S.O.L.I.D Principles

FEATURES OF THE TOPIC

Adrián Mora Rodríguez

adrian.mora.rodriguez.20@ull.edu.es

Diego Rodríguez Martín

diego.rodriguez.28@ull.edu.es

TABLE OF CONTENTS

01

Introduction

02

Single-Responsibility

03

Open-Close

04

Interface Segregation

05

Dependency Inversion

06

Liskov Substitution

INTRODUCTION

SOLID principles are a set of rules to follow in order to improve the development of class structures.

SOLID principles were first introduced by the famous computer scientist Robert C. Martin (also known as Uncle Bob).

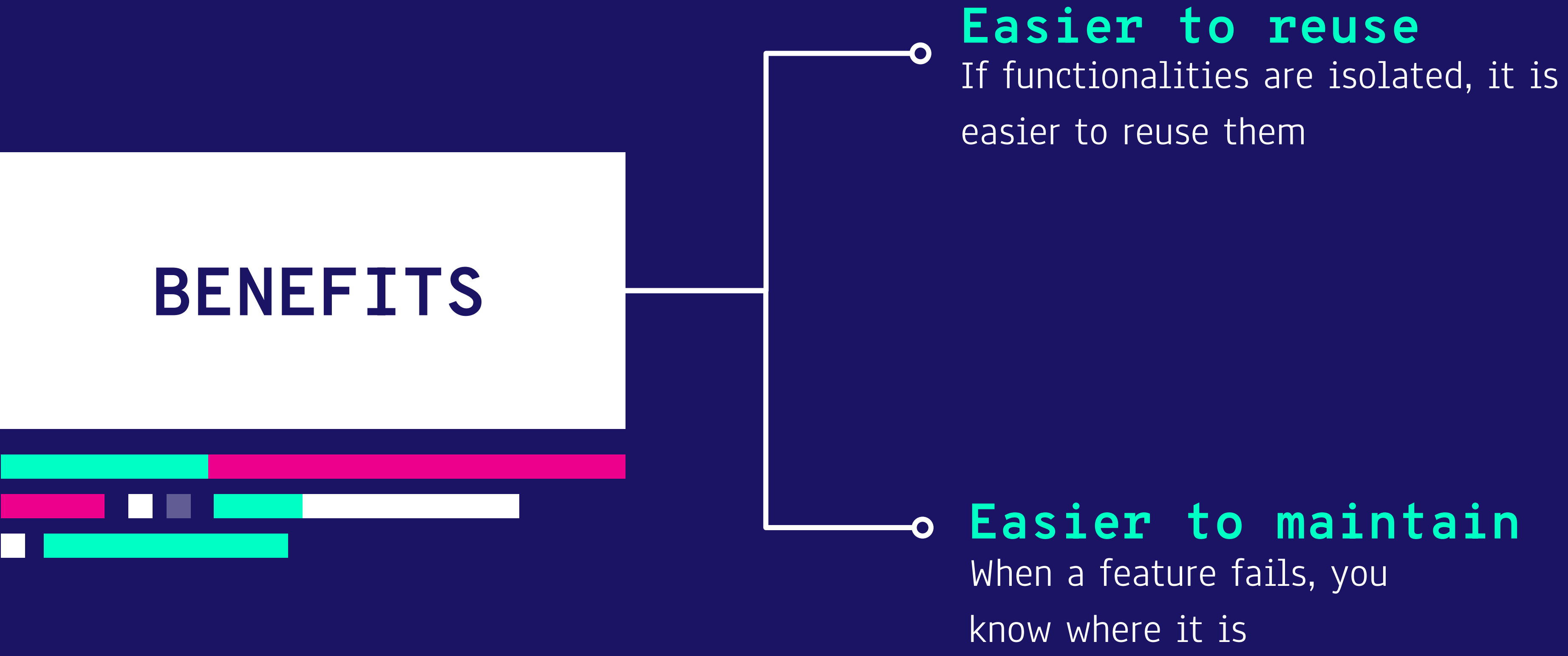
Although the acronym SOLID was later introduced by Michael Feathers.

> “The only way to go fast, is to go well.”
>
> –Robert C. Martin

SINGLE-RESPONSIBILITY

- One purpose per class
  - Specialized classes
  - Smaller classes
- Only one reason to change
  - Paradoxically, classes are more adaptable to change

WHY SHOULD WE APPLY THE SRP?



