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 Liskov Substitution

05 Interface Segregation

06 Dependency Inversion

INTRODUCTION

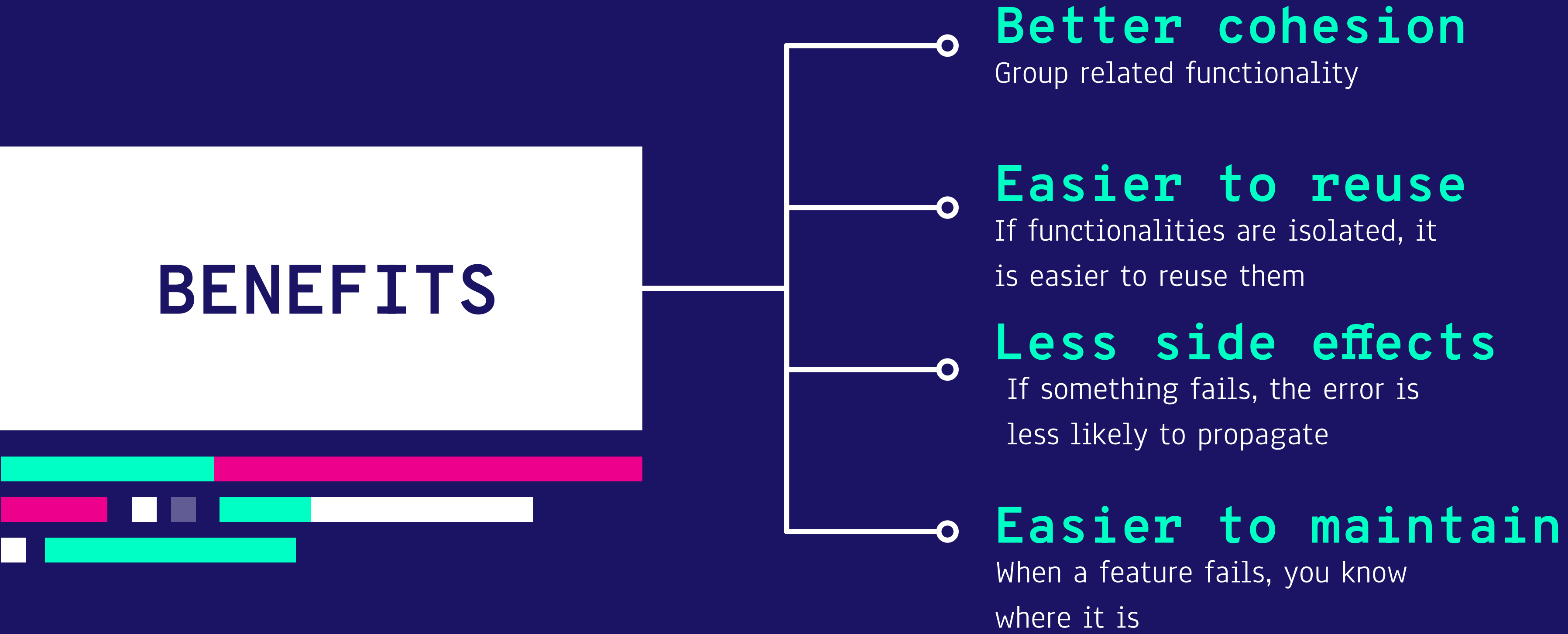
SOLID principles are a set of rules to follow in order to improve the development of class structures. SOLID principles were first introduced by the famous computer scientist Robert J. Martin (also known as Uncle Bob). Although the acronym SOLID was later introduced by Michael Feathers.

> “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

Better cohesion

Group related functionality

Easier to reuse

If functionalities are isolated, it is easier to reuse them

Less side effects

If something fails, the error is less likely to propagate

Easier to maintain

When a feature fails, you know where it is

BAD EXAMPLE

```
class Book {

  saveToFile(fileName: string): void {

    // some fs.write method
  }

  private title: string;

  private author: string;

  private description: string;

  private pages: number;
}
```

GOOD EXAMPLE

```
class Book {
  // constructor and other methods

  private title: string;
  private author: string;
  private description: string;
  private pages: number;
}

class Persistence {

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

BAD EXAMPLE

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

GOOD EXAMPLE

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

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.

Better modularity and scalability

Promotes the extension

With the use of design patterns like strategy pattern

BAD EXAMPLE

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

  calculatePrice(): number {
    if (this.transporter == 'Truck') {
      return (500 * this.volume);
    } else if (this.transporter == 'Ship') {
      return (300 * this.volume);
    }
    return 0;
  }
}
```

