Methodology

S.O.L.I.D Principles

- Single Responsibility
- Open Closed
- Liskov Substitution
- Interface Segregation
- Dependency Inversion

Robert C. Martin
(“Uncle Bob")

Design Principles and
Design Patterns (2000)

[S.O.L.I.D.] Single Responsibility

“every module, class or function in a computer program should have **responsibility over a single part** of that program's functionality, which it should encapsulate.”

based on the **principle of cohesion**, as described by Tom DeMarco (*Structured Analysis and System Specification*, 1979)

Related with the term **separation of concerns,** probably coined by Edsger W. Dijkstra in his 1974 paper "*On the role of scientific thought*"

```
function Employee(name, pos, hours) {

    this.name = name
    this.pos = pos
    this.hours = hours

    this.calculatePay = function() {
        ...

    }

    this.reportHours = function() {
        ...

    }

    this.save = function() {
        ...

    }
}
```


function EmployData(name, pos, hours) {
this.name = name
this.pos = pos
this.hours = hours
...
}
function PayCalculator(employData) {
this.employData = employData
this.calculatePay = function() {
...
}
}
function HourReporter(employData) {
this.employData = employData
this.reportHours = function() {
...
}
}
function EmployeeServer(employData) {
this.employData = employData
this.save = function() {
...
}
}

```
class UserRegistry {
    function createUser(email, password) {
        let salt = bcrypt.genSaltSync(10);
        let encriptedPassword = bcrypt.hashSync(password, salt);
        const newUser = new User(email, encriptedPassword);
        UserRepository.saveToDatabase(newUser);
    }
}
```

User registry + password encryption

```
class UserRegistry {
    function createUser(email, password) {
        let encriptedPassword = PasswordEncrypter.encrypt(password);
        const newUser = new User(email, encriptedPassword);
        UserRepository.saveToDatabase(newUser);
    }
}
```

User registry

```
class PasswordEncrypter {
    static function encrypt(password) {
        let salt = bcrypt.genSaltSync(10);
        let encriptedPassword = bcrypt.hashSync(password, salt);
        return encriptedPassword;
    }
}
```

Password encryption

[S.O.L.I.D.] Open Closed

“Software entities (classes, modules, functions, and so on) should be open for extension, but closed for modification”

```
class Rectangle {
    width: number;
    heigth: number;
}
```

```
class AreaCalculator {
    function computeArea(shapes: Rectangle[]) {
        let area = 0;
        for (let shape of shapes) {
          area += (shape.width * shape.heigth)
        }
        return area;
    }
}
```

New requirement: also, triangles

```
class AreaCalculator {
    function computeArea(shapes: Rectangle[]) {
        let area = 0;
        for (let shape of shapes) {
          if (typeof shape == 'Rectangulo') {
            area += (shape.width * shape.heigth)
          }
          if (typeof shape == 'Triangulo') {
            area += (shape.width * shape.heigth)/2
          }
        }
        return area;
    }
}
```

Broken Open Closed Principle

```
interface IShape {
    function area(): number;
}

class Rectangle implements IShape {
    width: number;
    heigth: number;

    function area() {
        return this.width * this.heigth;
    }
}

class Triangle implements IShape {
    width: number;
    heigth: number;

    function area() {
        return this.width * this.heigth / 2;
    }
}
```

```
class AreaCalculator {
    function computeArea(shapes: IShape[]) {
        let area = 0;
        for (let shape of shapes) {
          area += shape.area();
        }
        return area;
    }
}
```

```
let allowedRoles = ["ceo", "cto", "cfo", "staff"]

function checkPrivilege(employee) {
    if (allowedRoles.includes(employee.role)) {
        // employee has privilege
        return true;
    } else {
        return false
    }
}

function addRoles(role){
    allowedRoles.push(role)
}
```

[S.O.L.I.D.] Liskov Substitution

“To build software systems from interchangeable parts, those parts must adhere to a contract that allows those parts to be substituted one for another.”

