SOLID Principles *and UML*

Authors

Fabián González Lence

Contact:
fabian.gonzalez.29

Diego Hernández Chico

Contact:
diego.hernandez.22

Samuel Frías Hernández

Contact:
samuel.frias.40

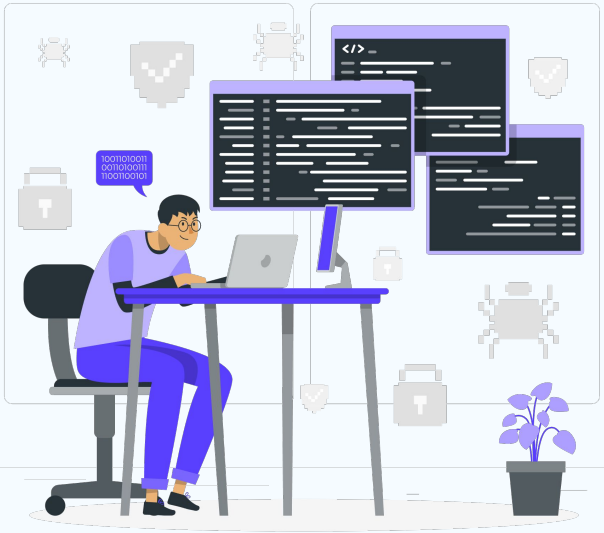
Table of contents

01

Introduction to SOLID Principles

Flexible, maintainable, and scalable code

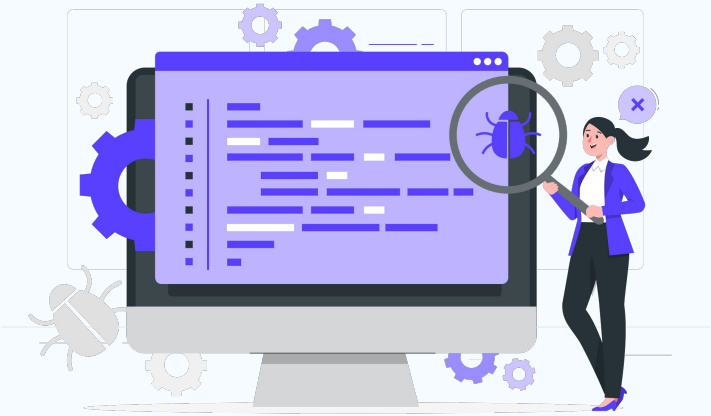
Why SOLID?



- Maintainable and scalable code.
- Easier modifications and extensions.
- Improved readability and reusability.
- Reduced complexity.
- Prevents “spaghetti code”.

What does SOLID mean?

S stands for: **Single Responsibility**
O stands for: **Open/Closed**
L stands for: **Liskov Substitution**
I stands for: **Interface Segregation**
D stands for: **Dependency Inversion**



02

Single Responsibility Principle (SRP)

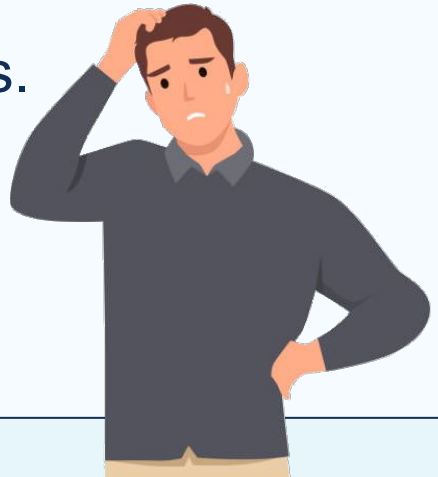
It is not possible to cover everything

“A class should have only one reason to change”

Robert C. Martin

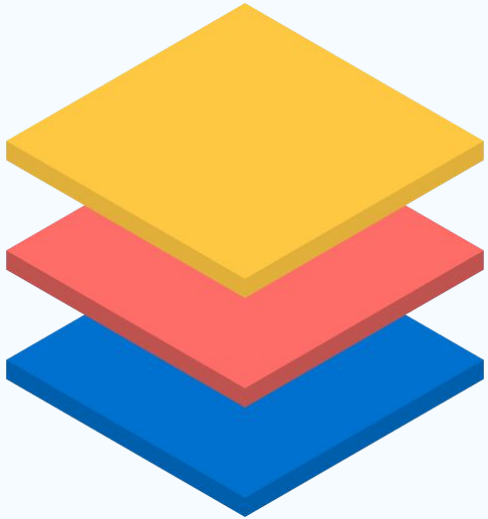
What means Single Responsibility?

- A class shouldn’t change without a good reason.
  - Also applicable to functions.
  - Changes in the business logic.
  - Affects different actors or teams.
  - Etc.
- Still a bit confusing, right?



When should we start suspecting? 🔍

- A class is responsible of more than one software layer.
  - Presentation.
  - Business logic.
  - Persistence.