BENEFITS

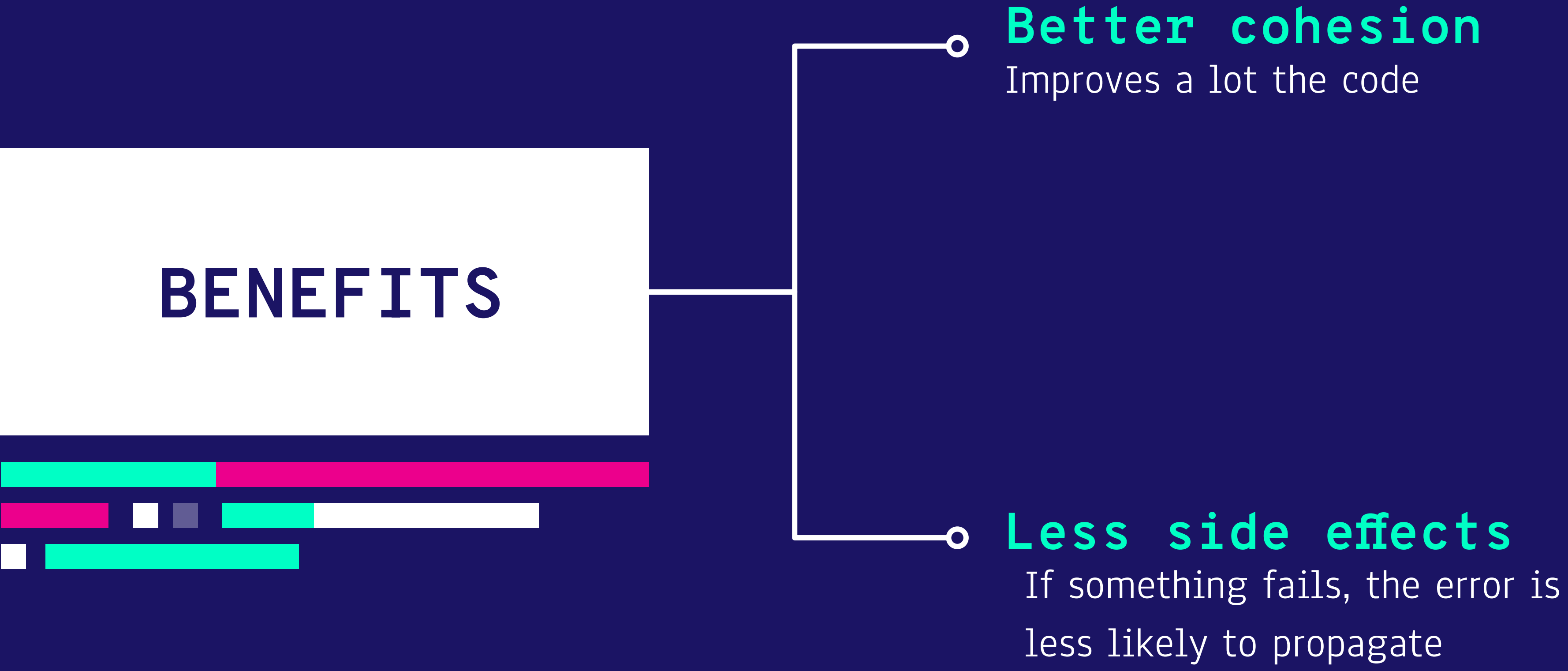
Easier to reuse

If functionalities are isolated, it is easier to reuse them

Easier to maintain

When a feature fails, you know where it is

WHY SHOULD WE APPLY THE SRP?



BENEFITS

Better cohesion

Improves a lot the code

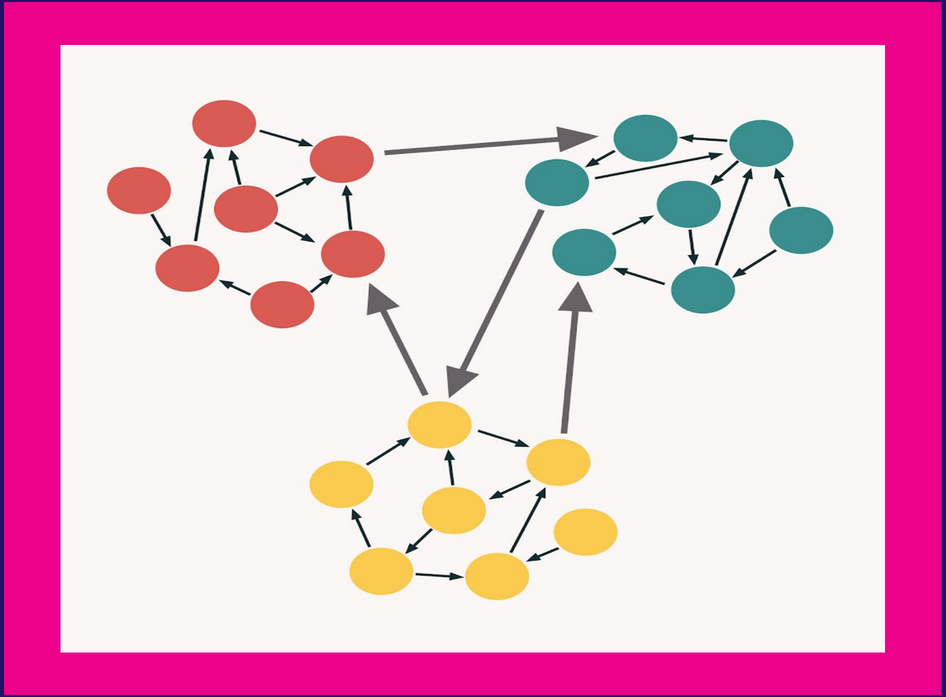
Less side effects

If something fails, the error is less likely to propagate

COHESION

- Cohesion is closely related to ensuring that the purpose for which a class is getting created in is well-focused and single.

- Cohesion high -> methods and variables are co-dependent and are a logical whole



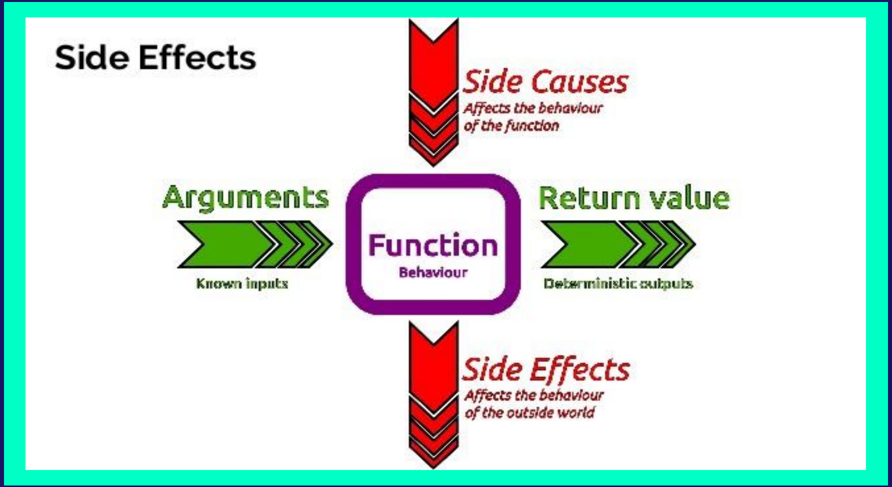
SIDE EFFECTS

- A piece of code that extends beyond its **primary purpose** may generate this effect

- This may not have **consequences** on the code block itself but **on the rest of the code block.**

- Here is were the **side effects** are generated

- These affects the behaviour of the rest of the code.



