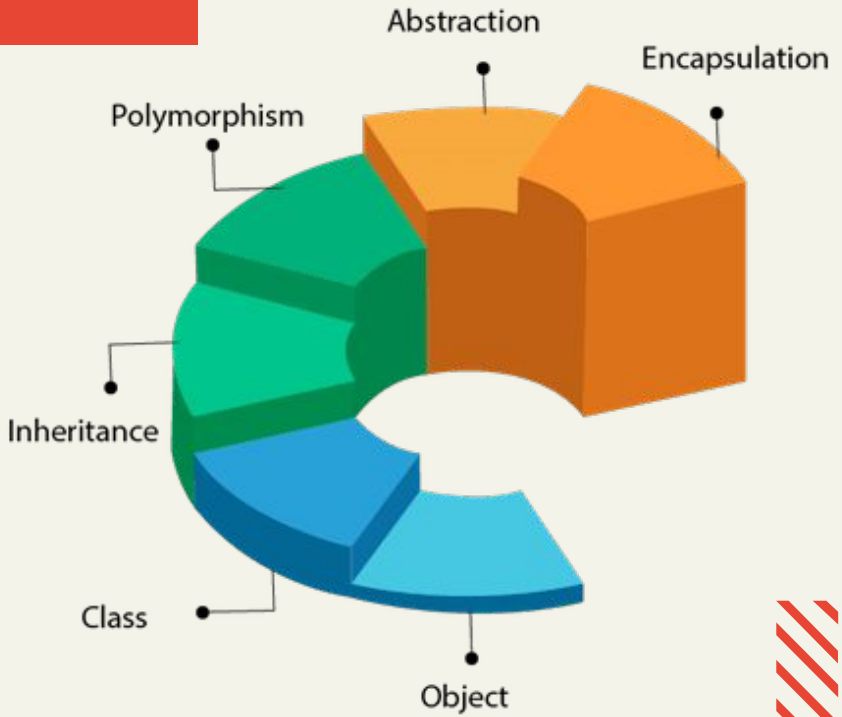


OOP

Best practices
Design principles
SOLID principles

Speakers

Miguel Luna García

miguel.luna.19@ull.edu.es

Adriano dos Santos Moreira

adriano.moreira.07@ull.edu.es

Contents

01
OOP Best Practices

- Introduction to OOP
- Visibility
- Style Guides

02
OOP Design Principles

- “Code smells”
- K.I.S.S.
- D.R.Y.
- Y.A.G.N.I.