- Closely related to MVC

When should we start suspecting?

bad_layer_implementation.ts

```ts
class MatrixHandler {
  private result: number[][]

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

  public multiply(matrixA: number[][], matrixB: number[][]) {
    ...
  }

  public display() {
    ...
  }
}
```

When should we start suspecting? 

good_layer_implementation.ts

```ts
class MatrixOperations {
  multiply(matrixA: number[][], matrixB: number[][]): number[][] {
    ...
  }
}

class MatrixView {
  display(matrix: number[][]) {
    ...
  }
}
```

When should we start suspecting?

- The number of public methods:
  - Lots of them.
  - Not related to each other.
- The antithesis of GOD classes
- You should refactor.

When should we start suspecting? 🔍

- Each group of methods uses an specific property and ignores the rest.

- You could probably split that class.
  - Two new simplified classes.
  - Interacting with each other.

bad_class_separation.ts

```ts
class DataManager {
  private userData: any;
  private systemData: any;

  constructor(userData: any, systemData: any) { ... }

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

  updateUserEmail(newEmail: string): void {
    this.userData.email = newEmail;
  }

  getSystemStatus(): string {
    return this.systemData.status;
  }

  uptdateSystemConfig(newConfig: any): void {
    this.systemData.config = newConfig;
  }
}
```

When should we start suspecting?

good_class_separation.ts

```ts
class UserDataManager {
  private userData: any;

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

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

  updateUserEmail(newEmail: string): void {
    this.userData.email = newEmail;
  }

  calculateUserScore(): number {
    return this.userData.age * 2;
  }
}
```

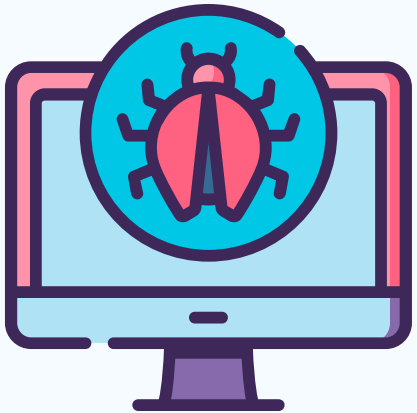
When should we start suspecting?

good_class_separation.ts

```ts
class SystemDataManager {
  private systemData: any;

  constructor(systemData: any) {
    this.systemData = systemData;
  }

  getSystemStatus(): string {
    return this.systemData.status;
  }

  updateSystemConfig(newConfig: any): void {
    this.systemData.config = newConfig;
  }
}
```

When should we start suspecting?

- It is difficult to make tests.
  - Developing unit test is demanding
  - Tests do not have enough granularity.

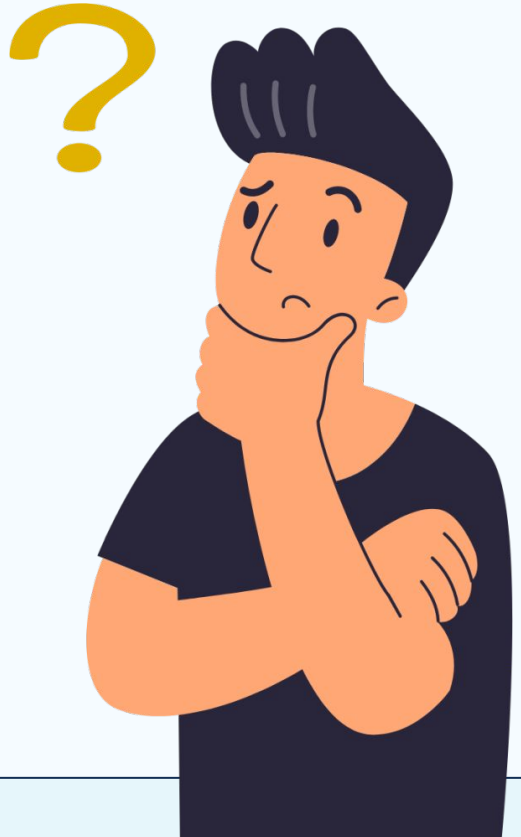


When should we start suspecting? 🔎

- Class changes each time that we add a functionality.
  - Every time that we expand the program.

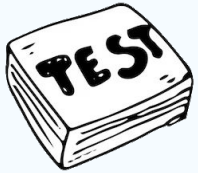
Importance of SRP

Is it really as relevant as people think?

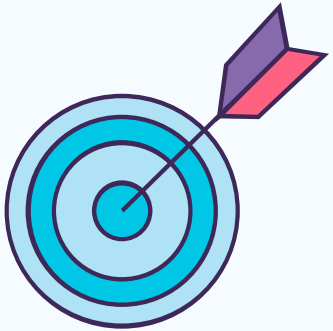


Importance of SRP

- Improves the code in:
  - Scalability
  - Maintainability
  - Reusability
  - And… much more!



Importance of SRP



- Lots of facilities to make tests
  - Well defined responsibilities.
  - More specific tests.
  - Easier to catch possible mistakes.

Importance of SRP

- It is really helpful for team developments
  - Separated responsibilities.
  - Allows parallel development.

03

Open–Closed Principle (OCP)

Software entities should be **open for extension** but **closed for modification**.

Open–Closed Principle (OCP)

Open for extension:

- New behavior can be added to a module without modifying its source code.
- Achieved through mechanisms like inheritance, interfaces, or abstract classes.