If it Looks Like A Duck,
Quacks Like A Duck,
But Needs Batteries

You Probably Have
The Wrong Abstraction

```
class Duck {
    function fly() {}
    function swim() {}
    function cuack() {}
}

class RubberDuck extends Duck {
    function fly() {
        throw new Error();
    }
    function swim() {
        console.log('le swim');
    }
    function cuack() {
        console.log('le cuack');
    }
}
```

```
class DuckProcesser {
    function makeDucksFly(ducks: Duck[]) {
        for (let duck of ducks) {
          try {
            duck.fly();
          } catch(error) {
            console.log('RubberDuck cant fly');
          }
        }
    }
}
```

Extra logic in the use of the child class

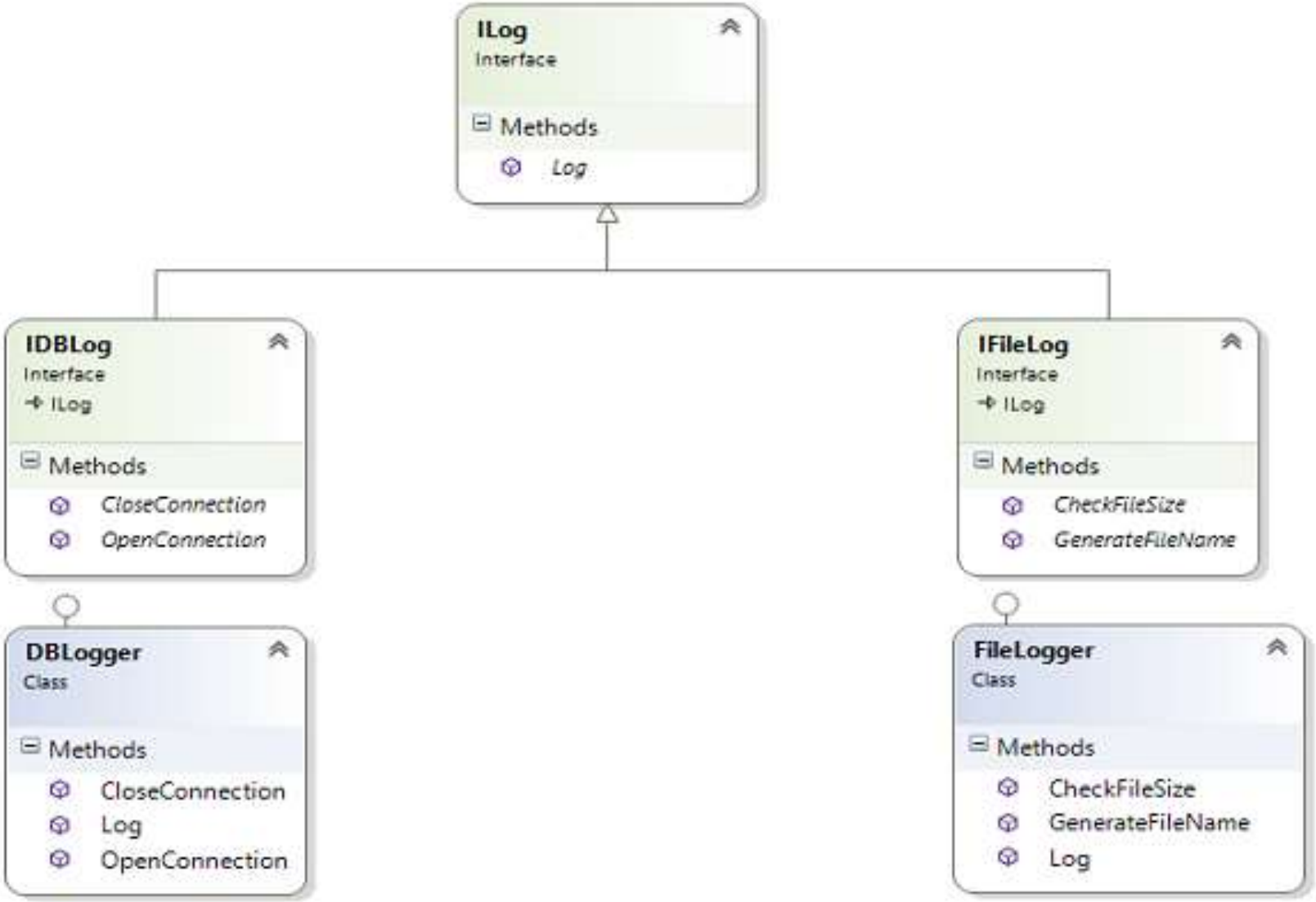
Liskov's Principle Violation

```
interface IFly {
  function fly(): void;
}

interface ISwim {
  function swim(): void;
}

interface ICuack {
  function cuack(): void;
}

class Duck implements IFly, ISwim, ICuack {
    function fly() {}
    function swim() {}
    function cuack() {}
}

class RubberDuck implements ISwim, ICuack {
    function swim() {
        console.log('le swim');
    }
    function cuack() {
        console.log('le cuack');
    }
}
```

```
class DuckProcesser {
    //No podremos pasar patos de goma aqui
    function makeDucksFly(ducks: IFly[]) {
        for (let duck of ducks) {
          duck.fly();
        }
    }
}
```

[S.O.L.I.D] Interface Segregation

“Many client-specific interfaces are better than one general-purpose interface”.
This principle advises software designers to avoid depending on things that they don’t use.