BAD EXAMPLE

```
class Book {
  private title: string;
  private author: string;
  private description: string;
  private pages: number;
  public saveToFile(fileName: string): void {
    // some fs.write method
  }
}
```

GOOD EXAMPLE

```
class Book {
  private title: string;
  private author: string;
  private description: string;
  private pages: number;
  // constructor and other methods
}

class Persistence {
  public saveToFile(book: Book): void {
    // some fs.write method
  }
}
```

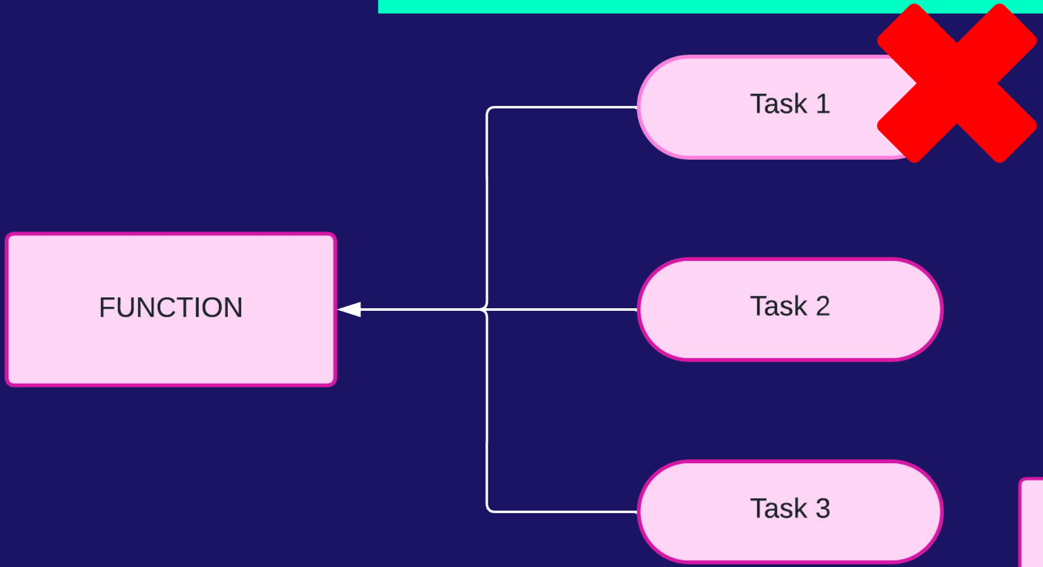
BAD EXAMPLE

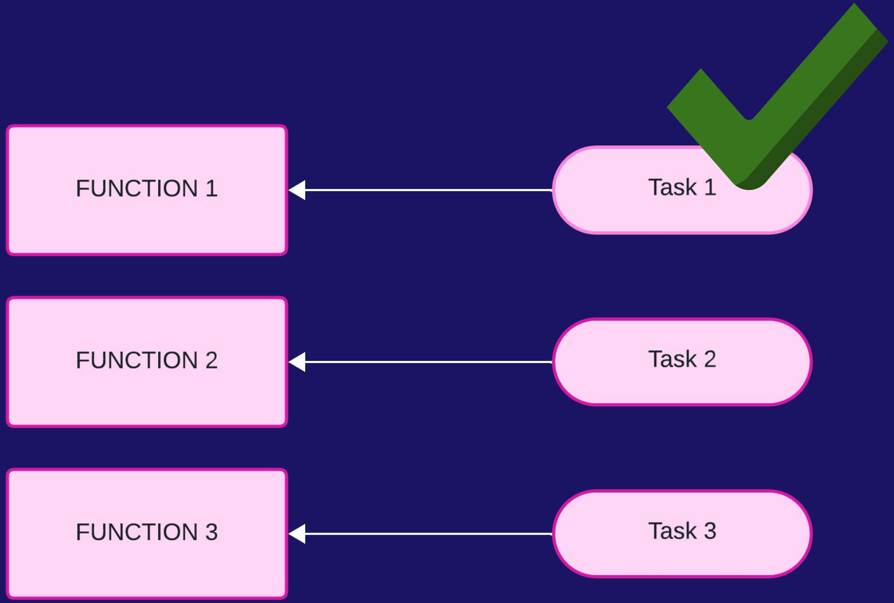
```
class FileManager {
  public read(file: string) {
    // Read file logic
  }
  public write(file: string, data: string) {
    // Write file logic
  }
  public compress(file: string) {
    // File compression logic
  }
  public encrypt(file: string) {
    // File encryption logic
  }
  // ...other methods for file operations
}
```

GOOD EXAMPLE

```
class MyFileReader {
  public read(file: string) {
    // Read file logic
  }
}
class FileWriter {
  public write(file: string, data: string) {
    // Write file logic
  }
}
class FileCompressor {
  public compress(file: string) {
    // File compression logic
  }
}
```

SRP SUMMARY


Task 1
Task 2
Task 3
FUNCTION


FUNCTION 1
Task 1
FUNCTION 2
Task 2
FUNCTION 3
Task 3

OPEN-CLOSE

- Classes should be open for extension but closed for modification.
    - Add new functionality without changing the existing code
    - Polymorphism

WHY SHOULD WE APPLY THE OCP?

BENEFITS

Reduces the risk of new errors

Thanks to the minimization of the modifications.

