Universidad de La Laguna

OOP Principles, Good Practices, Code Smells, Clean Code, Refactoring

Érika Crespo Molero
erika.crespo.27@ull.edu.es

Javier Acosta Portocarrero
javier.acosta.39@ull.edu.es

César Navarro Santos
cesar.navarro.10@ull.edu.es

Index

- OOP concepts
- Google TypeScript Style Guide
- Good practices
- Clean code
- Code smells

Visibility

- Visibility determines where a class's properties and methods can be accessed. TS supports:
  - Public
  - Protected
  - Private

Public Modifier

- Accessible inside and outside the class.

Public Modifier

- Accessible inside and outside the class.
- Declared using the public keyword.

Public Modifier

- Accessible inside and outside the class.
- Declared using the public keyword.
- Default visibility.

public-modifier.ts

Protected Modifier

- Accessible only inside the class and subclasses.

Protected Modifier

- Accessible only inside the class and subclasses.
- Declared using the protected keyword.

protected-modifier.ts

Private Modifier

- Accessible only inside the class.

Private Modifier

- Accessible only inside the class.
- Declared using the private keyword.

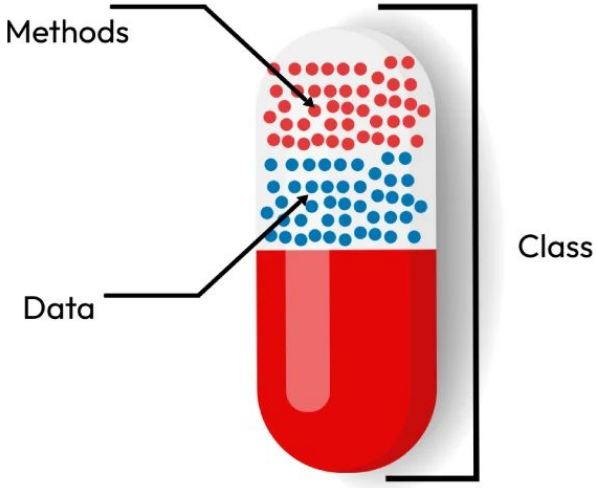
Private Modifier

- Accessible only inside the class.
- Declared using the private keyword.
- All class members should be private by default whenever possible.

`private-modifier.ts`

Encapsulation

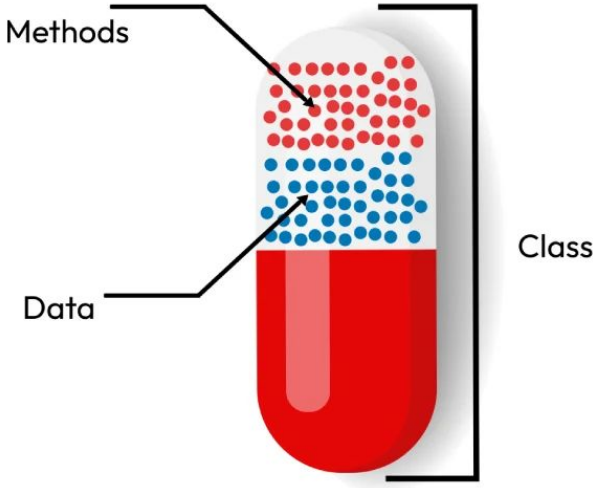
- Bundling data (properties) and behavior (methods) in a class.



Encapsulation

- Bundling data (properties) and behavior (methods) in a class.

- Restricting direct access to its internal state.



Abstraction

- Providing what an object does while hiding how it works, so the user doesn’t need to know the implementation details.

Abstraction

- Providing what an object does while hiding how it works, so the user doesn’t need to know the implementation details.

- Exposing an **external interface** for client interaction.

Abstraction

- Providing what an object does while hiding how it works, so the user doesn't need to know the implementation details.
- Exposing an **external interface** for client interaction.
- Using an **internal interface** for implementation details.

External Interface

- What other classes or code can access.

External Interface

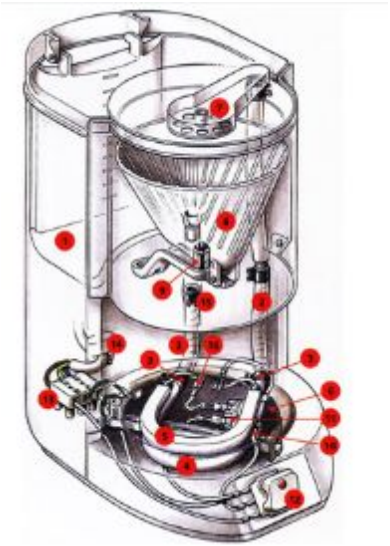
- What other classes or code can access.
- Exposes selected functionality via getters, setters, or public methods.

External Interface

- What other classes or code can access.
- Exposes selected functionality via getters, setters, or public methods.
- Ensures safe interaction with the object.