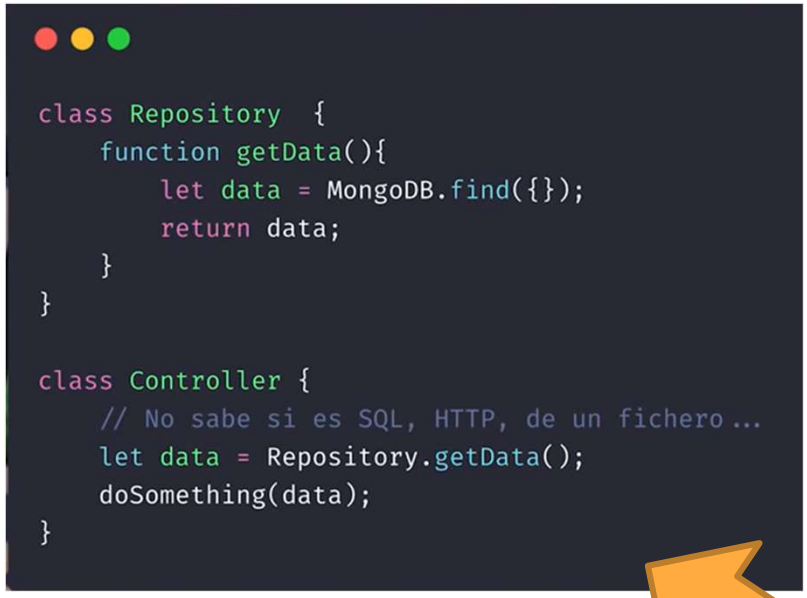

ILog
Interface
Methods
Log
IDBLog
Interface
ILog
Methods
CloseConnection
OpenConnection
IFileLog
Interface
ILog
Methods
CheckFileSize
GenerateFileName
DBLogger
Class
Methods
CloseConnection
Log
OpenConnection
FileLogger
Class
Methods
CheckFileSize
GenerateFileName
Log

```
class User {
    constructor(user) {
        this.user = user
        this.initiateUser()
    }
    initiateUser() {
        this.name = this.user.name
        this.getMenu()
    }
}
const user = new User({ userProperties, getMenu() { } })
```

```
class User {
    constructor(user) {
        this.user = user
        this.initiateUser()
        this.setupOptions = user.options
    }

    initiateUser() {
        this.name = this.user.name
        this.setupOptions()
    }
}
const user = new User({ userProperties, options: { getMenu() }{}})
```

[S.O.L.I.D] Dependency Inversion

High-level modules should not depend on low-level modules. Both should depend on abstractions (e.g. interfaces). Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions.



```
class Repository  {
    function getData(){
        let data = MongoDB.find({});
        return data;
    }
}

class Controller {
    // No sabe si es SQL, HTTP, de un fichero ...
    let data = Repository.getData();
    doSomething(data);
}
```

Change in the uncoupled module that access to the DB

```
class Repository  {
    function getData(){
        let data = SQLite.query('SELECT * FROM data');
        return data;
    }
}

class Controller {
    // No sabe si es SQL, HTTP, de un fichero ...
    let data = Repository.getData();
    doSomething(data);
}
```

No effects in the controller

Without DIP

```
$.get("/address/to/data", function (data) {
    $("#thingy1").text(data.property1)
    $("#thingy2").text(data.property2)
})
```

With DIP

```
fillFromServer("/address/to/data", thingyView)
```

```
function fillFromServer(url, view) {
    $.get(url, function (data) {
        view.setValues(data)
    })
}
```

```
var thingyView = {
    setValues: function (data) {
        $("#thingy1").text(data.property1)
        $("#thingy2").text(data.property2)
    }
}
```