Promotes the extension

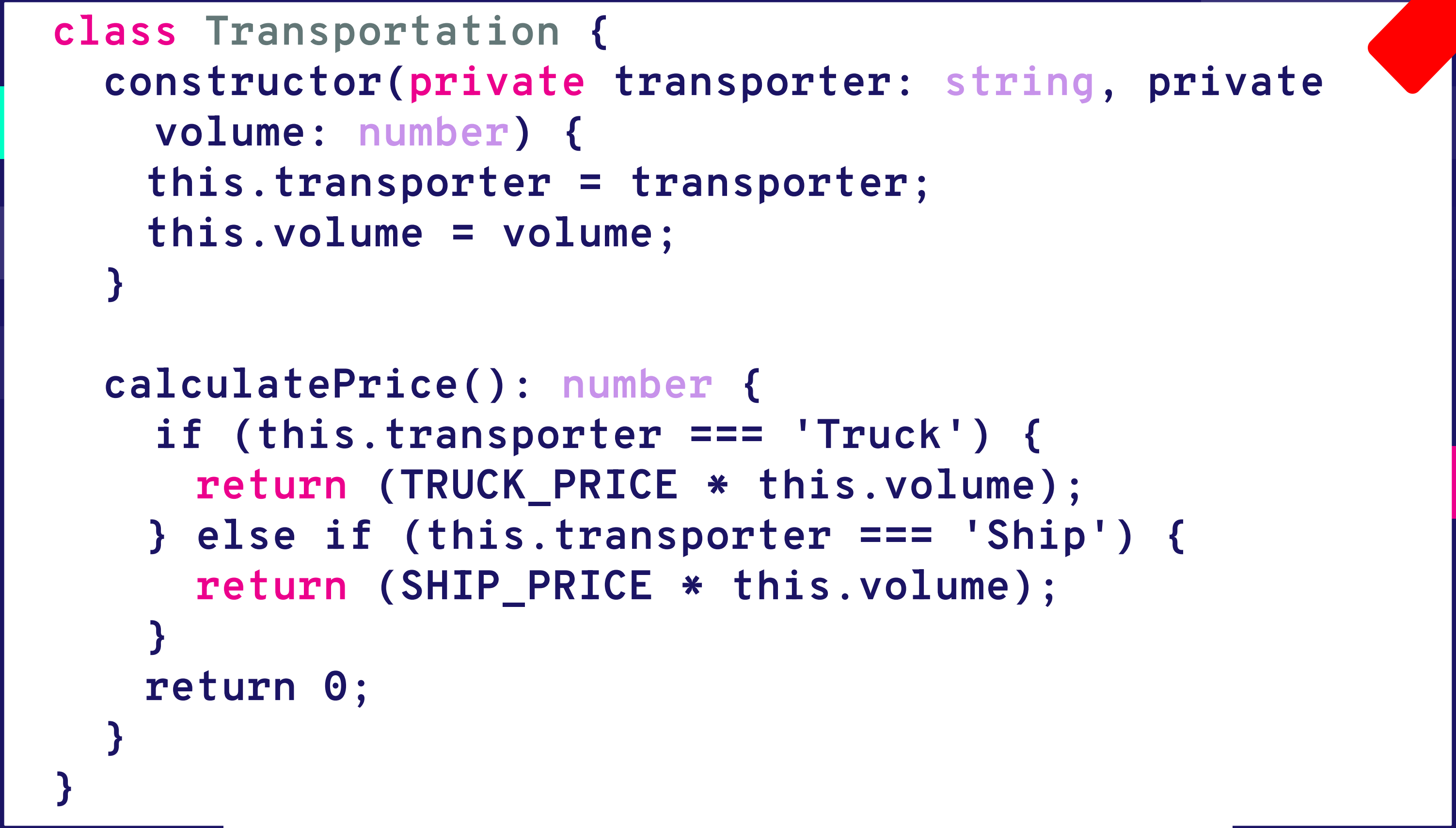
With the use of design patterns like strategy pattern

