number of interfaces in the codebase
- Can require more time to implement

DEPENDENCY INVERSION PRINCIPLE

DEPENDENCY INVERSION PRINCIPLE

- The Dependency Inversion Principle states that **high-level modules should not depend on low-level modules, but both should depend on abstractions**.
- The principle aims to reduce the coupling between high-level and low-level modules by introducing abstractions that both layers depend on.
- In JavaScript, this can be **achieved by using dependency injection, where dependencies are passed to a function or class instead of being created inside it.**

Without Dependency injection

```
class laserPrinter {
    print(document) {
        console.log(`Printing document with laser Printer: ${document}`);
    }
}
```

```
class DocumentPrinter {
    constructor() {
        this.laserPrinter = new laserPrinter();
    }

    printDocument(document) {
        this.laserPrinter.print(document);
    }
}

const documentPrinter = new DocumentPrinter();
documentPrinter.printDocument('Important Document');
```

With Dependency injection

```
class laserPrinter {
    print(document) {
        console.log(`Printing document with laser Printer: ${document}`);
    }
}
```

```
class DocumentPrinter {
    constructor(printer) {
        this.printer = printer;
    }

    printDocument(document) {
        this.printer.print(document);
    }
}
```

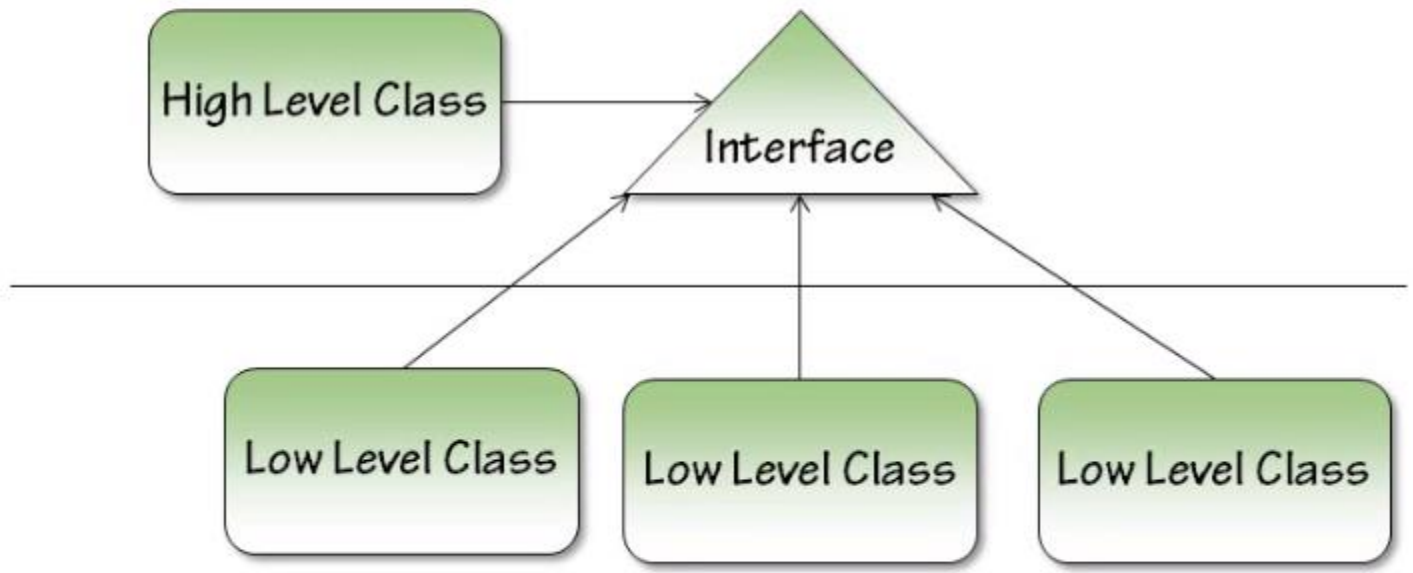
```
const laserPrinter = new laserPrinter();

const documentPrinter = new DocumentPrinter(laserPrinter);

documentPrinter.printDocument('Important Document');
```

- Allows you to change or extend the behavior of a component without modifying its code.
- You can easily swap out different implementations of dependencies.
- Reduces the coupling between components
- Makes Testing easier