Open–Closed Principle (OCP)

Closed for modification:

- Once a module is working correctly, its source code should not be changed.
- Prevents introducing new bugs or breaking existing functionality.

Theoretical Example

Scenario:

- A Calculator class performs operations (e.g., addition, subtraction).
- We want to add multiplication without modifying the Calculator class.

Theoretical Example

Solution:

- Use an interface (Operation) to define operations.
- Each operation implements Operation.
- Calculator works with any operation that implements Operation.

Practical Example

Step 1: Define the Abstraction.

operation-interface.ts

```ts
interface Operation {
    calculate(a: number, b: number): number;
}
```

Practical Example

Step 2: Implement the Details.

operation-interface.ts

```ts
class Sum implements Operation {
    calculate(a: number, b: number): number {}
}

class Subtract implements Operation {
    calculate(a: number, b: number): number {}
}

class Multiply implements Operation {
    calculate(a: number, b: number): number {}
}
```

Practical Example

Step 3: High-Level Module.

TS operation-interface.ts

```
class Calculator {
    private operations: Operation[];

    constructor(operations: Operation[]) {}
}
```

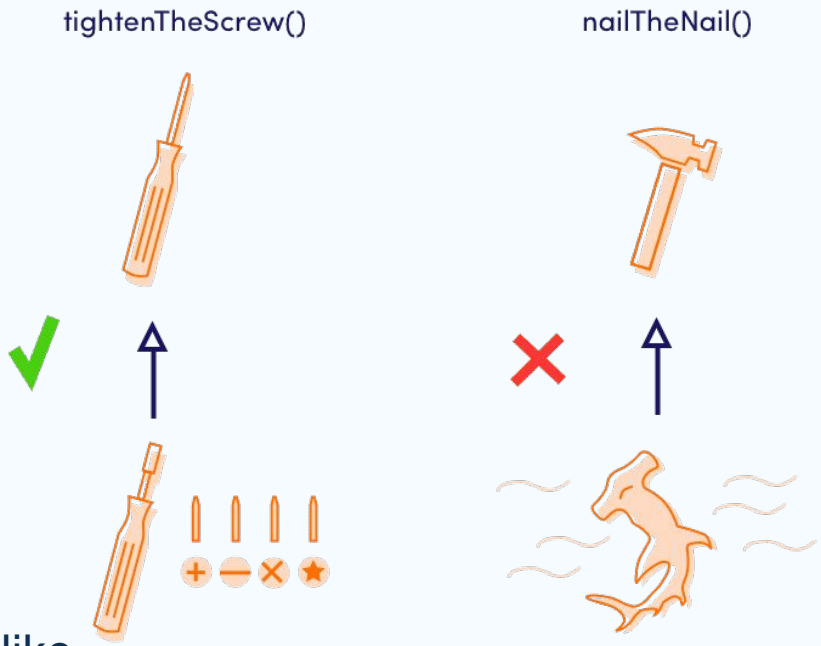
Benefits of OCP

1. **Maintainability**: No need to modify existing code.
2. **Scalability**: Easy to add new features.
3. **Reusability**: Abstractions promote code reuse.
4. **Testability**: Existing functionality doesn’t need to be retested.

04

Liskov Substitution Principle (LSP)

If it looks like a duck, it should quack like one too to call it a duck!



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

Robert C. Martin

What’s the meaning of that?

ClassA-ClassB.ts

```ts
class A {
  public print() {
    console.log("I'm a method from A");
  }
}

class B extends A {
  public print() {
    console.log("I'm a method from B");
  }
}
```

ClassA-ClassB.ts

```ts
function callPrint(instanceOfA: A) {
  instanceOfA.print();
}

let exampleA: A = new A();
callPrint(exampleA);
// I'm a method from A

let exampleB: A = new B();
callPrint(exampleB);
// I'm a method from B
```

LSP Definition

- Introduced by Barbara Liskov (1987)
  - Let Φ(x) be a property about objects x of type T.
  - Then Φ(y) should be true for objects y of type S where S is a subtype of T.

LSP Example

incorrect-example.ts

```ts
class Rectangle {
  protected width: number;
  protected height: number;

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

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

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

incorrect-example.ts

```ts
class Square extends Rectangle {
  public setWidth(width: number) {
    this.width = width;
    this.height = width;
  }

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

LSP Example

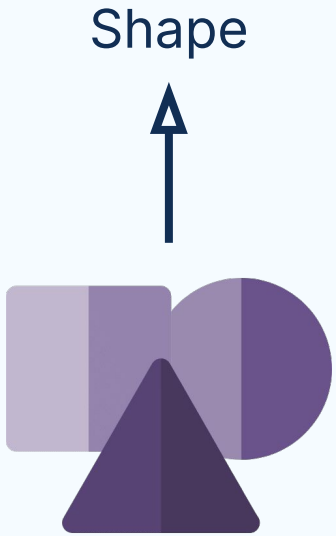
incorrect-example.ts

```ts
let rectangle = new Rectangle();
rectangle.setWidth(100);
rectangle.setHeight(50);
console.log(rectangle.getArea()); // 5000

let square = new Square();
square.setWidth(100);
square.setHeight(50);
console.log(square.getArea()); // 2500 (?)
```