MODULARITY AND SCALABILITY

- Divide your into:
  - Smaller
  - Independent
  - Cohesive

The ability to handle increasing amounts of work or to be readily enlarged

BAD EXAMPLE

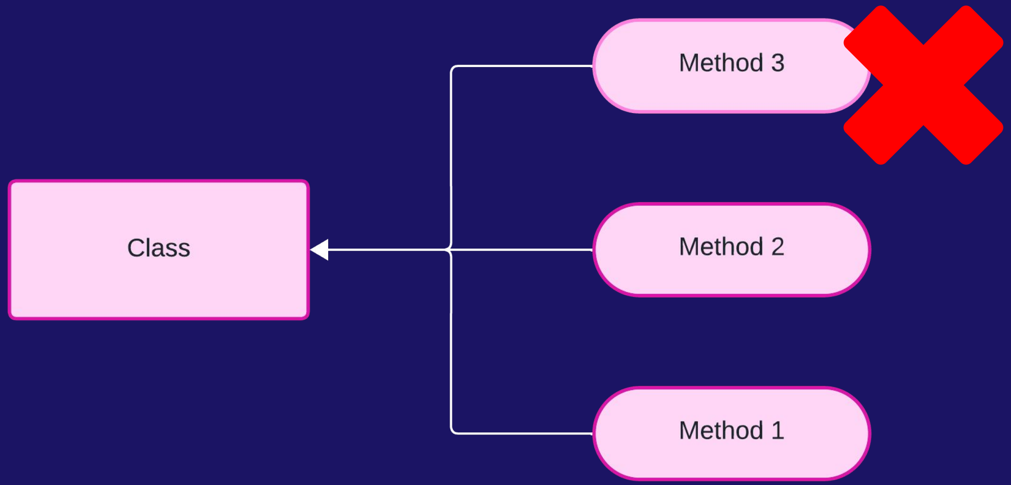


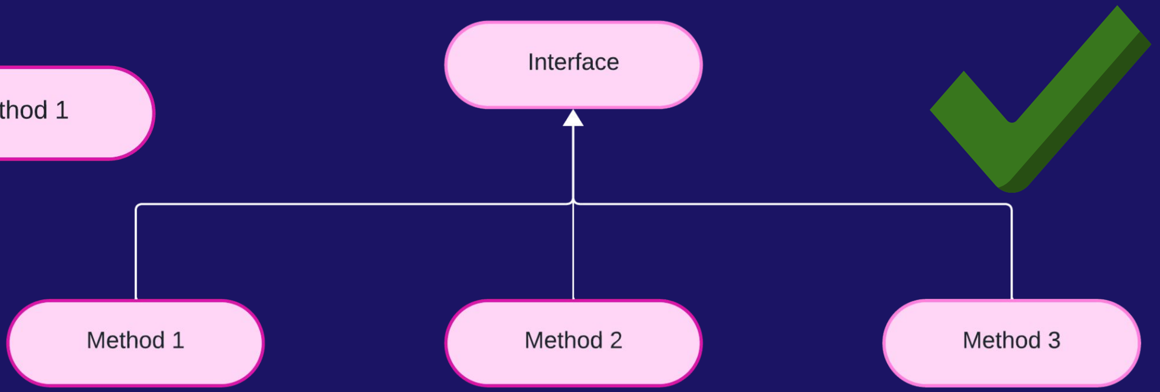
```
class Transportation {
  constructor(private transporter: string, private
    volume: number) {
    this.transporter = transporter;
    this.volume = volume;
  }

  calculatePrice(): number {
    if (this.transporter === 'Truck') {
      return (TRUCK_PRICE * this.volume);
    } else if (this.transporter === 'Ship') {
      return (SHIP_PRICE * this.volume);
    }
    return 0;
  }
}
```

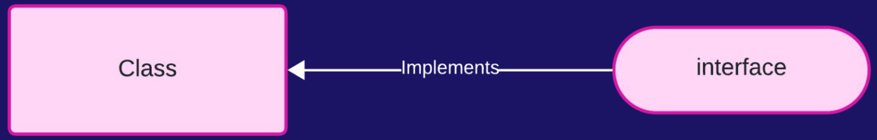
GOOD EXAMPLE

```
interface Transporter {
  type: string;
  calculatePrice(): number;
}

class Ship implements Transporter {
  private type: string
  private shipPrice: number = 300;
  constructor() { this.type = 'Ship' }
  public calculatePrice() {
    return shipPrice;
  }
}
```

OPC SUMMARY


Method 3
Class
Method 2
Method 1


Interface
Method 1
Method 2
Method 3


Class
Implements
interface

Interface Segregation

- Is better to have many specific interfaces than too few and general ones.
  - More interfaces -> less methods in each one. ✓
  - Few interfaces -> non-used methods may be implemented ✗

ADVANTAGES OF TYPESCRIPT TO APPLY ISP

Interface
explicit structure

Allows implementing more than 1 interface -> Multiple inheritance

Not virtual methods

WHY SHOULD WE APPLY THE ISP?

BENEFITS

Improves cohesion and modularity

Because the interfaces are smaller and more specific.

Easier implementation of interfaces by classes

Requires only the implementation of the relevant methods for each class.

BAD EXAMPLE

```
interface Character {
  shoot(): void;
  swim(): void;
  talk(): void;
  dance(): void;
}
class Troll implements Character {
  public shoot(): void {
    // a troll can shoot, poorly, but can
  }
  public swim(): void {
    // a troll can't swim
  }
  . . .
}
```

GOOD EXAMPLE

```
interface Shooter {
 shoot(): void;
}
interface Swimmer {
 swim(): void;
}
interface Dancer {
 dance(): void;
}

class Troll implements Shooter, Dancer {
  public shoot(): void
  public dance(): void
}
```

BAD EXAMPLE

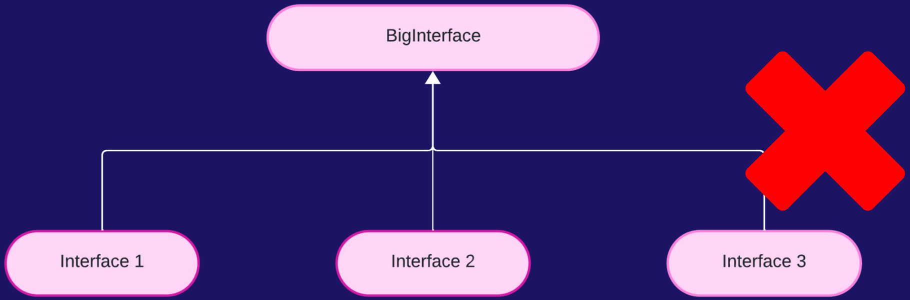
✖

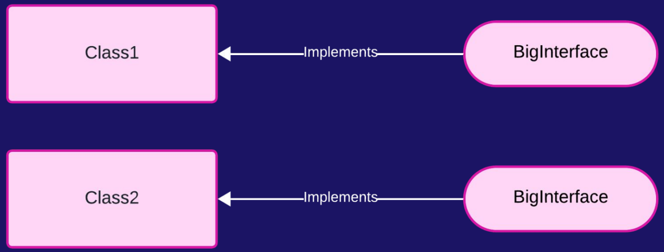
```
interface VehicleInterface {
  drive(): string;
  fly(): string;
}

class Car implements VehicleInterface {  ✖
  public drive() : string;
  public fly() : string;
}

class Airplane implements VehicleInterface {  ✖
  public drive() : string;
  public fly() : string;
}
```

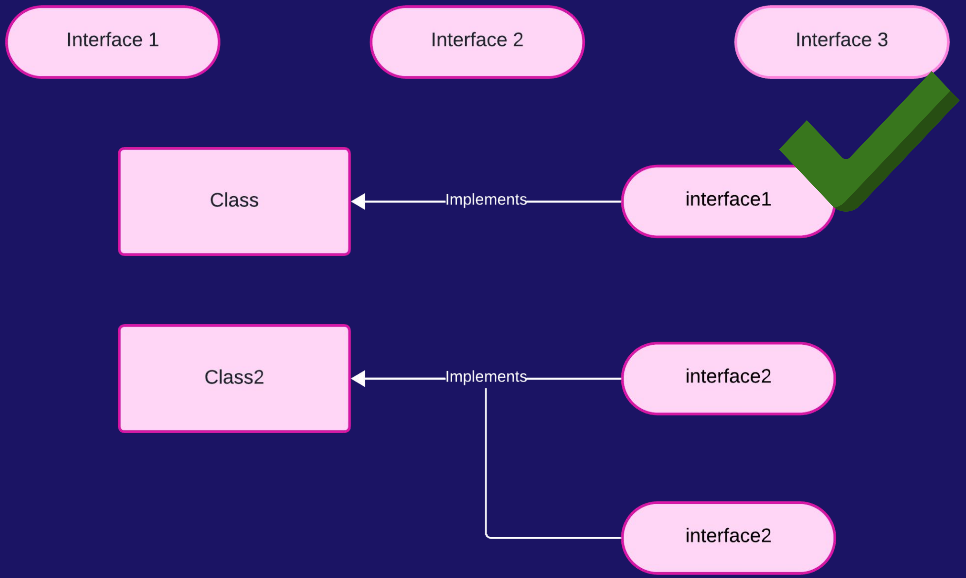
GOOD EXAMPLE

```
interface CarInterface {
  drive(): string;
}
interface Airplane Interface {
  fly(): string;
}
class Car implements CarInterface {
  public drive() : string;
}
class Airplane implements AirplaneInterface {
  public fly() : string;
}
class FutureCar implements CarInterface,AirplaneInterface{
  public drive() : string;
  public fly() : string;
}
```