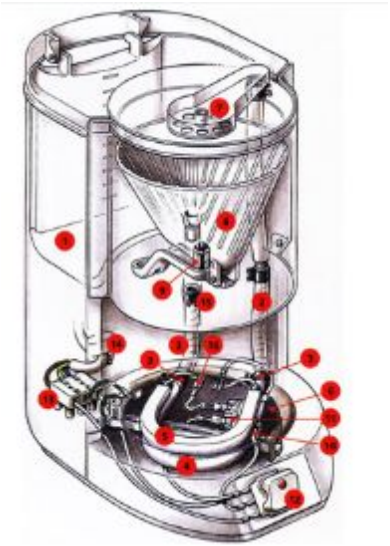
Internal Interface

- Used only within the class.



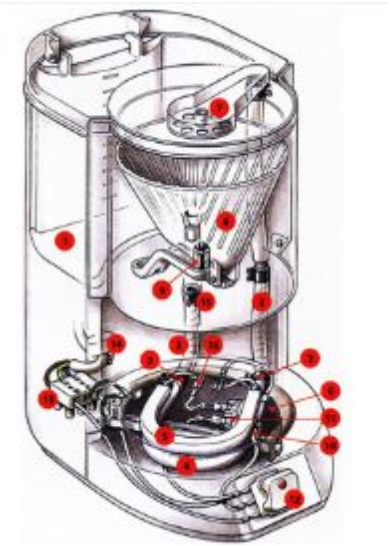
Internal Interface

- Used only within the class.
- Contains helper methods and internal data.



Internal Interface

- Used only within the class.
- Contains helper methods and internal data.
- Hidden from external code using private or protected visibility.



encapsulation-abstraction

Class relationships

- Association
- Aggregation
- Composition
- Inheritance

Association

- One object is linked to another object.

Association

- One object is linked to another object.

association.ts

Aggregation

- One object contains another object.

Aggregation

- One object contains another object.
- The contained object can exist independently of the container.

Aggregation

- One object contains another object.
- The contained object can exist independently of the container.

aggregation.ts

Composition

- One object contains another object.

Composition

- Object made out of other objects.
- The contained object cannot exist independently of the container

Composition

- Object made out of other objects.
- The contained object cannot exist independently of the container

composition.ts

Delegation

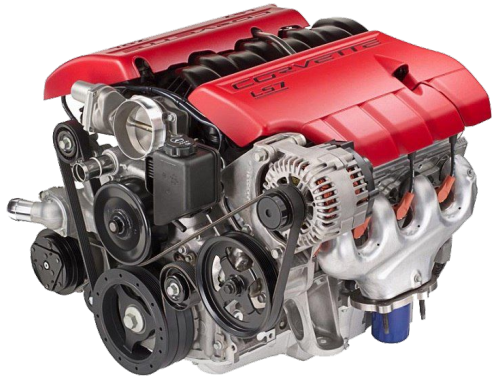
- Design pattern.

Delegation

- Design pattern.
- Handling the responsibility to other object.

Delegation

- Design pattern.
- Handling the responsibility to other object.



Inheritance

- It supports multi-level inheritance (class C inherits from B which inherits from A).

Inheritance

- It supports multi-level inheritance (class C inherits from B which inherits from A).
- Use of the keyword extends.

Inheritance

- It supports multi-level inheritance (class C inherits from B which inherits from A).
- Use of the keyword extends.

```
class Animal {
  constructor(protected age: number) {}
}

class Dog extends Animal {
  …
}
```

extends-super.ts

Inheritance

- It supports multi-level inheritance (class C inherits from B which inherits from A).
- Use of the keyword extends.
- Use of super().

Inheritance

- It supports multi-level inheritance (class C inherits from B which inherits from A).
- Use of the keyword extends.
- Use of super().