Example Correction

solution-example.ts

```ts
abstract class Shape {
  abstract getArea(): number;
}
```



Example Correction

solution-example.ts

```ts
class Rectangle extends Shape {
  protected width: number;
  protected height: number;

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

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

solution-example.ts

```ts
class Square extends Shape {
  private side: number;

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

  public setSide(side: number) {
    this.side = side;
  }

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

Example Correction

solution-example.ts

```ts
let rectangle: Shape = new Rectangle(100, 50);
console.log(rectangle.getArea()); // 5000

let square: Shape = new Square(100);
console.log(square.getArea()); // 10000
```

Another Example

incorrect-vehicle.ts

```ts
class Vehicle {
  protected fuel: number;

  constructor() {
    this.fuel = 100;
  }

  public drive(): void {
    if (this.fuel <= 0) {
      console.log('Without fuel.');
      return;
    }
    this.fuel -= 10;
    console.log('Driving...');
  }

  public refuel(): void {
    this.fuel = 100;
    console.log('Refueled!');
  }
}
```

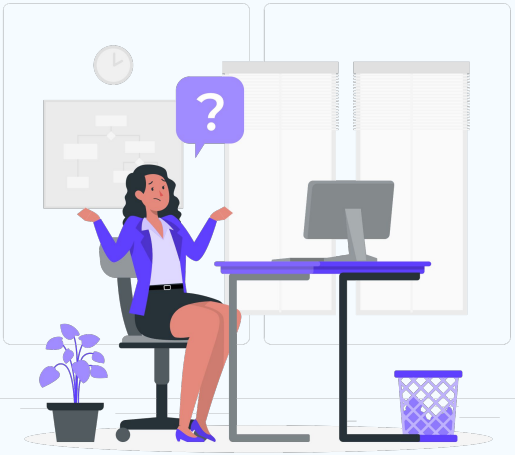
incorrect-vehicle.ts

```ts
class ElectricCar extends Vehicle {
  constructor() {
    super();
  }

  public refuel(): void {
    throw new Error("Electric cars don't refuel.");
  }
}
```

Another Example

incorrect-vehicle.ts

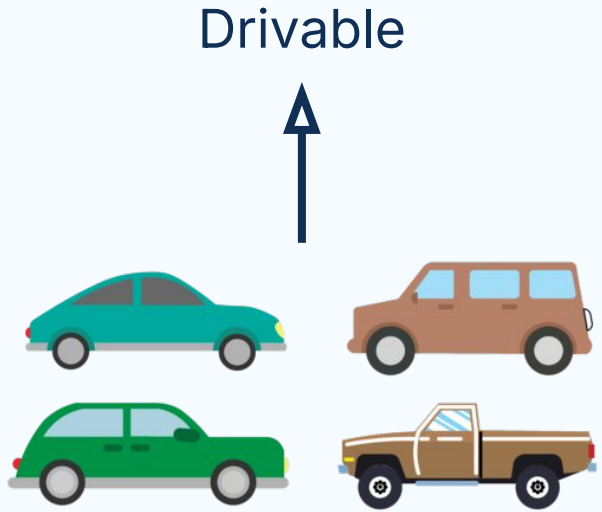
```ts
function testVehicle(vehicle: Vehicle) {
  vehicle.drive();
  vehicle.refuel(); // Error if vehicle is a ElectricCar
}

const myCar = new Vehicle();
testVehicle(myCar); // OK

const myTesla = new ElectricCar();
testVehicle(myTesla); // Error!
```



Another Solution Example

solution-vehicle.ts

```ts
abstract class Drivable {
  abstract drive(): void;
}
```



Another Solution Example

solution-vehicle.ts

```ts
class FuelCar extends Drivable {
  protected fuel: number;

  constructor() {
    super();
    this.fuel = 100;
  }

  public drive(): void {
    ...
    this.fuel -= 10;
    console.log('Driving...');
  }

  public refuel(): void {
    this.fuel = 100;
    console.log("Refueled!");
  }
}
```

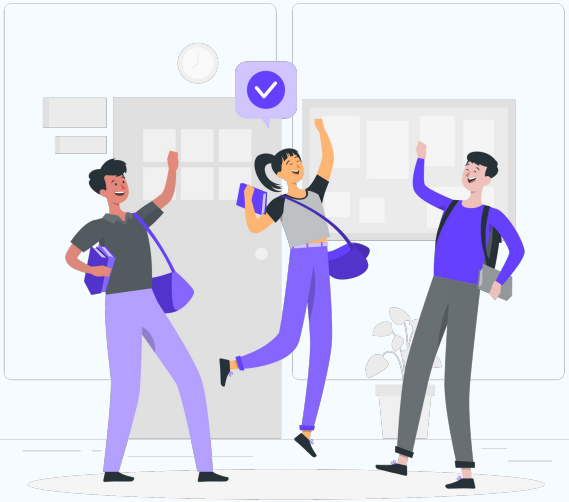
solution-vehicle.ts

```ts
class ElectricCar extends Drivable {
  protected battery: number;

  constructor() {
    super();
    this.battery = 100;
  }

  public drive(): void {
    ...
    this.battery -= 10;
    console.log('Driving...');
  }

  public recharge(): void {
    this.battery = 100;
    console.log('Recharged!');
  }
}
```

Another Solution Example

solution-vehicle.ts

```ts
// Now testVehicle works with any Drivable object
function testVehicle(vehicle: Drivable) {
  vehicle.drive();
}

const myCar = new FuelCar();
testVehicle(myCar); // Works correctly

const myTesla = new ElectricCar();
testVehicle(myTesla); // Also works correctly
```



Benefits of LSP

1. **Code Reusability:** Subclasses can be used in place of their parent without problems.