ISP SUMMARY





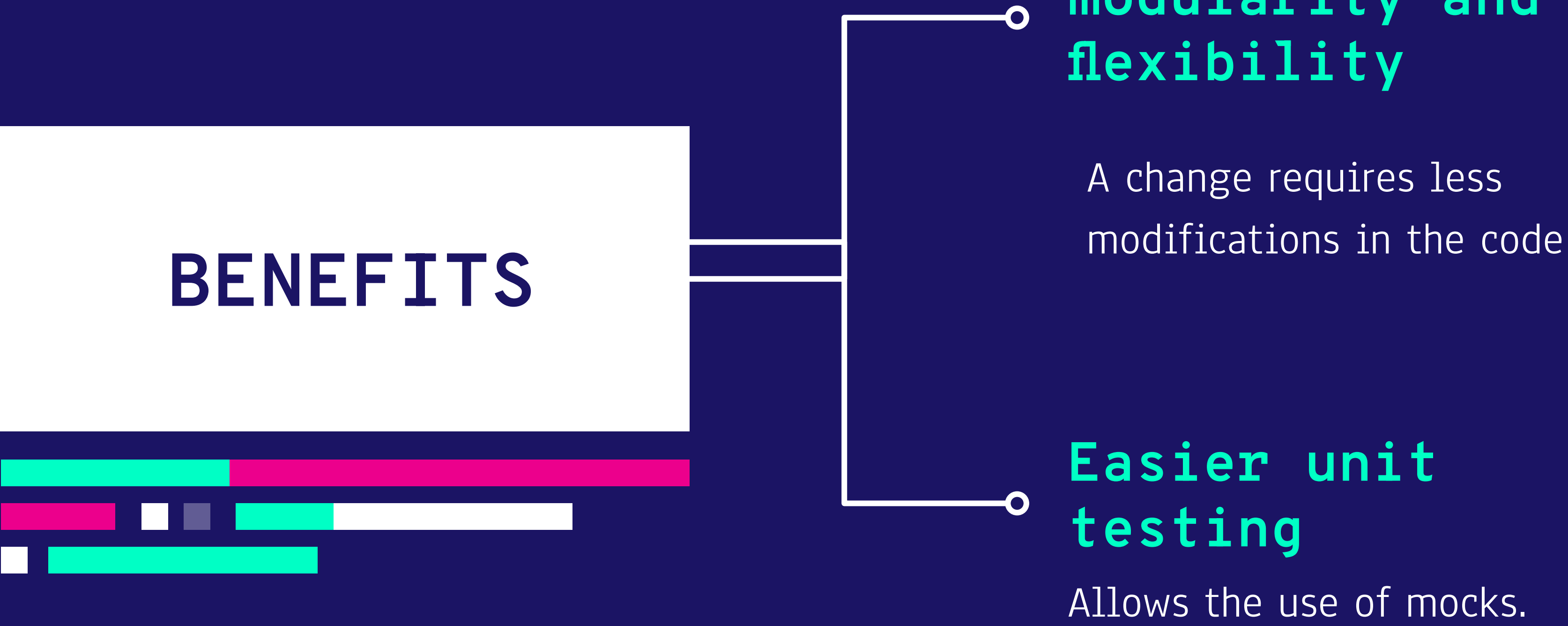


- YAGNI
- Don't make God classes

Dependency Inversion

- Implement classes and modules that depend of abstractions
  - Details should depend on the abstractions
  - Not the other way around

WHY SHOULD WE APPLY THE DIP?



BENEFITS

Better modularity and flexibility

A change requires less modifications in the code

Easier unit testing

Allows the use of mocks.

BAD EXAMPLE

```
class FrontendDeveloper {
  public writeHtmlCode(): void;
}
class BackendDeveloper {
  public writeTypeScriptCode(): void;
}
class SoftwareProject {
  public frontendDeveloper: FrontendDeveloper;
  public backendDeveloper: BackendDeveloper;
}
```

GOOD EXAMPLE

```
interface Developer {
  develop(): void;
}
class FrontendDeveloper implements Developer {
  public develop(): void {this.writeHtmlCode();}
  private writeHtmlCode(): void;
}
class BackendDeveloper implements Developer {
  public develop(): void {this.writeTypeScriptCode();}
  private writeTypeScriptCode(): void;
}
class SoftwareProject {
  public developers: Developer[];
}
```

