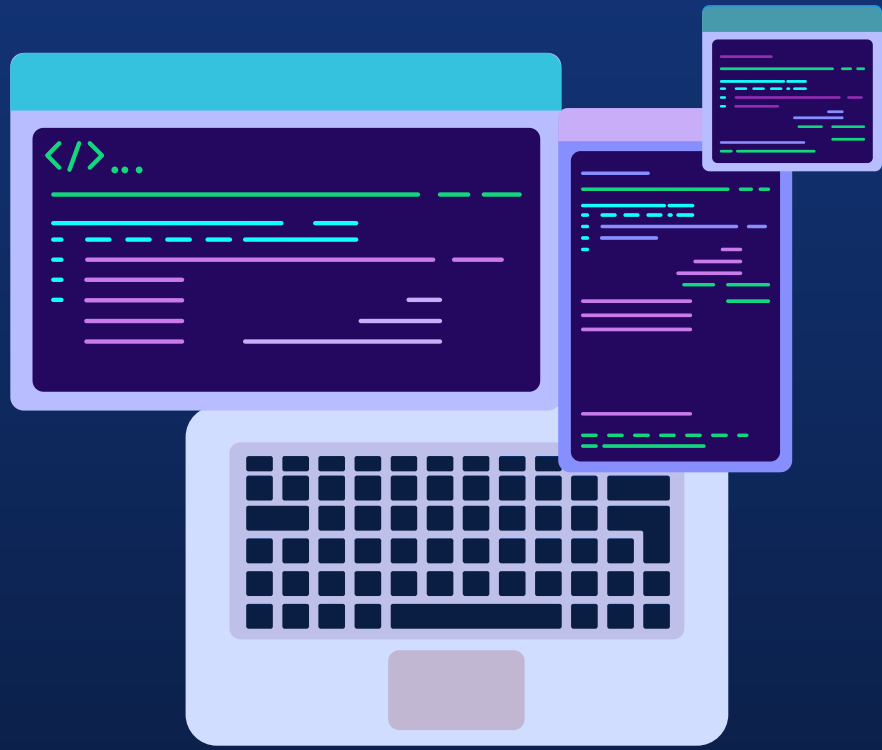
SOLID PRINCIPLES

In Object-oriented programming



About us

{

name: Gabriel Jonay Vera Estévez

email: alu0101398198@ull.edu.es

alternativeEmail: jonay.vera.32@ull.edu.es

}

{

name: Ginés Cruz Chávez

email: alu0101431079@ull.edu.es

alternativeEmail: gines.cruz.30@ull.edu.es

}

TABLE OF CONTENTS

01
The SOLID Principles
What are the SOLID Principles?

02
S-Principle
Single-responsibility principle

03
O-Principle
Open–closed principle

04
L-Principle
Liskov substitution principle

05
I-Principle
Interface segregation principle

06
D-Principle
Dependency inversion principle

01

Introduction to SOLID Principles

“The SOLID principles are a set of guidelines for software development that help us to create more maintainable, understandable, and flexible code. “

— Robert C. Martin

BENEFITS OF THE SOLID PRINCIPLES

- Increased maintainability and flexibility
- Better scalability and reusability
- Reduced coupling and improved cohesion
- Easier testing and debugging

02

Single-responsibility principle

“A class should have one and only one reason to change, meaning that a class should have only one job.”

— **Robert C. Martin**

```typescript
class User {
  private name: string;
  private email: string;

  constructor(name: string, email: string) {
    this.name = name;
    this.email = email;
  }

  public getName(): string {
    return this.name;
  }

  public getEmail(): string {
    return this.email;
  }

  public save(): void {
    // Save user to database
    return;
  }

  public getUserByEmail(email: string): User | undefined {
    // Get user from database
    return undefined;
  }
}
```

```typescript
interface IUserData {
  name: string;
  email: string;
}

class User {
  private userData: IUserData;

  constructor(userData: IUserData) {
    this.userData = userData;
  }

  public getName(): string {
    return this.userData.name;
  }

  public getEmail(): string {
    return this.userData.email;
  }
}

class UserManager {
  private users: User[];

  constructor() {
    this.users = [];
  }

  public addUser(userData: IUserData): void {
    const user = new User(userData);
    this.users.push(user);
  }

  public getUserByEmail(email: string): User | undefined {
    return this.users.find(user => user.getEmail() === email);
  }
}
```

DOS AND DON'TS

DON'T

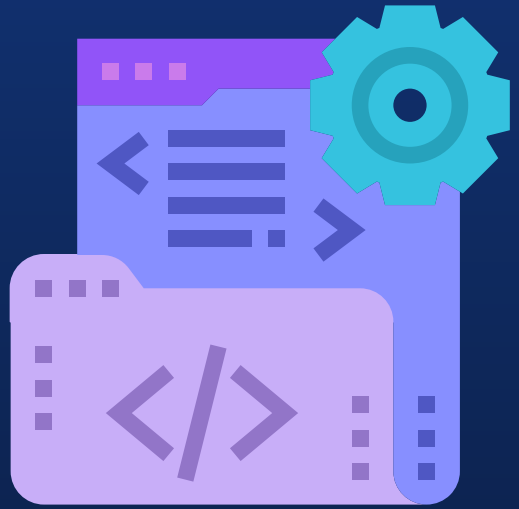
- Have multiple responsibilities in a single class.
- Mix concerns in functions or methods.
- Create methods that rely on internal state that isn't related to the method's purpose.