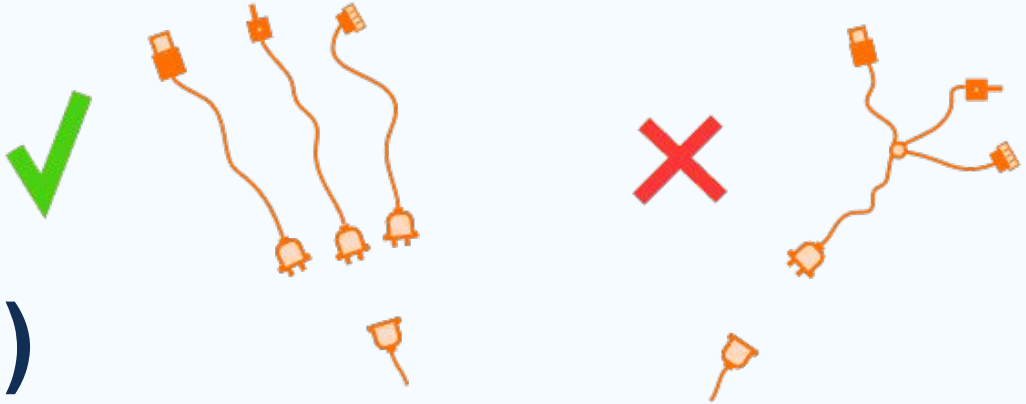
2. **Facilitates UT:** Tests for the base class work for subclasses.

3. **Prevents Unexpected Side Effects:** Derived classes don’t introduce side effects breaking functionality.

4. **Robust Inheritance:** Prevents the misuse of inheritance reducing inconsistencies.

05

Interface Segregation Principle (ISP)



You shouldn’t implement methods that you won’t use

“This principle advises software designers to avoid depending on things that they don’t use.”

Robert C. Martin

Example of ISP

- Library users
  - Two different interfaces:
  - Regular members.
  - Librarian.
  - Each kind of user worries only for what they need.

Example of ISP

library.ts

```ts
interface LibraryUser {
  searchBook(title: string): void;
  reserveBook(bookId: number): void;
}

class Member implements LibraryUser {
  searchBook(title: string) {
    console.log('Searching book: ' + title);
  }

  reserveBook(bookId: number) {
    console.log('Reserving book with ID:' + bookId);
  }
}
```

Example of ISP

library.ts

```ts
interface Librarian extends LibraryUser {
  addBook(book: unknown): void;
  grantLoan(userId: number, bookId: number): void;
}
```

Example of ISP

library.ts

```ts
class LibraryManager implements Librarian {
  searchBook(title: string) {
    ...
  }
  reserveBook(bookId: number) {
    ...
  }
  addBook(book: unknown) {
    ...
  }
  grantLoan(userId: number, bookId: number) {
    ...
  }
}
```

Good practices to implement the ISP

Following these steps will be really useful.

Good practices to implement the ISP

- Smaller is often better
  - Don’t implement “GOD” interfaces.
  - It is preferable to implement smaller interfaces.
  - Specific interfaces will be more affordable.

Good practices to implement the ISP

- Prioritize composition over inheritance
  - A bad use of inheritance may end up forcing you to define unnecessary methods.
  - Compositions allows more flexible classes.

❌ Good practices to implement the ISP

Birds example

```ts
class BaseBird {
  eat() {
    console.log("The bird is eating.");
  }

  fly() {
    console.log("The bird is flying.");
  }
}

class InheritingOstrich extends BaseBird {
  fly() {
    // Ostriches don't fly, but they are forced to implement this method
    throw new Error("Ostriches cannot fly.");
  }
}
```

✓ Good practices to implement the ISP

TS Birds example

```ts
class Animal {
  constructor() {  }

  eat() {
    console.log("is eating.");
  }
}

class FlyingBird {
  fly() {
    console.log("The bird is flying.");
  }
}
```

✓ Good practices to implement the ISP

TS Birds example

```ts
class Sparrow extends Animal {
  constructor(private flyer: FlyingBird) { super() }

  eat() {
    console.log("The sparrow ");
    super.eat();
  }

  fly() {
    this.flyer.fly();
  }
}
```

✓ Good practices to implement the ISP

TS Birds example

```
class Ostrich extends Animal {
  eat() {
    console.log("The ostrich");
    super.eat();
  }
}
```

Good practices to implement the ISP

- Do not put all possible methods in one interface “just in case.”
  - Don’t put effort in things that you won’t ever use.
  - This practice will overload the complexity of your code.