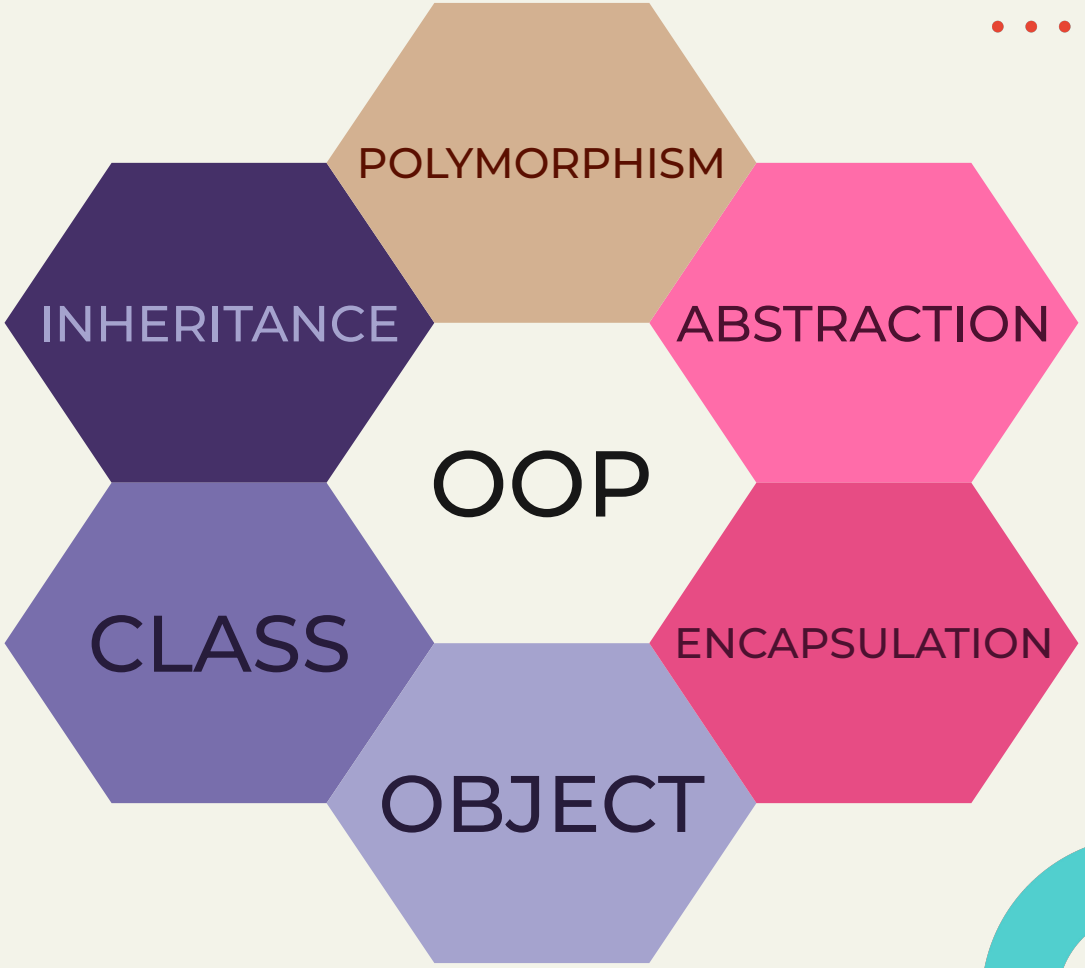
03
SOLID Principles

- Single-responsibility
- Open–closed
- Liskov substitution
- Interface segregation
- Dependency inversion

01

OOP Best Practices

Introduction to OOP in TypeScript



Object

Data field that has **unique attributes** and **behavior**

```
let employee: object;

employee = {
  firstName: 'John',
  lastName: 'Doe',
  age: 25,
  jobTitle: 'Web Developer'
};
```

- Primitive types
  - number
  - bigint
  - string
  - boolean
  - null
  - undefined
  - symbol
- Everything else is an object in JS/TS

Class

Object template

```
class Employee {
  constructor(firstName: string, lastName: string,
    age: number, jobTitle: string) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.age = age;
    this.jobTitle = jobTitle;
  }

  public toString(): string {
    return `${this.firstName} ${this.lastName}, ` +
      `${this.age}, ${this.jobTitle}`;
  }

  getAge(): number {   // Public by default.
    return this.age;
  }

  private readonly firstName: string;
  private readonly lastName: string;
  private age: number;
  private jobTitle: string;
}
```

- Constructor
- Methods
- Properties

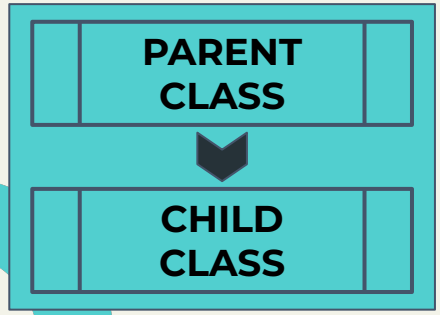
```
class Employee {
  constructor(firstName: string, lastName: string,
    age: number, jobTitle: string) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.age = age;
    this.jobTitle = jobTitle;
  }
  private readonly firstName: string;
  private readonly lastName: string;
  private age: number;
  private jobTitle: string;
}
```

```
class Employee {
  constructor(
    private readonly firstName: string,
    private readonly lastName: string,
    private age: number,
    private jobTitle: string) {
  }
```

Attributes and methods visibility

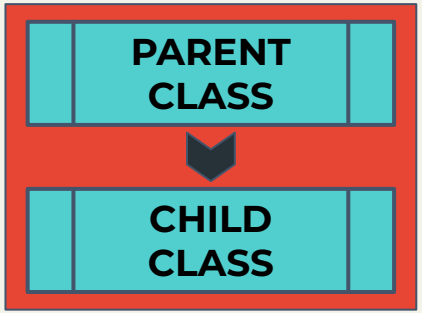
`public`

- **Outside** the class definition
- **Attributes** should NOT be public



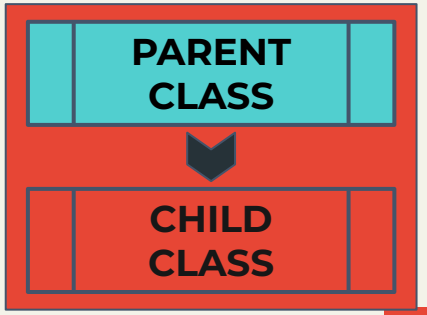
`protected`

- The **original** and **inherited** classes definition



`private`

- **Only** the **class** definition



Inheritance

Generate a new class **reusing** the properties and methods from **another class**

```
class AClass {
  constructor(private attributeA: number) {}
  getAttributeA(): number {
    return this.attributeA;
  }
}
```

```
class BClass extends AClass {
  constructor(attributeA: number, private attributeB: number) {
    super(attributeA);
  }
  getAttributeB(): number {
    return this.attributeB;
  }
}
```

Other ways to expand a class

Aggregation/Composition:

Using an **object** from a class **inside** another **class**

```
class AClass {
  sayHi(): void {
    console.log('Hi!');
  }
}
```

```
class BClass {
  constructor(private aClass?: AClass) {}
  sayHi(): void {
    if (this.aClass) this.aClass.sayHi();
  }
}
```