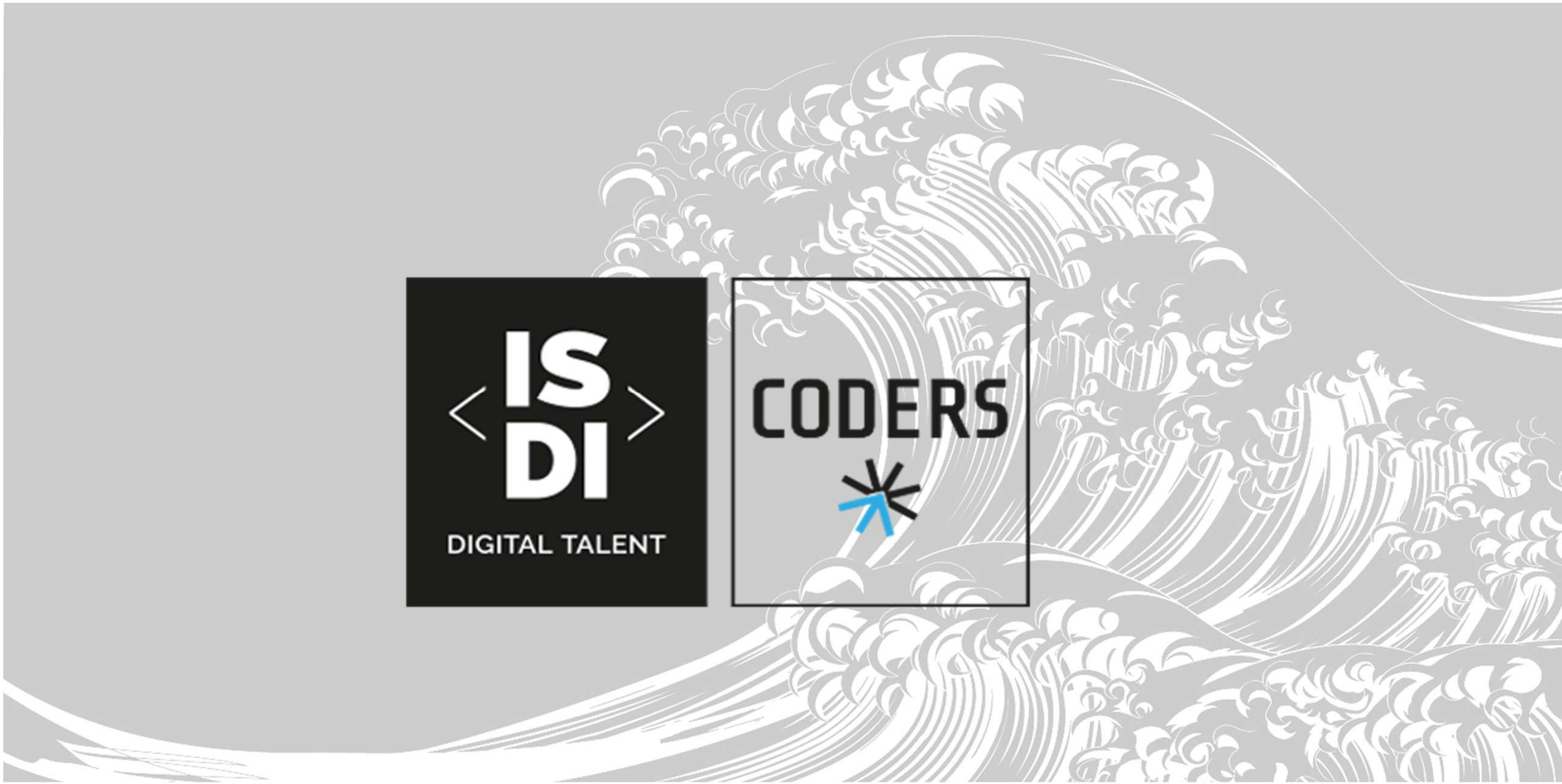
¿SOLID?

- Objectives: make code
  - More maintainable
  - Flexible (easy to change)
  - Extensible
  - more understandable
- It is only a tool with this objectives
  - If you don't get them or complicate the code unnecessarily, it stops making sense

Other principles

- GRASP (General Responsibility Assignment Software Patterns - object-oriented design)

- Don't repeat yourself (DRY)
- Keep it simple, stupid (KISS)
- You aren't gonna need it (YAGNI)


ISDI
DIGITAL TALENT
CODERS