Good practices to implement the ISP

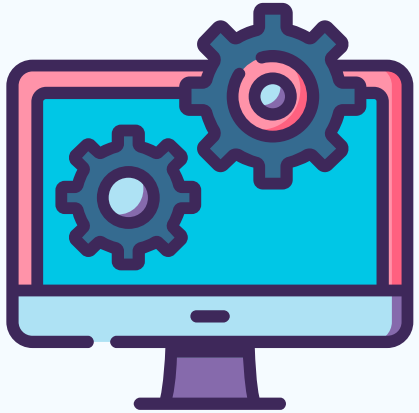
- Use abstract classes if necessary.
  - Many classes will use the same methods.
  - Abstract classes avoid repeating code.

Reasons to follow the ISP

Which benefits does it bring to us?

Reasons to follow the ISP

- Promotes the independence of the modules.
  - Reduces dependency between classes.
  - Benefits reusability in different contexts.



Reasons to follow the ISP

- Enhances flexibility.
  - Allows code to evolve independently.
  - Isolation of specific interfaces.

Reasons to follow the ISP

- Improves Code Readability.
  - Interfaces are easier to understand and implement.

- Facilitates Maintenance:
  - Reduces the likelihood of side effects when interfaces change.

06

Dependency inversion principle (DIP)

You must depend upon abstractions, not upon implementations.

Dependency Inversion Principle (DIP)

Key Idea:

1. High-level modules should not depend on low-level modules. Both should depend on abstractions.

2. Abstractions should not depend on details. Details should depend on abstractions.

Dependency Inversion Principle (DIP)

High-level modules should not depend on low-level modules. Both should depend on abstractions.

- High-level modules: Contain application logic.
- Low-level modules: Handle specific details.
- Abstractions: Interfaces or abstract classes.

Dependency Inversion Principle (DIP)

Abstractions should not depend on details.
Details should depend on abstractions.

- Abstractions: Define what should be done.
- Details: Define how it is done.
- Goal: Implementations depend on abstractions.

Theoretical Example

Scenario:

- A NotificationService sends notifications to users.
- Initially, it depends directly on EmailService.
- Problem: Adding a new notification method requires modifying NotificationService.

Theoretical Example

Solution:

- Introduce an abstraction (MessageService).
- NotificationService depends on MessageService.
- EmailService and SmsService implement MessageService.

Practical Example

Step 1: Define the Abstraction.

message-service-interface.ts

```ts
interface MessageService {
    sendMessage(message: string, recipient: string): void;
}
```

Practical Example

Step 2: Implement the Details.

message-service-interface.ts

```ts
class EmailService implements MessageService {
    sendMessage(message: string, recipient: string): void {}
}

class SmsService implements MessageService {
    sendMessage(message: string, recipient: string): void {}
}
```

Practical Example

Step 3: High-Level Module.

TS message-service-interface.ts

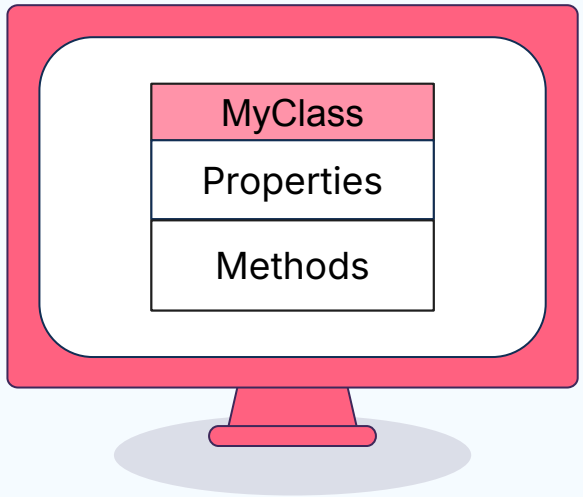
```
class NotificationService {
    private messageService: MessageService;

    notifyUser(message: string, recipient: string): void {
        this.messageService.sendMessage(message, recipient);
    }
}
```

Benefits of DIP

1. **Decoupling**: High-level modules are independent of low-level modules.
2. **Flexibility**: Easy to add new implementations without modifying existing code.
3. **Testability**: Mock dependencies in unit tests.
4. **Maintainability**: Code is easier to understand and modify.

07

Class diagrams with UML.



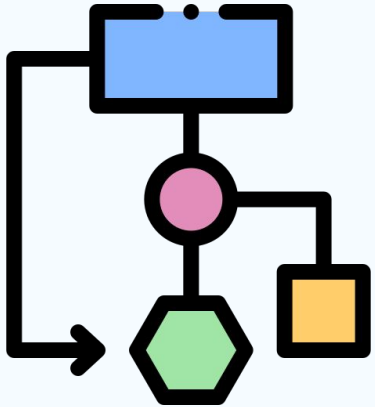
Visually represent the architecture, design, and implementation of complex software systems.

What is UML?

- Unified Modeling Language
- Standardized modeling language.
  - Integrated set of diagrams.
  - Helpful to specify, build and document.
  - Graphic notation diagrams.

What is UML?