Aggregation

```
class CClass {
  sayHi(): void {
    this.aClass.sayHi();
  }
  private aClass = new AClass();
}
```

Composition

Other ways to expand a class

Association:

Two **independent objects interacting** with each other

```
class AClass {
  sayHi(): void {
    console.log('Hi!');
  }
  sayBye(obj: BClass): void {
    obj.sayBye();
  }
}
```

```
class BClass {
  sayBye(): void {
    console.log('Bye!');
  }
  sayHi(obj: AClass): void {
    obj.sayHi();
  }
}
```

Polymorphism

Send **syntactically alike** messages to objects of **different types**

```
interface Person {
  firstName: string;
  lastName: string;
  occupation?: string;
}
```

Interfaces (TS only):

Ensures an object has the **properties needed** to execute a piece of code

```
class Employee implements Person {
  constructor(
    public readonly firstName: string,
    public readonly lastName: string,
    public readonly occupation: string
  ) {}
}
```

```
class Student implements Person {
  constructor(
    public readonly firstName: string,
    public readonly lastName: string
  ) {}
}
```

```
function printInfo(person: Person) {
  console.log(`${person.firstName} ${person.lastName}, ` +
    (person.occupation ? person.occupation : 'Student'));
}
```

Encapsulation

Hide the implementation details of a feature

- **Information hiding**: Segregating the design decisions that are most likely to change, protecting those who will probably remain the same.

```
let aVariable = 0;

for () {
  //do something complicated
}
```

```
function somethingComplicated(aVariable) {
  for () {
    //do something complicated
  }
}

let aVariable = 0;
somethingComplicated(aVariable);
```

Abstraction

Dismiss features or attributes to focus on details of greater importance.

Address the actual problem **rather** than its details.

Example: `Vector`

Involves:

- Start pointer.
- Linked list.
- ...

No need to worry about the details!

Just use the `Vector` abstraction.

Style Guides

Best Style Practices

[Google Style for JavaScript]

[Google Style for TypeScript]

Why use style guides?

*“[...]it doesn’t matter a whit where you put your braces so long as you all agree on **where to put them**.”*

***G24**: Follow Standard Conventions, page 299.*

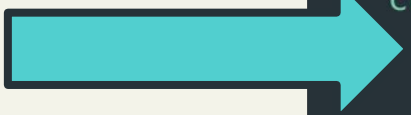
Clean Code: A Handbook of Agile Software Craftsmanship

- Robert C. Martin

JavaScript

Call `super()` **before**:

- Using `this`
- Setting any fields.



```
class Shape {
  constructor(positionX, positionY) {
    this.positionX_ = positionX;
    this.positionY_ = positionY;
  }
}

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

JavaScript

Private fields **must** have a trailing underscore_

```
class Shape {
  constructor(positionX, positionY) {
    this.positionX_ = positionX;
    this.positionY_ = positionY;
  }
}

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

JavaScript

Visibility annotations.

```
class Car {
  constructor(plate, color, brand) {
    /** @private @const {string} */
    this.plate_ = plate;
    /** @protected {string} */
    this.color = color;
    // Public fields do not need visibility annotations.
    this.brand = brand;
  }
}
```

JS JavaScript

Avoid **static** methods, prefere functions.

Do **not** manipulate `prototypes`, worsens clarity.

Do **not** use `get` and `set`

```
class Car {
  constructor(brand) {
    this.brand = brand;
  }
  get getBrand() {
    return this.brand;
  }
  set setBrand(brand) {
    this.brand = brand;
  }
}
```

TypeScript

Do not create **empty constructors**
or constructors that simply call `super()`.

```
class UnnecessaryConstructor {
  constructor() {}
}
class UnnecessaryConstructorOverride extends Base {
    constructor(value: number) {
      super(value);
    }
}
```

TypeScript

Use parameter properties.

```
class BadTeam {
  #size: number;
  constructor(size: number) {
    this.#size = size;
  }
}

class GoodTeam {
  constructor(private size: number) {}
}
```

TypeScript

Do **not** use `#private`.

```
class BadTeam {
  #size: number;
  constructor(size: number) {
    this.#size = size;
  }
}

class GoodTeam {
  constructor(private size: number) {}
}
```

TypeScript

Constructor calls ***must*** use parentheses:

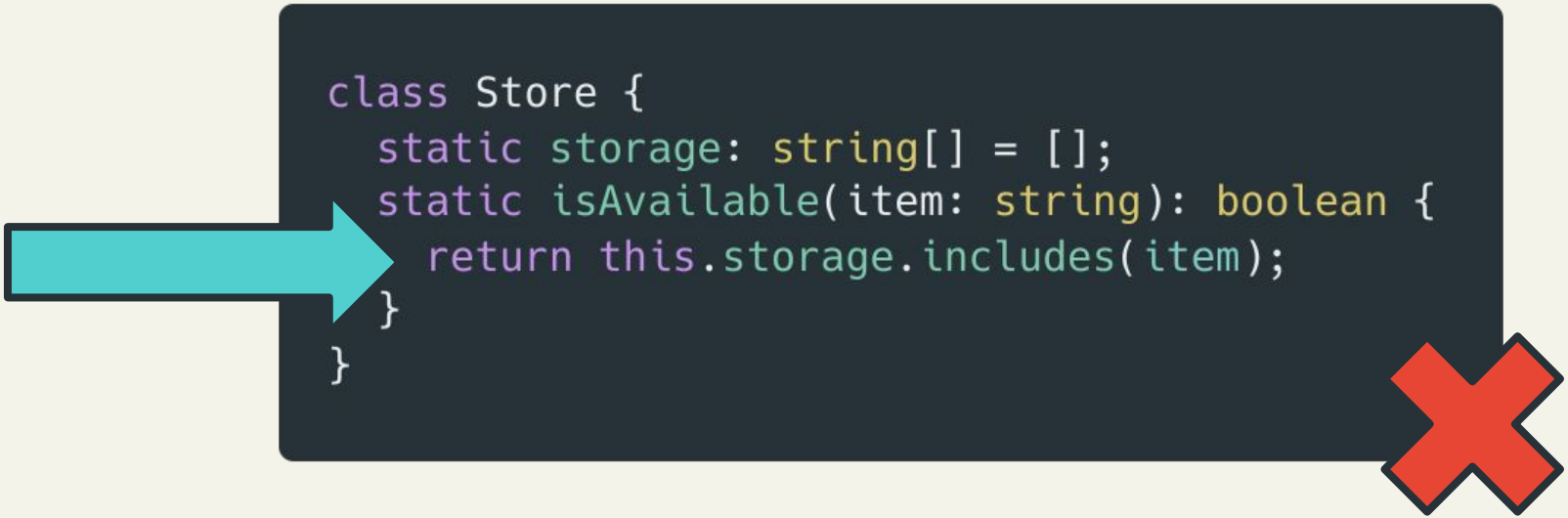
```
let myTeam = new Team();
```

```
myTeam['size'];
```

Never access private fields like this.

Breaks encapsulation!!!

TypeScript



Do **not** use `this` in static methods.

TypeScript

Recommended options for `tsconfig.json`

[Settings documentation]

```
"compilerOptions": {
  "target": "es2022",
  "module": "commonjs",
  "allowJs": true,
  "rootDir": "./",
  "outDir": "./dist",
  "declaration": true,
  "alwaysStrict": true,
  "strict": true,
  "forceConsistentCasingInFileNames": true,
  "noImplicitReturns": true,
  "noFallthroughCasesInSwitch": true,
}
```

TypeScript

```
"compilerOptions": {
  "target": "es2022",
  "module": "commonjs",
  "allowJs": true,
  "rootDir": "./",
  "outDir": "./dist",
  "declaration": true,
  "alwaysStrict": true,
  "strict": true,
  "forceConsistentCasingInFileNames": true,
  "noImplicitReturns": true,
  "noFallthroughCasesInSwitch": true,
}
```

TypeScript

```
"compilerOptions": {
  "target": "es2022",
  "module": "commonjs",
  "allowJs": true,
  "rootDir": "./",
  "outDir": "./dist",
  "declaration": true,
  "alwaysStrict": true,
  "strict": true,
  "forceConsistentCasingInFileNames": true,
  "noImplicitReturns": true,
  "noFallthroughCasesInSwitch": true,
}
```

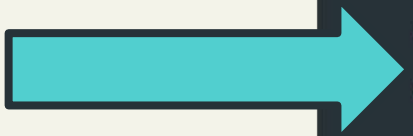
TypeScript

```
"compilerOptions": {
  "target": "es2022",
  "module": "commonjs",
  "allowJs": true,
  "rootDir": "./",
  "outDir": "./dist",
  "declaration": true,
  "alwaysStrict": true,
  "strict": true,
  "forceConsistentCasingInFileNames": true,
  "noImplicitReturns": true,
  "noFallthroughCasesInSwitch": true,
}
```

TypeScript

```
"compilerOptions": {
  "target": "es2022",
  "module": "commonjs",
  "allowJs": true,
  "rootDir": "./",
  "outDir": "./dist",
  "declaration": true,
  "alwaysStrict": true,
  "strict": true,
  "forceConsistentCasingInFileNames": true,
  "noImplicitReturns": true,
  "noFallthroughCasesInSwitch": true,
}
```