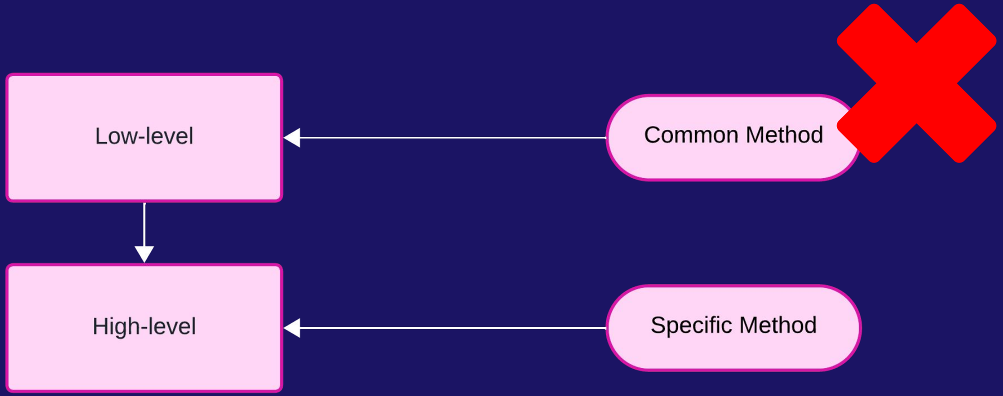
BAD EXAMPLE

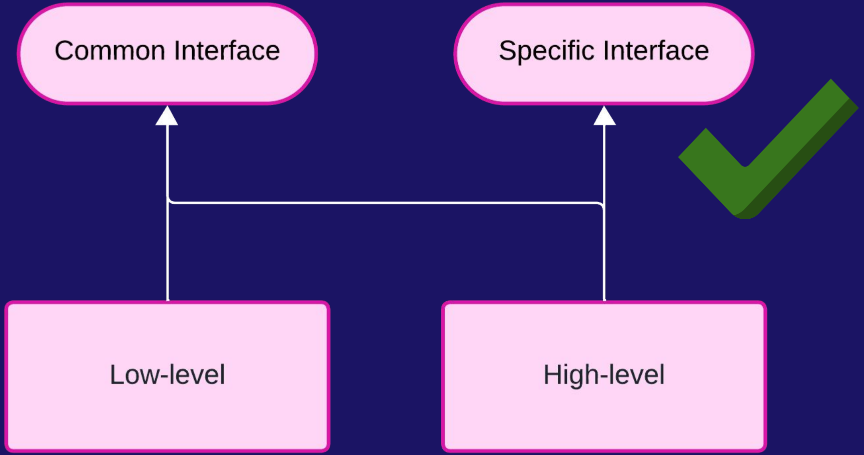
```
class MultiplyInstruction {
  public multiply(firstOperand: number, secondOperand:
number): number;
}

class AddInstruction {
  public add(firstOperand: number, secondOperand:
number): number;
}

class ArithmeticLogicUnit {
  private multiplyInstruction: MultiplyInstruction;
  private addInstruction: AddInstruction;
}
```

GOOD EXAMPLE

```
interface Instruction {
  execute(firstOperand: number, secondOperand: number):
number;
}
class MultiplyInstruction implements Instruction {
  public execute(firstOperand: number, secondOperand:
number): number {
    this.multiply(a, b);
  }
  public multiply(firstOperand: number, secondOperand:
number): number;
}
```

DIP SUMMARY

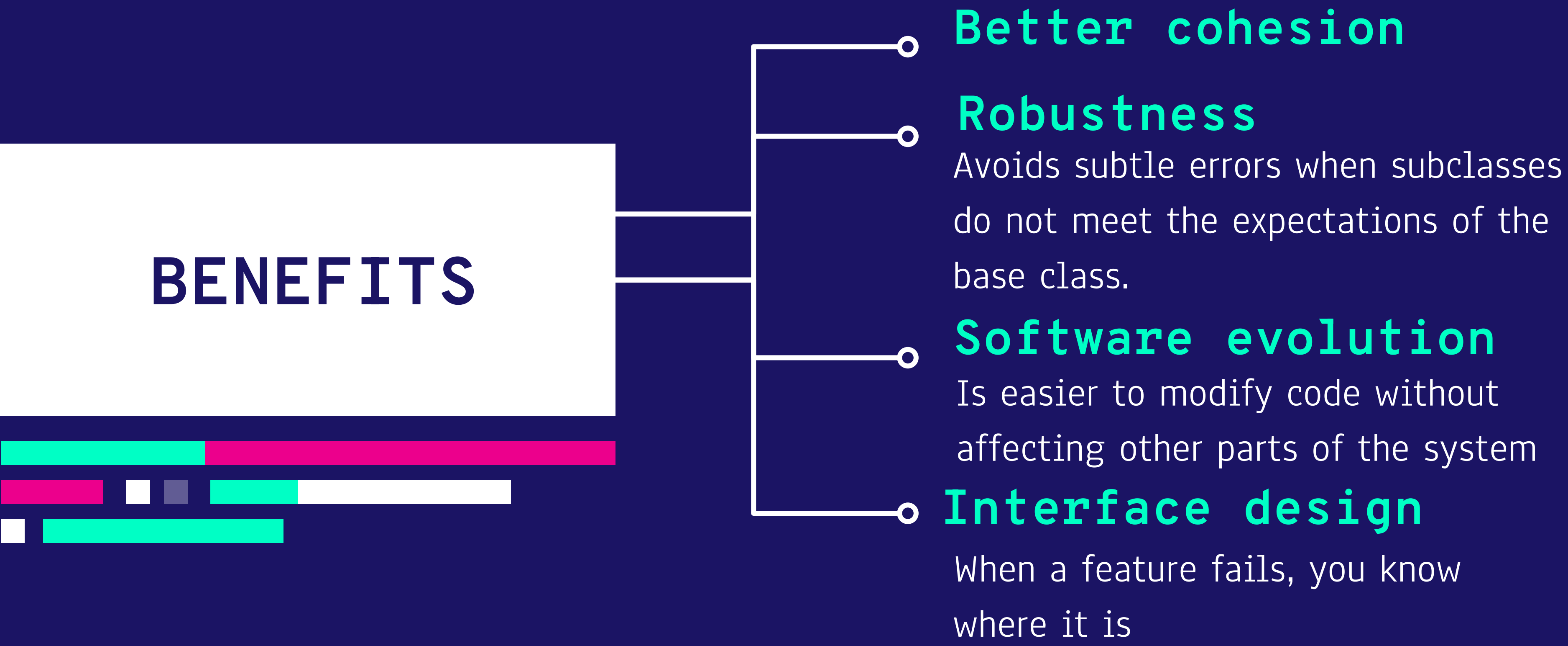

Low-level
Common Method
High-level
Specific Method


Common Interface
Specific Interface
Low-level
High-level

Liskov Substitution

- Objects must be replaceable by instances of their subtypes.
- Changing the type should not affect the behavior of the program.