- Born in the 90’s
- Specially useful for OOP.
  - Communication diagram.
  - Activity diagram.
  - Sequence diagram.
  - …

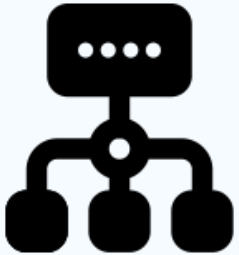


What is UML?

- We will focus on Class diagrams
  - Model the different classes of a system
  - Properties.
  - Methods.
  - Operations.
  - Relations.

Why UML?

- Widely used in work environments.
  - In a few years you will jump into it.
- Provides a visual interface.
- Support for high-level development concepts

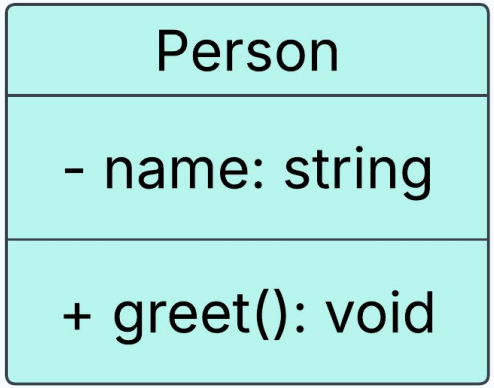


Class Representation

- A class is represented by a rectangle divided into three sections:
  - Class Name: Top section.
  - Attributes: Middle section.
  - Methods: Bottom section.

Class Representation

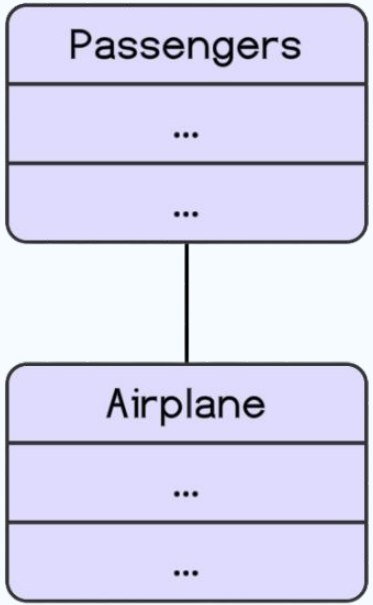
- Visibility:
  - `+` for public.
  - `–` for private.
  - `#` for protected.



- Data types: Specified for attributes and methods.

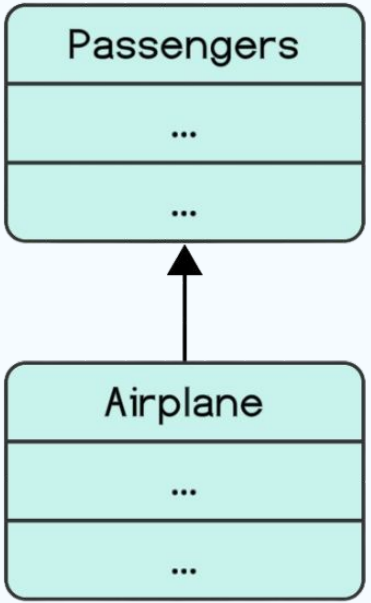
Association

- A basic relationship between two classes.
- A solid line connecting two classes.



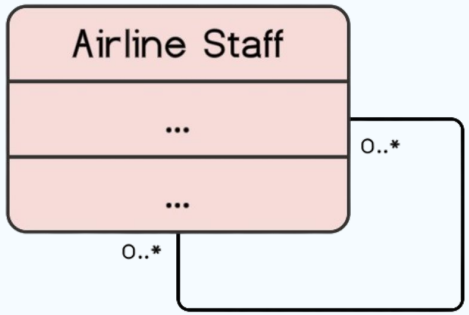
Directed Association

- A directional relationship between two classes.
- An airplane carries passengers but **not the other way around**.



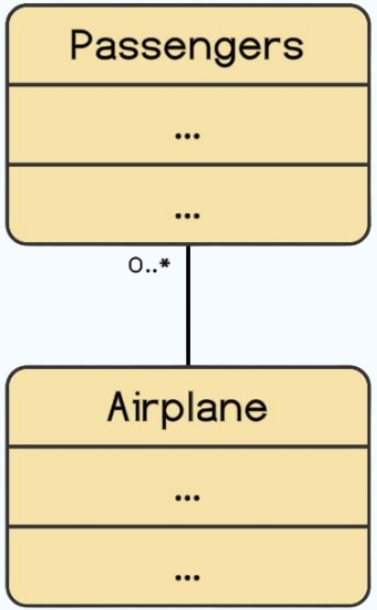
Reflexive Association

- A relationship where a class is associated with itself.
- A staff member has a supervisor, **who is also an employee**.