TypeScript



```
"compilerOptions": {
  "target": "es2022",
  "module": "commonjs",
  "allowJs": true,
  "rootDir": "./",
  "outDir": "./dist",
  "declaration": true,
  "alwaysStrict": true,
  "strict": true,
  "forceConsistentCasingInFileNames": true,
  "noImplicitReturns": true,
  "noFallthroughCasesInSwitch": true,
}
```

TypeScript

`"strict": true` enables the following:

```
"compilerOptions": {
    "strictNullChecks": true,
    "strictBindCallApply": true,
    "strictFunctionTypes": true,
    "strictPropertyInitialization": true,
    "noImplicitAny": true,
    "noImplicitThis": true,
    "useUnknownInCatchVariables": true
  }
```

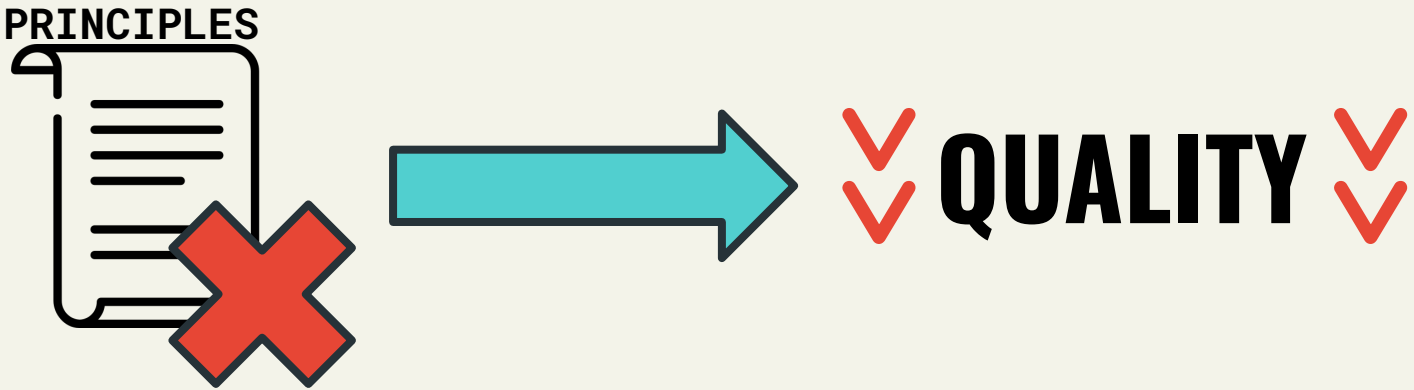
Settings documentation

02

OOP Design Principles

Code Smells

*"Structures in the code that indicate **violation** of fundamental design principles and negatively impact design quality".*



Code Smells

- Not technically incorrect.
- Do not prevent code from working.

Smells **awful** but still works!

Code Smells

- Bug risk. 
- **Slower** development.

Indicates the need to **REFACTOR**

Code Smells

Common Smells

Duplicated code

Duplicated code

```
function addOne(input: number) {
  return input + 1;
}
function addTwo(input: number) {
  return input + 2;
}
function addThree(input: number) {
  return input + 3;
}
```

Code Smells

Common Smells

Mysterious Name.

???

```
class DtaRcrd102 {
  private genymdhms: number = 2;
  private modymdhms: number = 3;
  private pszqint: string = '102';
  /* ... */
};
```

Code Smells

Common Smells

Loooong things.

```
public class SuperDashboard extends JFrame implements MetaDataUser
  public String getCustomizerLanguagePath()
  public void setSystemConfigPath(String systemConfigPath)
  public String getSystemConfigDocument()
  public void setSystemConfigDocument(String systemConfigDocument)
  public boolean getGuruState()
  public boolean getNoviceState()
  public boolean getOpenSourceState()
  public void showObject(MetaObject object)
  public void showProgress(String s)
  public boolean isMetadataDirty()
```

```
public void setIsMetadataDirty(boolean isMetadataDirty)
public Component getLastFocusedComponent()
public void setLastFocused(Component lastFocused)
public void setMouseSelectState(boolean isMouseSelected)
public boolean isMouseSelected()
public LanguageManager getLanguageManager()
public Project getProject()
public Project getFirstProject()
public Project getLastProject()
public String getNewProjectName()
public void setComponentSizes(Dimension dim)
public String getCurrentDir()
public void setCurrentDir(String newDir)
public void updateStatus(int dotPos, int markPos)
public Class[] getDataBaseClasses()
public MetadataFeeder getMetadataFeeder()
public void addProject(Project project)
public boolean setCurrentProject(Project project)
public boolean removeProject(Project project)
public MetaProjectHeader getProgramMetadata()
public void resetDashboard()
public Project loadProject(String fileName, String projectName)
```

```
public void setCanSaveMetadata(boolean canSave)
public MetaObject getSelectedObject()
public void deselectObjects()
public void setProject(Project project)
public void editorAction(String actionName, ActionEvent event)
public void setMode(int mode)
public FileManager getFileManager()
public void setFileManager(FileManager fileManager)
public ConfigManager getConfigManager()
public void setConfigManager(ConfigManager configManager)
public ClassLoader getClassLoader()
public void setClassLoader(ClassLoader classLoader)
public Properties getProps()
public String getUserHome()
public String getBaseDir()
public int getMajorVersionNumber()
public int getMinorVersionNumber()
public int getBuildNumber()
public MetaObject pasting(
MetaObject target, MetaObject pasted, MetaProject project)
public void processMenuItems(MetaObject metaObject)
public void processMenuSeparators(MetaObject metaObject)
```

```
  public void processTabPages(MetaObject metaObject)
  public void processPlacement(MetaObject object)
  public void processCreateLayout(MetaObject object)
  public void updateDisplayLayer(MetaObject object, int layerIndex)
  public void propertyEditedRepaint(MetaObject object)
  public void processDeleteObject(MetaObject object)
  public boolean getAttachedToDesigner()
  public void processProjectChangedState(boolean hasProjectChanged)
  public void processObjectNameChanged(MetaObject object)
  public void runProject()
  public void setAçowDragging(boolean allowDragging)
  public boolean allowDragging()
  public boolean isCustomizing()
  public void setTitle(String title)
  public IdeMenuBar getIdeMenuBar()
  public void showHelper(MetaObject metaObject, String propertyName)
  // ... many non-public methods follow ...
}
```