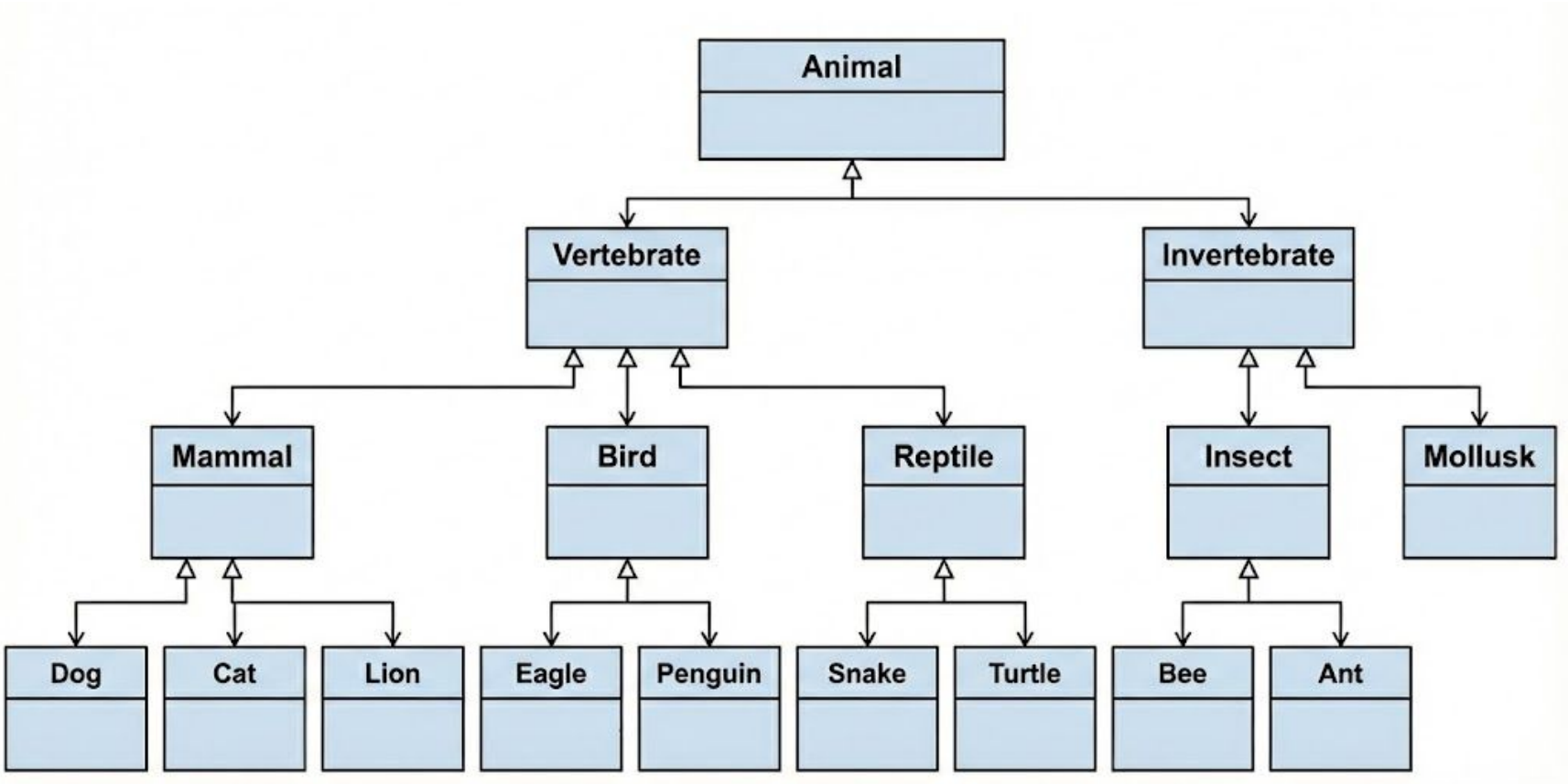
```
class Animal {
  constructor(protected age: number) {}
}

class Dog extends Animal {
  constructor(age: number, private breed: string) {
    super(age);
  }
}
```

Inheritance

- It supports multi-level inheritance (class C inherits from B which inherits from A).
- Use of the keyword extends.
- Use of super().
- Method overriding.

method-overriding.ts

Inheritance



Inheritance

- Composition over Inheritance.
- Has-A | Is-A.

composition.ts

Inheritance

- Composition over Inheritance.
- Has-A | Is-A.
- Interfaces.

interfaces.ts

Polymorphism

- It allows us to treat objects of different classes uniformly.

Polymorphism

- It allows us to treat objects of different classes uniformly.
- In Typescript, this is achieved through interfaces.

polymorphism.ts

Abstract Classes

- Abstract classes are base classes from which other classes may be derived.

Abstract Classes

- Abstract classes are base classes from which other classes may be derived.
  - They may not be instantiated directly.

```
abstract class Animal {
  abstract makeSound(): void;
}
```

Abstract Classes

- Abstract methods must be implemented in derived classes.

`abstract-example.ts`

Interfaces

- An interface is a design contract or abstract specification that defines.

Interfaces

- An interface is a design contract or abstract specification that defines.
  - Methods.

Interfaces

- An interface is a design contract or abstract specification that defines.
  - Methods.
  - Properties.

Interfaces

- An interface is a design contract or abstract specification that defines.
  - Methods.
  - Properties.
- Entities must possess them.

Interfaces

- Object shape and strong type checking

shape-strong-type-checking.ts

Interfaces

- Function Types

function-type.ts

Interfaces

- Class contracts and multiple interfaces

class-contracts.ts

Universidad de La Laguna

Google TypeScript Style Guide

Class Declarations

- Must not be terminated with a semicolon:

```
// Don't do this
class BadExample {
}; // Unnecessary semicolon
```

```
// Do this
class GoodExample {
}
```

Class Declarations

- In contrast, class expressions must be terminated with semicolons:

```
export const Baz = class extends Bar {
  method(): number {
    return this.x;
  }
}; // Semicolon here as this is a statement
```

Class Declarations

- Blank lines separating class declaration braces from other class content are valid and optional.

```
// No spaces around braces - fine.
class Baz {
  method(): number {
    return this.x;
  }
}
```

Class Declarations

- Blank lines separating class declaration braces from other class content are valid and optional.

```
// A single space around both braces - also fine.
class Foo {

  method(): number {
    return this.x;
  }

}
```

Constructors

- Constructor calls must use parentheses:

```
// Don't do this
const x = new Foo;
```

```
// Do this
const x = new Foo();
```

constructor-calls.ts

Constructors

- Don’t provide an empty constructor without parameters.

Constructors

- Don’t provide an empty constructor without parameters.
- Don’t provide a constructor that only delegates into its parent class.

Constructors

- Don’t provide an empty constructor without parameters.
- Don’t provide a constructor that only delegates into its parent class.
- Constructors that define properties, visibility, or decorators must not be omitted.