DO

- Separate concerns into different classes or modules.
- Focus on one responsibility per class.
- Use interfaces and abstract classes to define contracts

03

Open-closed principle



“Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification. This means that you should be able to extend the functionality of a software entity without modifying its existing code.”

— **Robert C. Martin**

```typescript
class Square {
  public length: number;

  constructor(length: number) {
    this.length = length;
  }
}

class Circle {
  public radius: number;

  constructor(radius: number) {
    this.radius = radius;
  }

  getRadius(): number {
    return this.radius;
  }
}

class AreaCalculator {

  constructor(private shapes: (Circle | Square)[]) { }

  public sum(): number {
    let area: number = 0;

    for (let shape of this.shapes) {
      if (shape instanceof Square) {
        area += Math.pow(shape.length, 2);
      } else if (shape instanceof Circle) {
        area += Math.PI * Math.pow(shape.getRadius(), 2);
      }
    }
    return area;
  }
}
```

```typescript
interface ShapeInterface {
  area(): number;
}

class Square implements ShapeInterface {
  public length: number;

  constructor(length: number) {
    this.length = length;
  }

  area(): number {
    return Math.pow(this.length, 2);
  }
}

class Circle implements ShapeInterface {
  public radius: number;

  constructor(radius: number) {
    this.radius = radius;
  }

  area(): number {
    return Math.PI * Math.pow(this.radius, 2);
  }
}

class AreaCalculator {

  constructor(private shapes: ShapeInterface[]) { }

  public sum(): number {
    let area: number = 0
    for (let shape of this.shapes) {
      area += shape.area();
    }

    return area
  }
}
```

DOS AND DON'TS

DON'T

- Modify existing code.
- Tightly couple modules or components to each other.
- Use inheritance in ways that violate the Liskov Substitution Principle.

DO

- Design modules and components that are open for extension
- Use inheritance, composition, and other techniques.
- Use interfaces and abstract classes to define the behavior of components

04

Liskov Substitution principle

“Objects of a superclass should be replaceable with objects of its subclasses without breaking the application.”

— Robert C. Martin

```typescript
interface Shape {
  area(): number;
}

class Rectangle implements Shape {
  private width: number;
  private height: number;

  constructor(width: number, height: number) {
    this.width = width;
    this.height = height;
  }

  public area(): number {
    return this.width * this.height;
  }

  public getWidth(): number {
    return this.width;
  }

  public getHeight(): number {
    return this.height;
  }

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

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

class Square extends Rectangle {
  private sideLength: number;

  constructor(sideLength: number) {
    super(sideLength, sideLength);
    this.sideLength = sideLength;
  }

  public getSideLength(): number {
    return this.sideLength;
  }

  public setSideLength(sideLength: number): void {
    this.sideLength = sideLength;
  }
}

```

```typescript
interface Shape {
  area(): number;
}

class Rectangle implements Shape {
  private width: number;
  private height: number;

  constructor(width: number, height: number) {
    this.width = width;
    this.height = height;
  }

  public area(): number {
    return this.width * this.height;
  }

  public getWidth(): number {
    return this.width;
  }

  public getHeight(): number {
    return this.height;
  }

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

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

class Square implements Shape {
  private sideLength: number;

  constructor(sideLength: number) {
    this.sideLength = sideLength;
  }

  public area(): number {
    return this.sideLength * this.sideLength;
  }

  public getSideLength(): number {
    return this.sideLength;
  }

  public setSideLength(sideLength: number): void {
    this.sideLength = sideLength;
  }
}

```

DOS AND DON'TS

DON'T

- Change the behavior of a method in a subtype
- Throw exceptions that are not declared in the supertype
- Add new methods to a subtype that are not declared in the supertype

DO

- Create subtypes that can be used interchangeably with their supertype
- Ensure that subclasses don't violate the contracts of their supertypes
- Create a clear and well-defined hierarchy of classes and interfaces

05

Interface segregation principle

“Clients should not be forced to depend on interfaces they do not use. This means that an interface should define only the methods that are relevant to the client that uses it.”