Stinky class...

Code Smells

Common Smells

Lazy class
(or function)

```
class MyNumber {
  constructor(public number: number) {}
}
```

Too small!!!

Code Smells

Common Smells

Shotgun surgery.

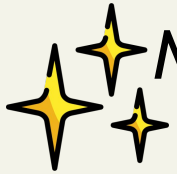
If we need to **add information** to the logs.

Every function would require changes.

```
function myFunction() {
  console.log('Entering myFunction');
  /* ... */
}
function myFunction2() {
  console.log('Entering myFunction2');
  /* ... */
}
function myFunction3() {
  console.log('Entering myFunction3');
  /* ... */
}
```

Code Smells

Common Smells



Magic Numbers

Arbitrary and usually **meaningless** by themselves

```
function badCalculatePrice(base: number,
                           type: string) {
  if (type === 'AC524') {
    return base * 2.34;
  }
  return base * 1.58;
}
```

Code Smells

Common Smells

Deeply nested statements

Needs ***deep*** refactoring.

```
function deeplyNested(input1: number,
                      input2: number) {
  if (!Number.isNaN(input1)) {
    if (!Number.isNaN(input2)) {
      if (input1 > 0) {
        if (input1 < Infinity) {
          switch (input1) {
            case 1:
              if (input2 > 0) {
                /** Code */
              }
          }
        }
      }
    }
  }
}
```

K.I.S.S.

Keep It Simple, Stupid

Design principle

Simple design > Unnecessary complexity

K.I.S.S.

Keep It Simple, Stupid

```
function goodIsEven(input: number) {
  if (Number.isNaN(input)) {
    throw new Error('Not a number.');
  }
  return input % 2 === 0;
}
```

```
function badIsEven(input: number) {
  const NUMBER_REGEX = /[0123456789]/;
  const INPUT_AS_TEXT = String(input);
  for (const NUMBER of INPUT_AS_TEXT) {
    if (!NUMBER_REGEX.test(NUMBER)) {
      throw new Error('Not a number.');
    }
  }
  let isEven: boolean;
  if (input % 2 === 0) {
    isEven = true;
  } else {
    isEven = false;
  }
  return isEven;
}
```

D.R.Y.

Don’t Repeat Yourself

“Every piece of knowledge must have a ***single****, unambiguous, authoritative* ***representation*** *within a system.”*

Remember this slide?

Code Smells

Common Smells

How ironic...

Duplicated code

Duplicated code

```
function addOne(input: number) {
  return input + 1;
}
function addTwo(input: number) {
  return input + 2;
}
function addThree(input: number) {
  return input + 3;
}
```

Y.A.G.N.I

You Aren't Going to Need It

*"Implement things when you **actually** need them"*

Don't bother

Getting ahead is **rarely profitable**.

Y.A.G.N.I

You Aren't Going to Need It

```
class badCar {
  constructor(private plate: string,
              private brand: string) {}
  getPlate(): string {
    return this.plate;
  }
  setPlate(plate: string): void {
    this.plate = plate;
  }
  getBrand(): string {
    return this.brand;
  }
  setBrand(brand: string): void {
    this.brand = brand;
  }
  /** More public methods */
}
```

```
class goodCar {
  constructor(private plate: string,
              private brand: string) {}
  /** More public methods */
}
```

Getters / Setters
may be unnecessary.