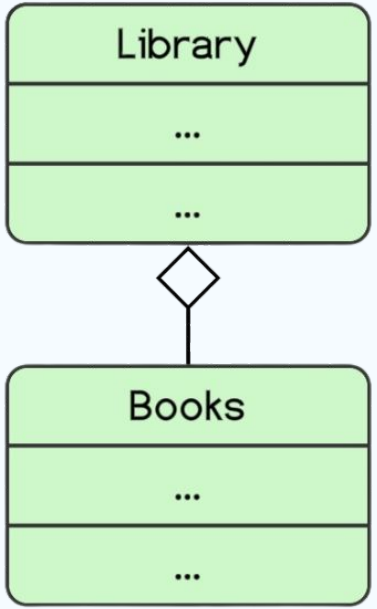
Multiplicity

- Defines how many instances of a class relate to another.
- An Airplane **has zero or more** Passengers.



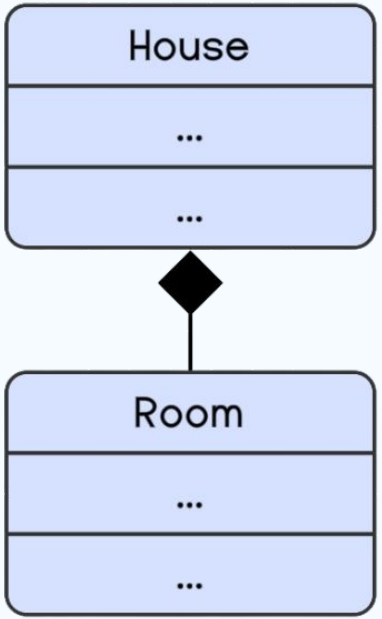
Aggregation

- Weak “whole-part” relationship.
- A Library has books, but **Books can exists without a Library**.



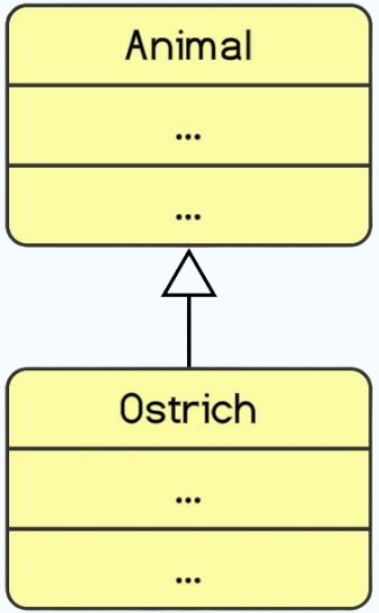
Composition

- Strong “whole-part” relationship.
- A House is composed by Rooms, but **if the House is destroyed, the Rooms are also destroyed**.



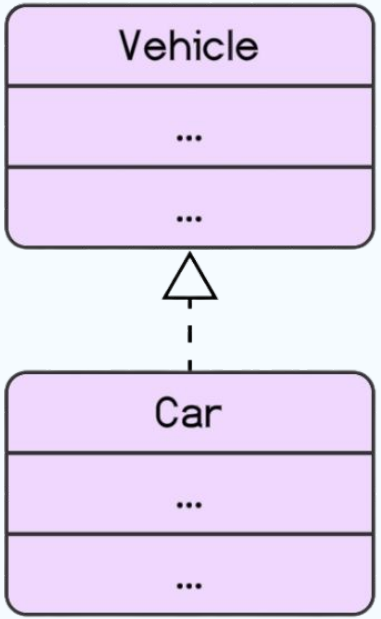
Inheritance

- Represents **parent-child relationship** where a subclass inherits from a superclass.
- **An Ostrich is an Animal**.



Realization

- A class implements an interface.
- A **Car** implements methods of the **Vehicle** interface.



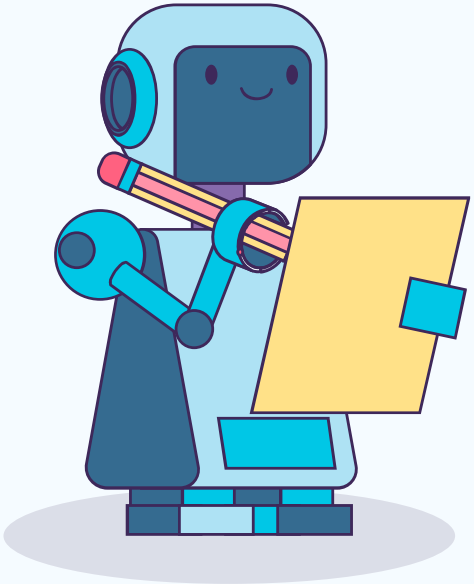
Tools to use UML

- Tons of them.
  - StarUml.
  - ArgoUML.
  - Visual Paradigm
  - Mermaid.
- We will focus on Lucidchart.

References

- Single-responsibility principle - Wikipedia
- SOLID: The First 5 Principles of Object Oriented Design | DigitalOcean
- SOLID: S - Single Responsibility Principle (SRP)
- Principio de segregación de la interfaz - Wikipedia, la enciclopedia libre
- SOLID: I - Interface Segregation Principle (ISP) - DEV Community
- SOLID Design Principles Explained: The Liskov Substitution Principle with Code Examples
- Applying SOLID principles to TypeScript
- Relaciones de diagramas de clases en UML explicadas con ejemplos
- What is Unified Modeling Language (UML)?
- UML Class Diagram Relationships Explained with Examples

Thanks!



Do you have any questions?

diego.hernandez.22

fabian.gonzalez.29

samuel.frias.40