`empty-constructor.ts`

Constructors

- The constructor must be preceded and followed by exactly one blank line.

```
// Don't do this
class BadExample {
  myField = 10;
  constructor(private readonly ctorParam) {}
  doThing() {
    console.log(ctorParam.getThing() + myField);
  }
}
```

Constructors

- The constructor must be preceded and followed by exactly one blank line.

```
// Do this
class GoodExample {
  myField = 10;

  constructor(private readonly ctorParam) {}

  doThing() {
    console.log(ctorParam.getThing() + myField);
  }
}
```

Class method declarations

- Must not use a semicolon to separate individual method declarations.

Class method declarations

- Must not use a semicolon to separate individual method declarations.

```
// Don't do this
class BadExample {
  badMethod() {
    console.log('Bad');
  }; // Unnecessary semicolon
}
```

```
// Do this
class GoodExample {
  goodMethod() {
    console.log('Good');
  }
}
```

Class method declarations

- Should be separated from surrounding code by a single blank line.

Class method declarations

- Should be separated from surrounding code by a single blank line.

```
// Do this
class Good {
  doThing() {
    console.log('A');
  }

  getOtherThing(): number {
    return 4;
  }
}
```

Class method declarations

- Overriding toString.

Class method declarations

- Overriding toString: must always succeed and never have visible side effects.

Static methods

- Shared among all instances of a class.

Static methods

- Shared among all instances of a class.
- Avoid private static methods.

Static methods

- Shared among all instances of a class.
- Avoid private static methods.
- Do not rely on dynamic dispatch.

Static methods

- Shared among all instances of a class.
- Avoid private static methods.
- Do not rely on dynamic dispatch.
- Code must not use this in a static context.

Static methods

- Shared among all instances of a class.
- Avoid private static methods.
- Do not rely on dynamic dispatch.
- Code must not use this in a static context.

`bad_static_this.ts`

Class Members

- readonly.
- Parameter properties.
- Field initializers.
- Getters / Setters.

readonly

- Mark with readonly the properties that will never be reassigned outside of the constructor.

```
// Do this
class Foo {
  constructor(private readonly name: string) {}
}
```

Parameter properties

- Do not introduce an obvious initializer into a class member.

```
// Don't do this
class Animal {
  private readonly age: number;
  constructor(age: number) {}
}
```

Parameter properties

- Instead use TypeScript parameter property.

```
// Do this
class Animal {
  constructor(private readonly age: number) {}
}
```

Field initializers

- If the class member is not a parameter.

```
// Don't do this
class Game {
  private readonly score: number;
  constructor() {
    this.score = 3;
  }
}
```

Field initializers

- Initialize it where it's declared.

```
// Do this
class Game {
  private readonly score: number = 3;
}
```

Getters

- The getter method must be a pure function.

```
// Don't do this
class Foo {
  nextID = 0;
  …
  getNext() {
    return this.nextId++;
  }
}
```

Getters

- The getter method must be a pure function.

```
// Don't this
class Foo {
  nextID = 0;
  …
  getNext() {
    return this.nextId++;
  }
}
```

```
// Do this
class Foo {
  nextID = 0;
  …
  getNext() {
    return this.nextId;
  }
}
```

Getters / Setters

- Use the prefix|suffix *internal* or *wrapped*.

```
// Don't this
class Foo {
  private bar_ = '';
}
```

```
// Do this
class Foo {
  private wrappedBar = '';
}
```

Getters / Setters

- Access the value through the accessor whenever possible.

```
// Don't do this
class Foo {
  private wrappedBar = '';
  …
  setBar(wrapped: string) {
    this.wrappedBar = wrapped.trim();
  }

  foo() {
    this.wrappedBar = 'foo';
  }
}
```

Getters / Setters

- Access the value through the accessor whenever possible.

```
// Do this
class Foo {
  private wrappedBar = '';
  …
  setBar(wrapped: string) {
    this.wrappedBar = wrapped.trim();
  }

  foo() {
    this.setBar('foo');
  }
}
```

Getters / Setters

- Do not define pass-through accessors only for the purpose of hiding a property.

```
// Don't do this
class Foo {
  private wrappedBar = '';
  …
  getBar() {
    return this.wrappedBar;
  }

  setBar(wrapped: string) {
    this.wrappedBar = wrapped;
  }
}
```

Getters / Setters

- Do not define pass-through accessors only for the purpose of hiding a property.

```
// Do this
class Foo {
  private wrappedBar = '';
  …
  getBar() {
    return this.wrappedBar || 'bar';
  }

  setBar(wrapped: string) {
    this.wrappedBar = wrapped.trim();
  }
}
```

Visibility

- Decoupling and Minimum Visibility.
  - Restricting visibility helps with keeping code decoupled.

bad-visibility.ts

Visibility

- Decoupling and Minimum Visibility.
  - Restricting visibility helps with keeping code decoupled.

- Bringing private out of the class.
  - Consider converting private methods into non-exported functions.

`function-out-class.ts`

Visibility

- The ban on the word public.
  - Never use the public modifier except when declaring non-readonly public parameter properties (in constructors).