03

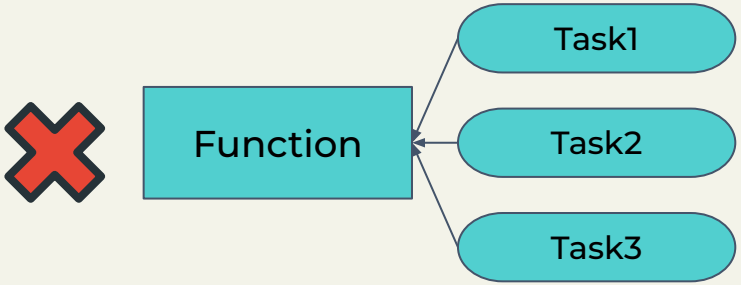
OOP SOLID Principles

S	O	L	I	D
Single-responsibility principle	**Open-closed principle**	**Liskov substitution principle**	**Interface segregation principle**	**Dependency inversion principle**

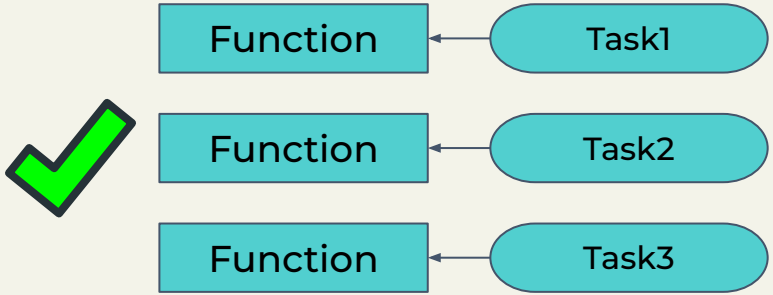
Single-responsibility principle

"There should never be more than one reason for a class to change."

Pieces of code **responsible** for **different tasks**.



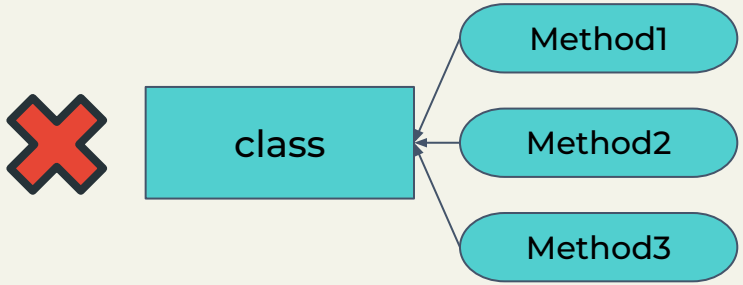
Create different classes, functions, etc... to **separate tasks**.



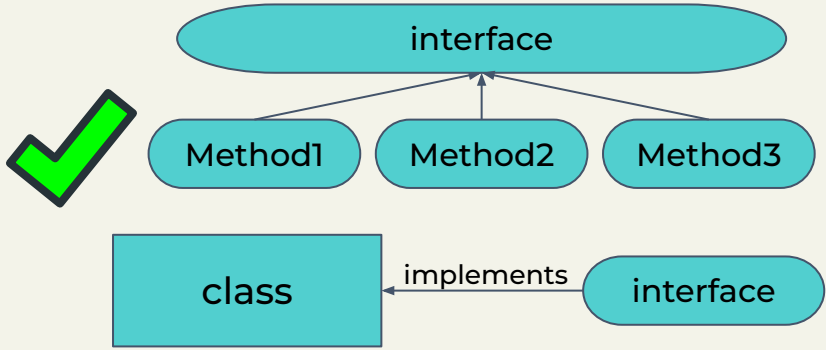
Example of SRP

Open-closed principle

"Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification."



Code that **needs to be modified** in order to expand its functionality



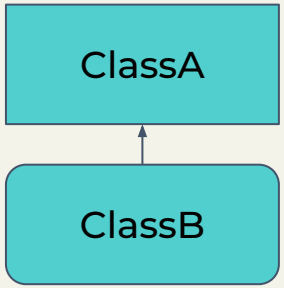
Code that **can be expanded** without modifying the source code.

[Example of OCP]

Liskov substitution principle

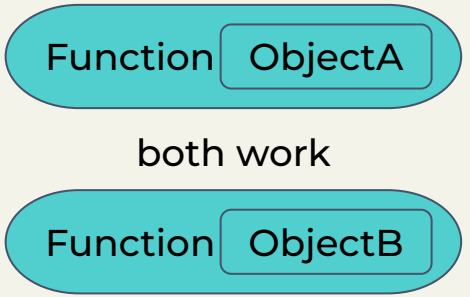
"Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it."