Dependency Inversion



Pros Of Dependency Inversion Principle:

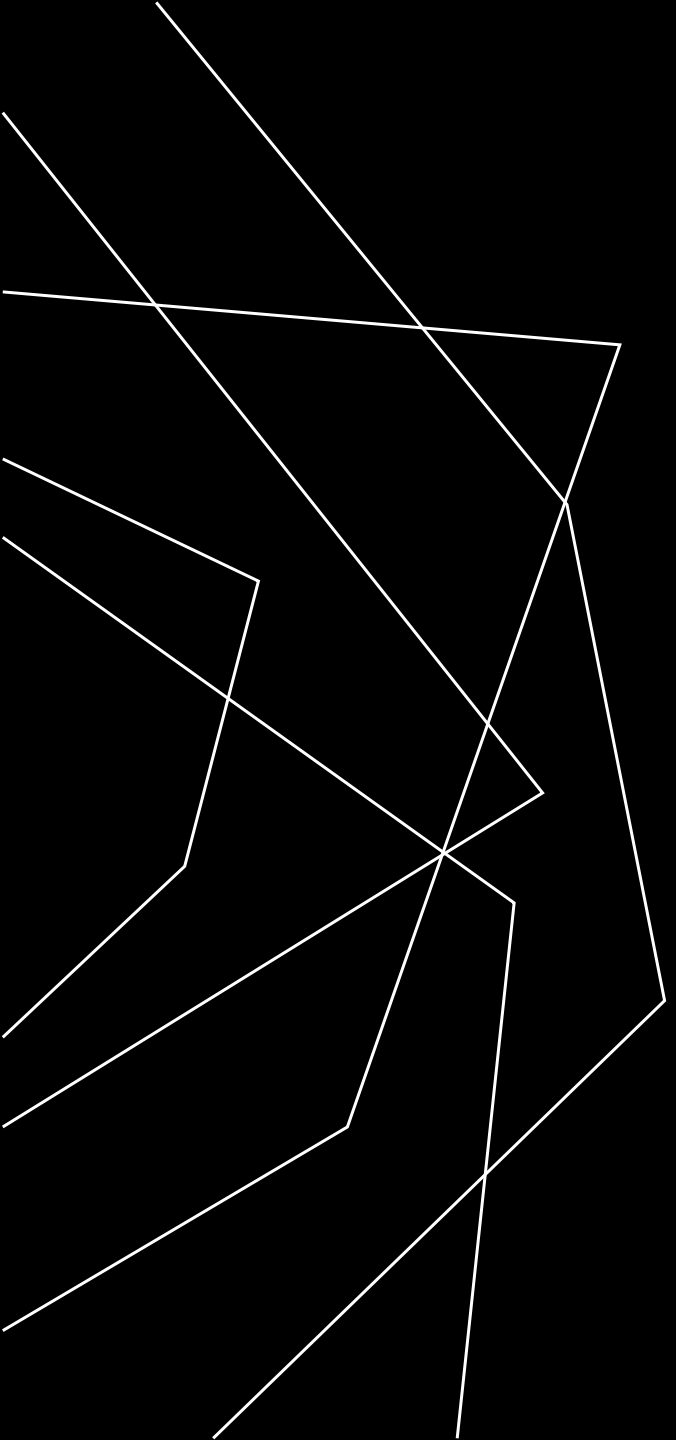
- Promotes loose coupling between modules.
- Increases code flexibility and maintainability.
- Easier to change or modify lower-level modules without affecting higher-level modules.
- Promotes unit testing and improves testability.

Cons Of Dependency Inversion Principle:

- Can be time-consuming to implement in larger projects.
- May require additional interfaces or abstraction layers, which can increase the complexity.

SUMMARY

SOLID Principle	Benefit
Single Responsibility Principle (SRP)	Easier to maintain and update; changes in one responsibility don't affect others.
Open/Closed Principle (OCP)	Existing code remains stable; new features can be added with minimal disruption.
Liskov Substitution Principle (LSP)	Subtypes can be used without altering program behavior or introducing bugs.
Interface Segregation Principle (ISP)	Clients are not forced to implement unused methods, leading to cleaner, more focused interfaces.
Dependency Inversion Principle (DIP)	High-level code is decoupled from low-level details, making the system more flexible and easier to extend.



THANK YOU