Universidad de La Laguna

Good Practices

KISS Principle

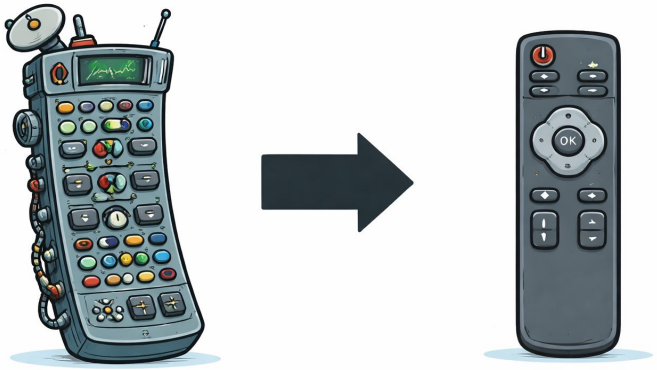
- Design principle that promotes maximum simplicity.

KISS Principle

- Design principle that promotes maximum simplicity.
- Avoids unnecessary complexity in systems and designs.

KISS Principle

- Design principle that promotes maximum simplicity.
- Avoids unnecessary complexity in systems and designs.



KISS Principle OOP

- Create only methods that are actually necessary.

KISS Principle OOP

- Create only methods that are actually necessary.
- Avoid “just in case” functionalities.

KISS Principle OOP

- Create only methods that are actually necessary.
- Avoid “just in case” functionalities.
- Prefer clear, small classes over large, complex ones.

KISS Principle OOP

- Create only methods that are actually necessary.
- Avoid “just in case” functionalities.
- Prefer clear, small classes over large, complex ones.
- Keep methods short and focused on one responsibility.

kiss-example

KISS Principle Benefits

- Helps to identify and resolve issues more quickly.

KISS Principle Benefits

- Helps to identify and resolve issues more quickly.
- Significantly reduces technical debt, as your code is more simple and understandable.

KISS Principle Benefits

- Helps to identify and resolve issues more quickly.
- Significantly reduces technical debt, as your code is more simple and understandable.
- Makes it easier to make changes and pivot your project if necessary.

KISS Principle Benefits

- Helps to identify and resolve issues more quickly.
- Significantly reduces technical debt, as your code is more simple and understandable.
- Makes it easier to make changes and pivot your project if necessary.
- Clean, focused classes and methods scale better as your project grows.

DRY Principle

- Don’t Repeat Yourself.

DRY Principle

- Don’t Repeat Yourself.
- Duplication is waste.

DRY Principle

- Don’t Repeat Yourself.
- Duplication is waste.

```
//Don’t do this
class Dog {
  constructor(private name: string) {}

  logName(): void {
    console.log(this.name);
  }
}
```

DRY Principle

- Don’t Repeat Yourself.
- Duplication is waste.

```
//Don’t do this
class Cat {
  constructor(private name: string) {}

  logName(): void {
    console.log(this.name);
  }
}
```

DRY Principle

- Don’t Repeat Yourself.
- Duplication is waste.

```
class Pet {
  constructor(private name: string) {}

  logName(): void {
    console.log(this.name);
  }
}

class Dog extends Pet {}
class Cat extends Pet {}
```

DRY Principle

- Don’t Repeat Yourself.
- Duplication is waste.
- Repetition in process calls for automation.

DRY Principle

- Don’t Repeat Yourself.
- Duplication is waste.
- Repetition in process calls for automation.
- Repetition in logic calls for abstraction.

DRY Principle Benefits

- Improved maintainability.

DRY Principle Benefits

- Improved maintainability.
- Removes the risk of inconsistent changes.

DRY Principle Benefits

- Improved maintainability.
- Removes the risk of inconsistent changes.
- Enhanced readability and clarity.

DRY Principle Benefits

- Improved maintainability.
- Removes the risk of inconsistent changes.
- Enhanced readability and clarity.
- Increased productivity.

DRY Principle Benefits

- Improved maintainability.
- Removes the risk of inconsistent changes.
- Enhanced readability and clarity.
- Increased productivity.
- Easier testing.

YAGNI

- You Aren’t Going to Need It.

YAGNI

- You Aren't Going to Need It

```
// Don't do this
class Foo {
  …
  get bar() {...}
  get foo() {...}
  get baz() {...}
  set bar(...) {...}
  set foo(...) {...}
  set baz(...) {...}
  …
}
```

YAGNI

- You Aren't Going to Need It

```
// Don't do this
class Foo {
  …
  get bar() {...}
  get foo() {...}
  get baz() {...}
  set bar(...) {...}
  set foo(...) {...}
  set baz(...) {...}
  …
}
```

```
// Do this
class Foo {
  …
  get bar() {...}
}
```

YAGNI Benefits

- Saving time and resources.

YAGNI Benefits

- Saving time and resources.
- Code simplification.

YAGNI Benefits

- Saving time and resources.
- Code simplification.
- Improving the maintainability of the project.

YAGNI Benefits

- Saving time and resources.
- Code simplification.
- Improving the maintainability of the project.
- Reduction of errors and bugs.