— **Robert C. Martin**

```typescript
interface Animal {
    walk(): void;
    run(): void;
    fly(): void;
}

class Dog implements Animal {
    walk() {
      console.log("Dog is walking.");
    }
    run() {
      console.log("Dog is running.");
    }
    fly() {
      throw new Error("Dog cannot fly.");
    }
}

class Bird implements Animal {
    walk() {
      console.log("Bird is walking.");
    }
    run() {
      console.log("Bird is running.");
    }
    fly() {
      console.log("Bird is flying.");
    }
}
```

```typescript
interface WalkingAnimal {
  walk(): void;
  run(): void;
}

interface FlyingAnimal {
  fly(): void;
}

class Dog implements WalkingAnimal {
  walk() {
    console.log("Dog is walking.");
  }
  run() {
    console.log("Dog is running.");
  }
}

class Bird implements WalkingAnimal, FlyingAnimal {
  walk() {
    console.log("Bird is walking.");
  }
  run() {
    console.log("Bird is running.");
  }
  fly() {
    console.log("Bird is flying.");
  }
}
```

DOS AND DON'TS

DON'T

- Define large, general-purpose interfaces that contain a large number of methods.
- Force a class to implement methods that it does not need or use.
- Create interfaces that are too specific or too tightly coupled to a particular implementation, making it difficult to swap out the implementation with another one.

DO

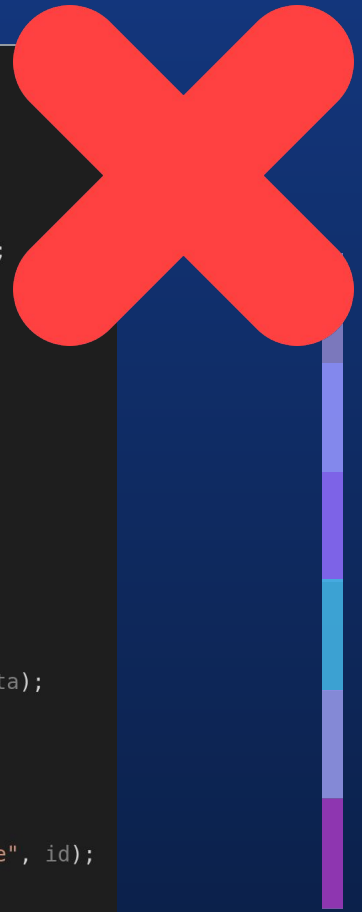
- Create interfaces that are specific to the needs of a particular class or module.
- Define small, cohesive interfaces that contain only the methods needed by the classes that implement them.
- Implement interfaces that contain more methods than needed only if you want to provide a common interface for multiple related classes.

06

Dependency inversion principle

“Entities must depend on abstractions, not on concretions. It states that the high-level module must not depend on the low-level module, but they should depend on abstractions.”

— **Robert C. Martin**



```ts
class UserService {
  private storage: DatabaseStorage;

  constructor() {
    this.storage = new DatabaseStorage();
  }

  saveUser(user: string): void {
    this.storage.save(user);
    return;
  }

  getUser(userId: string): string {
    return this.storage.get(userId);
  }
}

class DatabaseStorage {
  save(data: any): void {
    console.log("Saving to database", data);
    // Save data to a database
    return;
  }

  get(id: string): string {
    console.log("Retrieving from database", id);
    // Retrieve data from a database
    return "User data";
  }
}
```

```ts
interface Storage {
  save(data: any): void;
  get(id: string): string;
}

class UserService {
  private storage: Storage;

  constructor(storage: Storage) {
    this.storage = storage;
  }

  saveUser(user: any) {
    this.storage.save(user);
  }

  getUser(userId: string) {
    return this.storage.get(userId);
  }
}

class DatabaseStorage implements Storage {
  save(data: any): void {
    console.log("Saving to database", data);
    // Save data to a database
    return;
  }

  get(id: string): string {
    console.log("Retrieving from database", id);
    // Retrieve data from a database
    return "User data";
  }
}
```

DOS AND DON'TS

DON'T

- Hardcode dependencies.
- Use a specific implementation.
- Create circular dependencies.

DO

- Define interfaces for any external dependencies that your classes rely on.
- Use dependency injection to inject dependencies into classes.
- Make sure that high-level modules do not depend on low-level modules, and that both depend on abstractions.

REFERENCES

- https://www.digitalocean.com/community/conceptual-articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design
- https://docs.google.com/presentation/d/1SfOcc7SJ4FUz2nbD8gsBeGSQO0wDRsVrcLs4k0vz7dc/edit
- https://github.com/alu0101329888/Introduction-to-TypeScript
- https://www.typescripttutorial.net/
- https://www.typescriptlang.org/docs/handbook/2/classes.html

THANKS

Do you have
any questions?