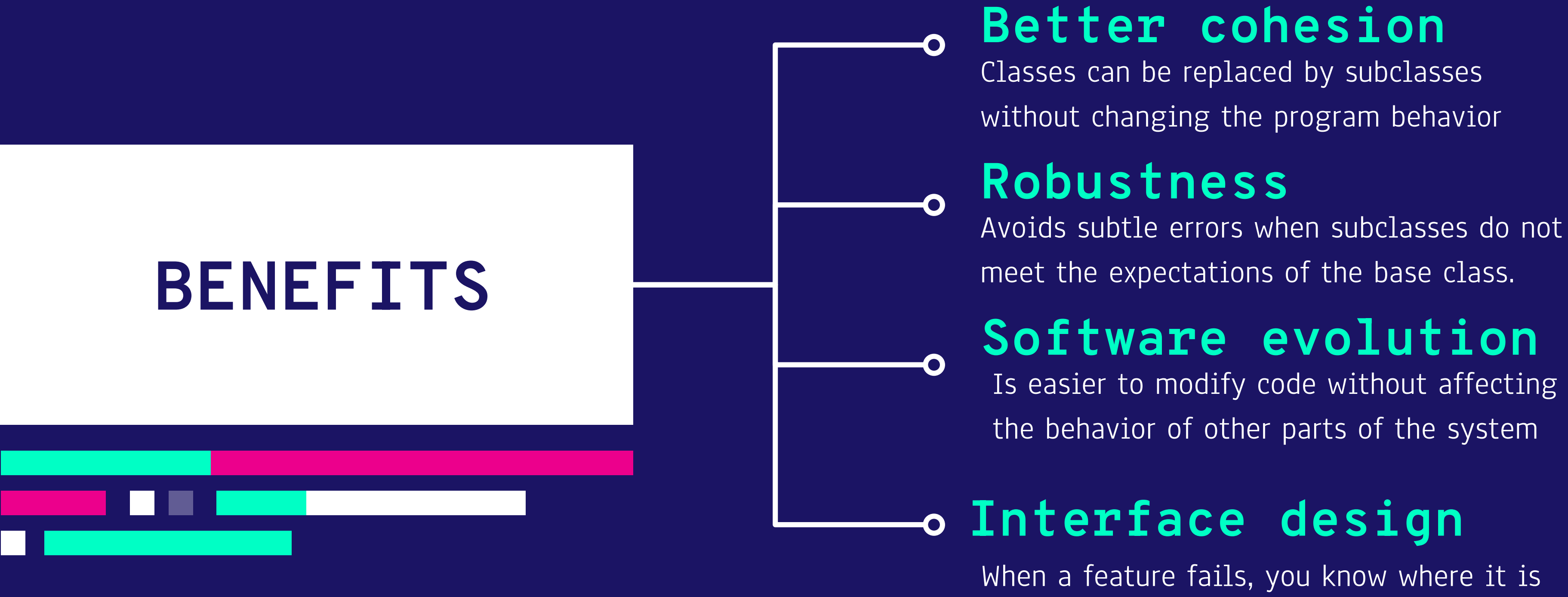
GOOD EXAMPLE

```
interface Transporter {
  type: string;

  calculatePrice(): number;
}

class Ship implements Transporter {
  public type: string

  constructor() { this.type = 'Ship' }

  calculatePrice() {
    return 300;
  }
}
```