Universidad de La Laguna

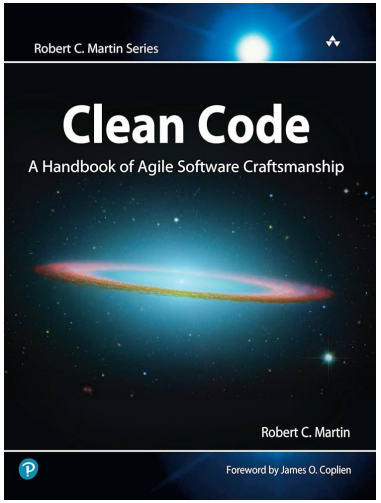
Clean Code

Clean Code

- Code that is easy to read, understand, and maintain.

Clean Code

- Code that is easy to read, understand, and maintain.
- Made popular by Robert Cecil Martin.



Read Clean Code

Clean Code

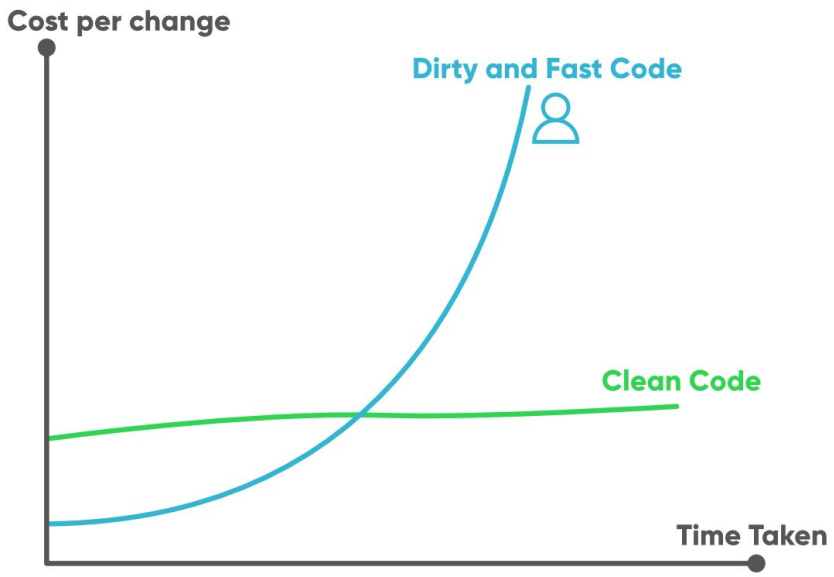
- Does it actually make a difference?

Clean Code

- Does it actually make a difference?
- Dirty fast code VS Clean Code.

Clean Code

- Does it actually make a difference?
- Dirty fast code VS Clean Code.



Cohesion

- Classes should have a small number of instance variables.

Cohesion

- Classes should have a small number of instance variables.
- Each of the methods of a class should manipulate one or more of those variables.

Cohesion

- Classes should have a small number of instance variables.
- Each of the methods of a class should manipulate one or more of those variables.
- High cohesion low coupling.

Cohesion

- Classes should have a small number of instance variables.
- Each of the methods of a class should manipulate one or more of those variables.
- High cohesion low coupling.

bad_cohesion.ts

good_cohesion.ts

Functions

- Too Many Arguments.

Functions

- Too Many Arguments.
- Output Arguments.

Functions

- Too Many Arguments.
- Output Arguments.
- Flag Arguments.

Functions

- Too Many Arguments.
- Output Arguments.
- Flag Arguments.
- Dead Function.

Universidad de La Laguna

Code Smells

Code Smells

- Hint that our code is not “clean”.

Code Smells

- Hint that our code is not “clean”.
- Surface indication that usually corresponds to a deeper problem in the system.

Code Smells

- Hint that our code is not “clean”.
- Surface indication that usually corresponds to a deeper problem in the system.
- Can be used to track down the problem.

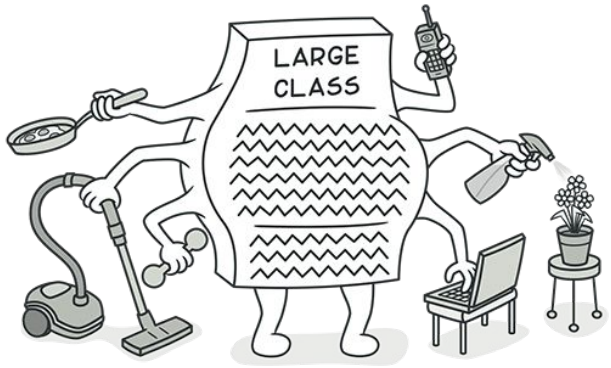
Code Smells

- Hint that our code is not “clean”.
- Surface indication that usually corresponds to a deeper problem in the system.
- Can be used to track down the problem.
- Calling something a CodeSmell is not an attack; it's simply a sign that a closer look is warranted.

More Information Here

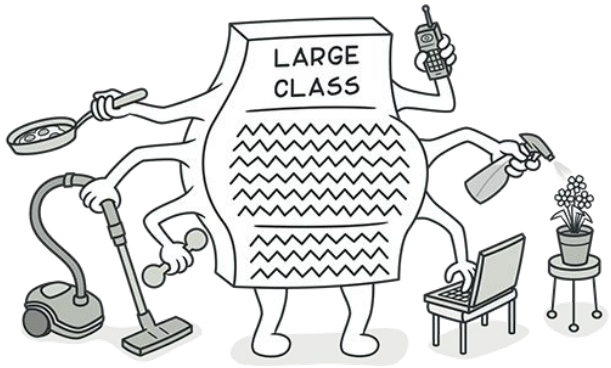
God Class

- The class that does everything.