WHY SHOULD WE APPLY THE LSP?



BENEFITS

Better cohesion

Robustness

Avoids subtle errors when subclasses do not meet the expectations of the base class.

Software evolution

Is easier to modify code without affecting other parts of the system

Interface design

When a feature fails, you know where it is

BAD EXAMPLE

```
class Rectangle {
  constructor(width: number, length: number) {}
  public setWidth(width: number) {this.width = width;}
  public setLength(length: number) {
    this.length = length;
  }
  public getArea() {
    return this.width * this.length;
  }
}
```

BAD EXAMPLE

```
class Square extends Rectangle {
  constructor(side: number) {super(side, side);}
  public setWidth(width: number) {
    super.setWidth(width);
    super.setLength(width);
  }
  public setLength(length: number) {
    super.setWidth(length);
    super.setLength(length);
  }
}
```

GOOD EXAMPLE

```
interface Shape {getArea: () => number;}

class Rectangle implements Shape {
  constructor(width: number,length: number) {
    this.width = width;
    this.length = length;
  }
  public getArea(): number {
    return this.width * this.length;
  }
}
```

GOOD EXAMPLE

```
class Square implements Shape {
  constructor(private sizeOfSides: number) {
    this.sizeOfSides = sizeOfSides
  }
  public getArea(): number {
    return this.sizeOfSides * this.sizeOfSides;
  }
}
```

BAD EXAMPLE

```
interface Vehicle {
  startEngine(): void;
}
class Car implements Vehicle {
  public startEngine(): void {
    console.log("Starting the car engine...");
    // Código para arrancar el motor del coche
  }
}
class Motorcycle implements Vehicle {
  public startEngine(): void {
    console.log("Starting the motorcycle engine...");
    // Código para arrancar el motor de la motocicleta
  }
}
```

BAD EXAMPLE

```
class Bicycle implements Vehicle {
  public startEngine(): void {
    console.log("Bicycles don't have engines to start.");
    // Las bicicletas no tienen motor que arrancar
  }
}

function startAllVehicles(vehicles: Vehicle[]): void {
  vehicles.forEach(vehicle => {
    vehicle.startEngine();
  });
}
```

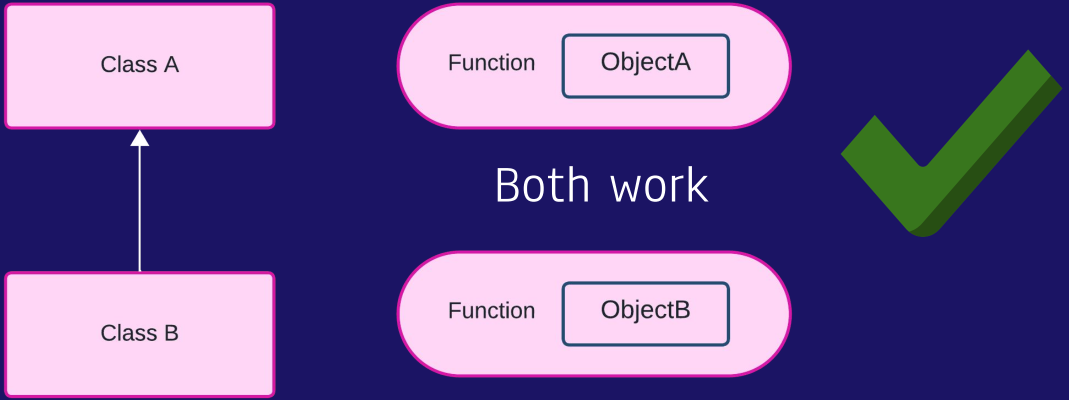
GOOD EXAMPLE

```
interface MotorVehicle {
  startEngine(): void;
}
class Car extends MotorVehicle {
  public startEngine(): void {
    console.log("Starting the car engine...");
    // Código para arrancar el motor del coche
  }
}
class Motorcycle extends MotorVehicle {
  public startEngine(): void {
    console.log("Starting the motorcycle engine...");
    // Código para arrancar el motor de la motocicleta
  }
}
```

GOOD EXAMPLE

```
interface VehicleWithoutMotor {
  ride(): void;
}

class Bicycle extends VehicleWithoutMotor {
  public ride(): void {
    console.log(“Riding a bicycle”);
  }
}

function startAllVehicles(vehicles: MotorVehicle[]):
void {
  vehicles.forEach(vehicle => {
    vehicle.startEngine();
  });
}
```

LSP SUMMARY



Class A

Class B

Function ObjectA

Both work

Function ObjectB

ARE ALWAYS GOOD?

- In most cases: yes
- **The programmer's judgment must always come first**
- If SOLID complicates the understanding of the code, do not follow it to the letter.

> “Rules are for the guidance of wise men and the obedience of fools”
>
> –Douglas Bader

“REAL LIFE” EXAMPLE

- GitHub Repository of this presentation

A FUNNY REFERENCE

- The SOLID Principles, Explained with Motivational Posters : Global Nerdy

RESOURCES

- SOLID Principles Series: Understanding the Single Responsibility Principle (SRP) in Node.js with TypeScript - DEV Community
- 10 OOP Design Principles Every Programmer Should Know | HackerNoon
- SOLID Principles in TypeScript (2022) | Bits and Pieces
- SOLID Principles with Javascript Examples | by Hayreddin Tüzel | Medium
- JavaScript — Principios SOLID. Temario | by Mauricio Garcia
- Chapter 10: Classes - Clean Code
- The SOLID Principles, Explained with Motivational Posters : Global Nerdy

THANKS!

Do you have any questions?

adrian.mora.rodriguez.20@ull.edu.es
diego.rodriguez.28@ull.edu.es

CREDITS: This presentation template was created by Slidesgo.