Barbara Liskov



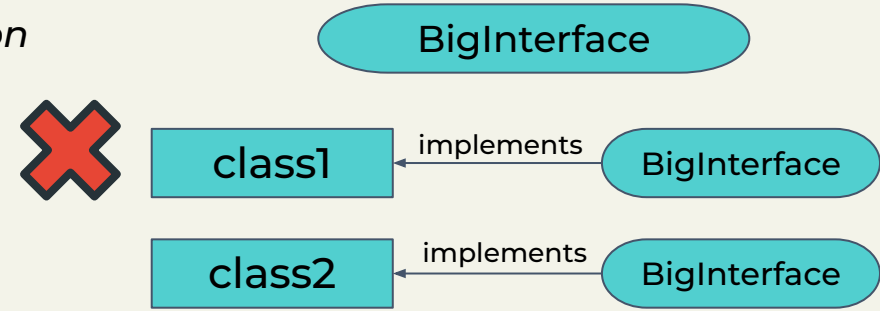
Inherited class objects that **cannot use every** piece of **code** the original class could use

Objects that **can be replaced** with other objects from an inherited class



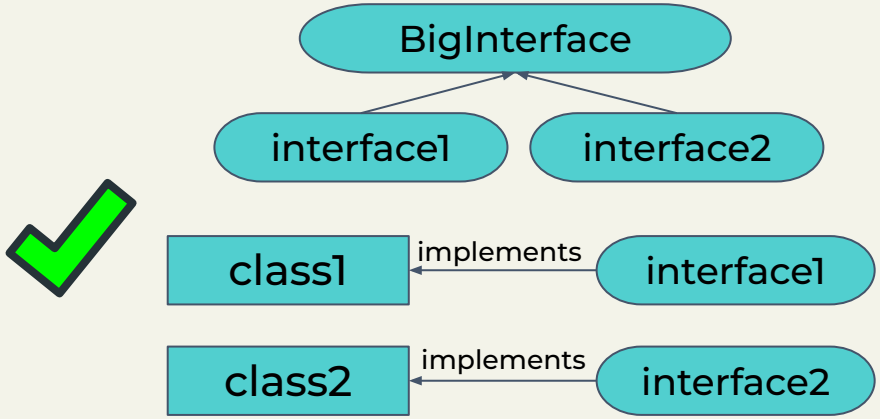
Interface segregation principle

"Clients should not be forced to depend upon interfaces that they do not use."



Code **dependant** on **methods** it does **not use**

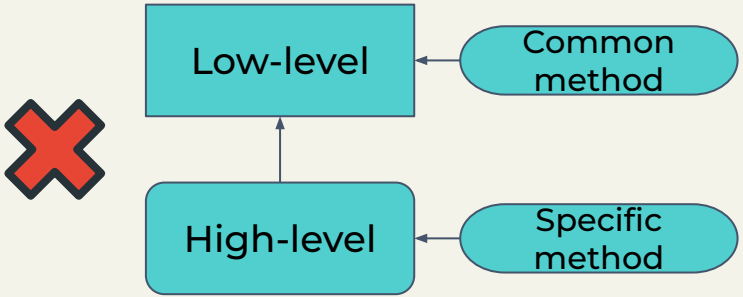
Splits interfaces that are very large into **smaller** and more **specific** ones



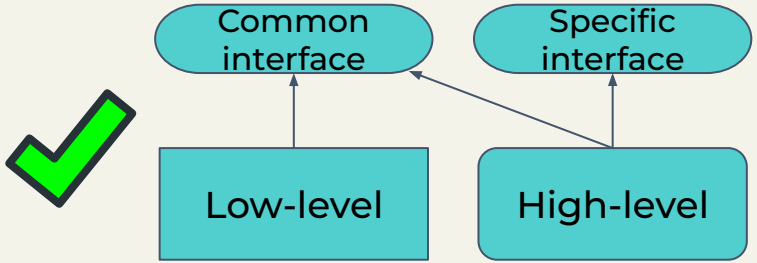
Dependency inversion principle

"Depend upon abstractions, [not] concretions."

High-level class inherited from low level class



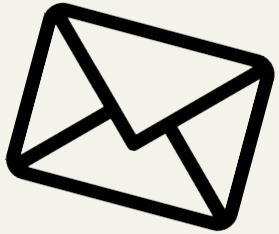
Two independent classes with shared interface.



[Example of DIP]

Thanks!

Send us an **email** if you have any questions!



Miguel Luna García

miguel.luna.19@ull.edu.es

Adriano dos Santos Moreira

adriano.moreira.07@ull.edu.es

References

- Wikipedia - Code Smell
- Wikipedia - Shotgun Surgery
- Libro - Clean code : a handbook of agile software craftsmanship
- Wikipedia - Don't Repeat Yourself
- Wikipedia - You Aren't Gonna Need It
- Stackify - Dependency Inversion Principle
- Slides - OOP in TypeScript
- GitHub - Introduction to TypeScript
- TypeScript Tutorial
- TypeScript Documentation: Classes
- Wikipedia - SOLID
- Google TypeScript Style Guide
- Google JavaScript Style Guide
- IONOS - What is code smell?
- IONOS - Clean code: what makes programming code clean