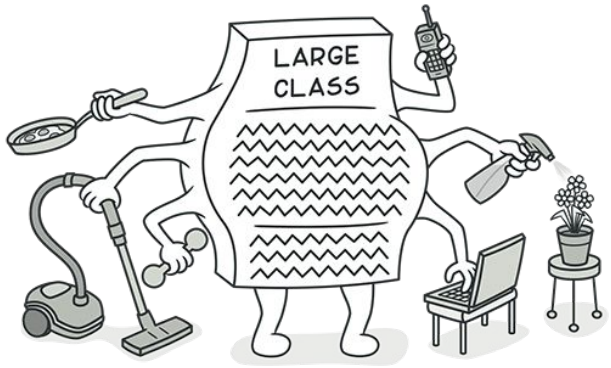
God Class

- The class that does everything.
- Centralizes logic that should be distributed across multiple classes.



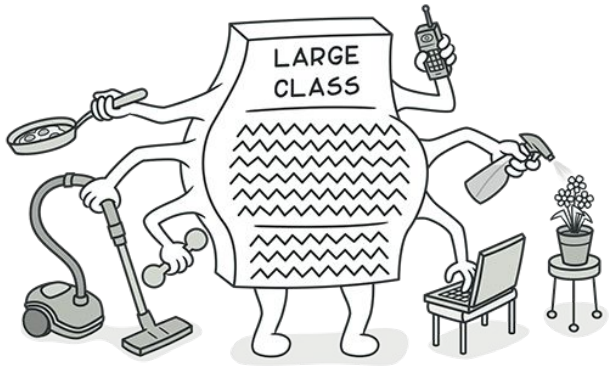
God Class

- The class that does everything.
- Centralizes logic that should be distributed across multiple classes.
- Breaks KISS and violates Single Responsibility Principle.



God Class

- The class that does everything.
- Centralizes logic that should be distributed across multiple classes.
- Breaks KISS and violates Single Responsibility Principle.
- Makes the code difficult to understand, maintain, and extend.



god-class

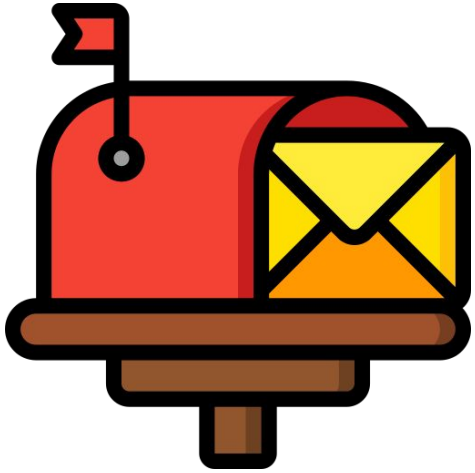
Data Clumps

- What's wrong with this code?

bad_data_clumps.ts

Data Clumps

- Two or more variables are always used in group.



Data Clumps

- Two or more variables are always used in group.
- It wouldn't make sense to use one of the variables by itself.

Data Clumps

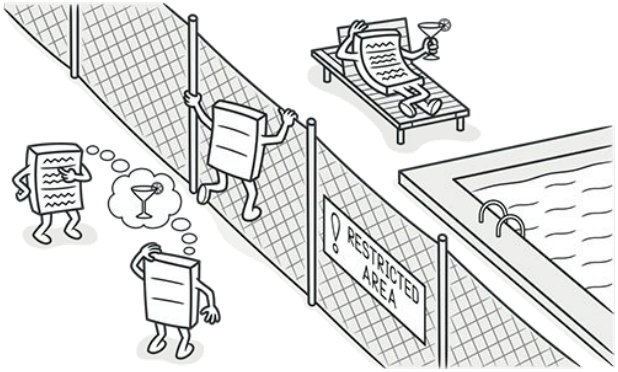
- Two or more variables are always used in group.
- It wouldn't make sense to use one of the variables by itself.
- Violation of DRY principle.

Data Clumps

- Two or more variables are always used in group.
- It wouldn't make sense to use one of the variables by itself.
- Violation of DRY principle.
- This group of variables should be extracted into a class.

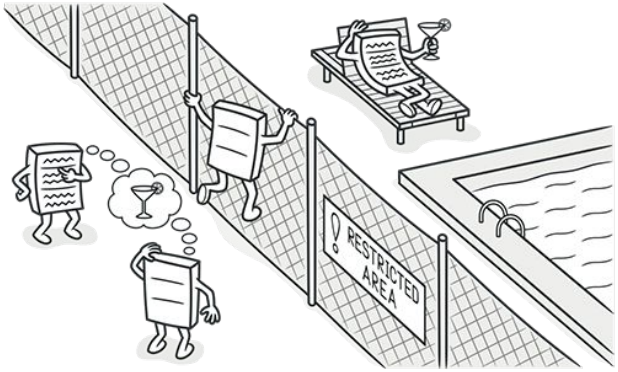
Feature Envy

- Occurs when a class frequently uses the methods, properties, or data of another class.