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

Classes can be replaced by subclasses without changing the program behavior

Robustness

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

Software evolution

Is easier to modify code without affecting the behavior of 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;}
  getArea(): number {return this.width *
this.length;}
}
```

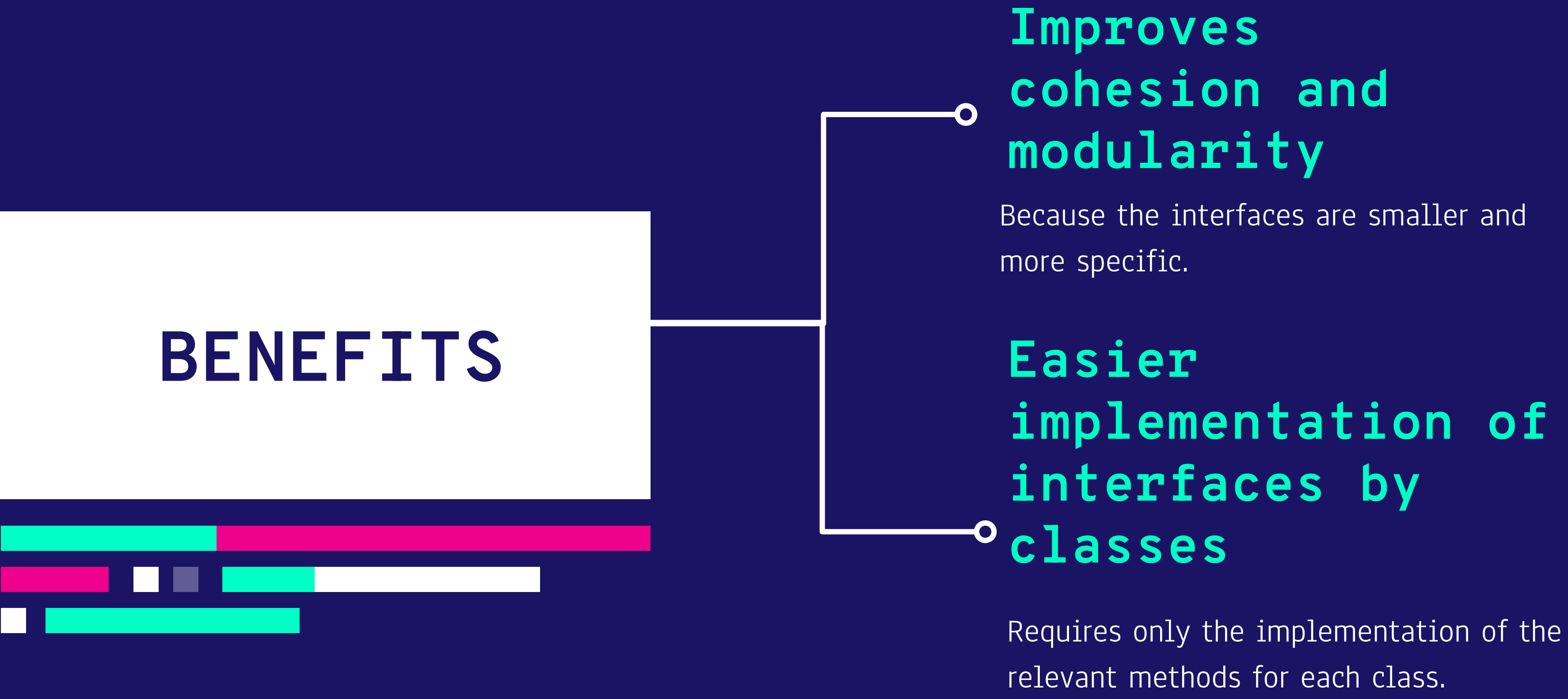
GOOD EXAMPLE

```
class Square implements Shape {

  constructor(private sizeOfSides: number) {

   this.sizeOfSides = sizeOfSides

  }

  getArea(): number {

   return this.sizeOfSides * this.sizeOfSides;

  }

}
```

Interface Segregation Principle

- 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 ✗

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;
}
```

BAD EXAMPLE

```
class Troll implements Character {
  public shoot(): void {
  }
  public swim(): void {
   // a troll can't swim
  }
 . . .
}
```

GOOD EXAMPLE

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

GOOD EXAMPLE ✓

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

  public dance(): void {
  }
}
```

BAD EXAMPLE

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

BAD EXAMPLE ✖

```
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 AirplaneInterface {
  fly(): string;
}
```

GOOD EXAMPLE

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

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;
}
```

GOOD EXAMPLE

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

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.

“REAL LIFE” EXAMPLE

- https://github.com/ULL-ESIT-PAI-2023-2024/2023-2024-pai-solid-principles-2023-2024-pai-solid-adrianmr-diegorm.git

RESOURCES

- https://dev.to/ruben_alapont/solid-principles-series-understanding-the-single-responsibility-principle-srp-in-nodejs-with-typescript-57e8
- https://hackernoon.com/10-oop-design-principles-every-programmer-should-know-f187436caf65
- https://blog.bitsrc.io/solid-principles-in-typescript-153e6923ffdb
- https://medium.com/@hayreddintuzel/solid-principles-with-examples-12f36f61796c
- https://mauriciogc.medium.com/javascript-principios-solid-e93a17e950bb

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, incluiding icons by Flaticon, and infographics & images by Freepik.