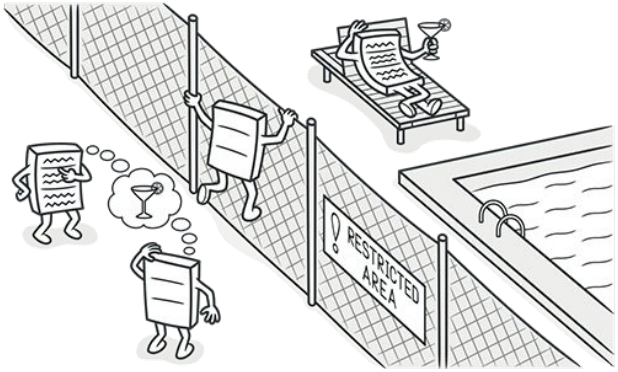
Feature Envy

- Occurs when a class frequently uses the methods, properties, or data of another class.
- It can lead to:



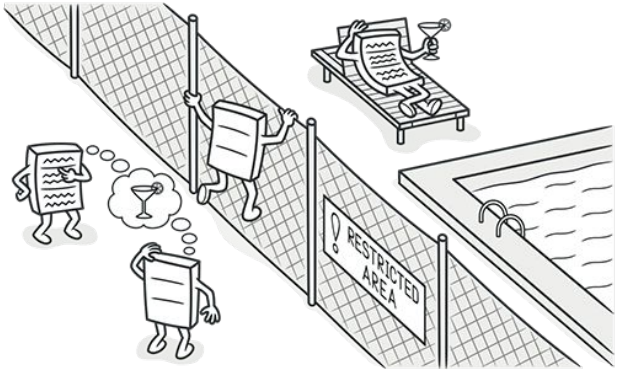
Feature Envy

- Occurs when a class frequently uses the methods, properties, or data of another class.
- It can lead to:
  - Tight coupling.



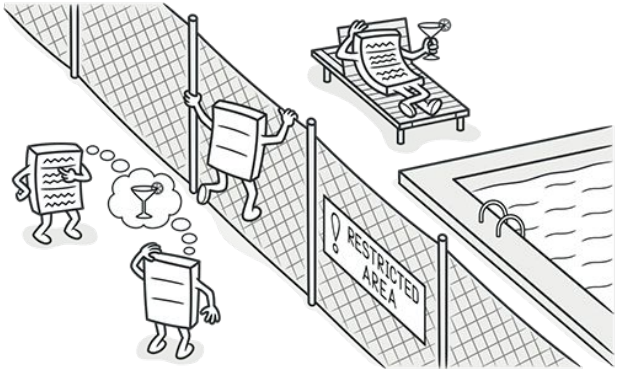
Feature Envy

- Occurs when a class frequently uses the methods, properties, or data of another class.
- It can lead to:
  - Tight coupling.
  - Reduced encapsulation.



Feature Envy

- Occurs when a class frequently uses the methods, properties, or data of another class.
- It can lead to:
  - Tight coupling.
  - Reduced encapsulation.
  - Code that is difficult to understand and maintain.



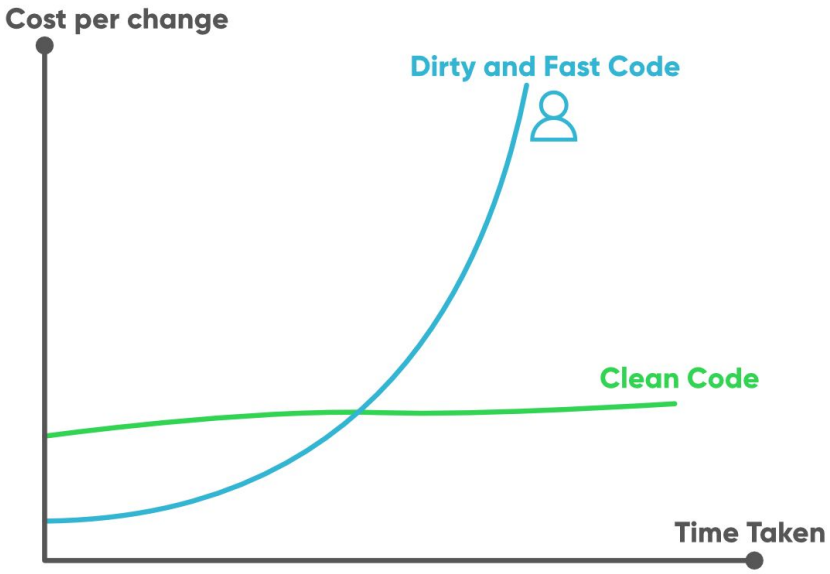
bad-feature-envy.ts

Refactoring

- From code smells to clean code.

Refactoring

- From code smells to clean code.



Bibliography

- Clean code book
- Code smells
- More code smells
- Encapsulation Article
- KISS Article
- YAGNI principle

Bibliography

- DRY principle
- Clean code classes examples
- Association vs Aggregation vs Composition vs Inheritance

Universidad de La Laguna

Questions??


Universidad
de La Laguna