

Norms of Object-oriented programming and SOLID

Authors:

Carlos Díaz Calzadilla

Sergio Guerra Arencibia



Index

1. OOP
 - a. Objects
 - b. Abstraction
 - c. Encapsulation
 - d. Inheritance
 - e. Polymorphism
2. OOP principles
3. SOLID
 - a. SRP
 - b. OCP
 - c. LSP
 - d. ISP
 - e. DIP



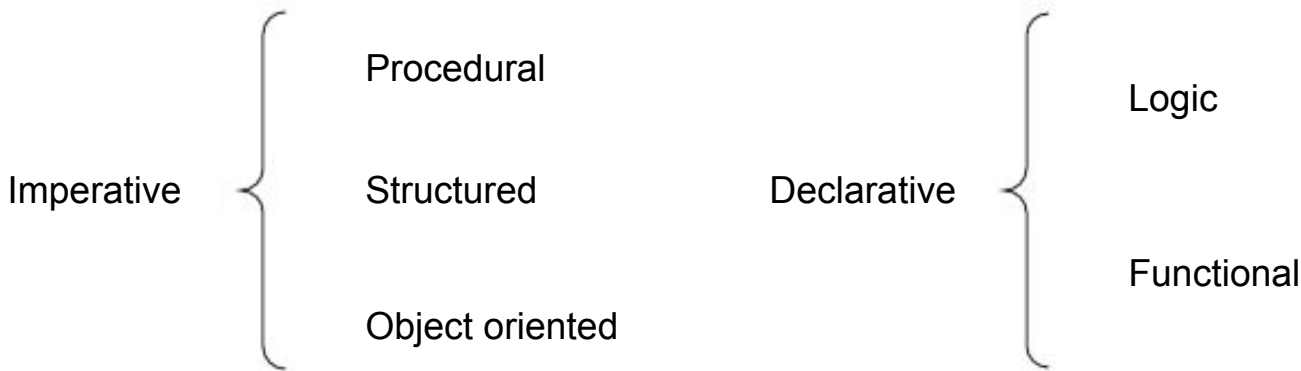


Programming paradigm

- What are they?

“A programming paradigm is a style, or “way,” of programming.”

- Some history...





OOP

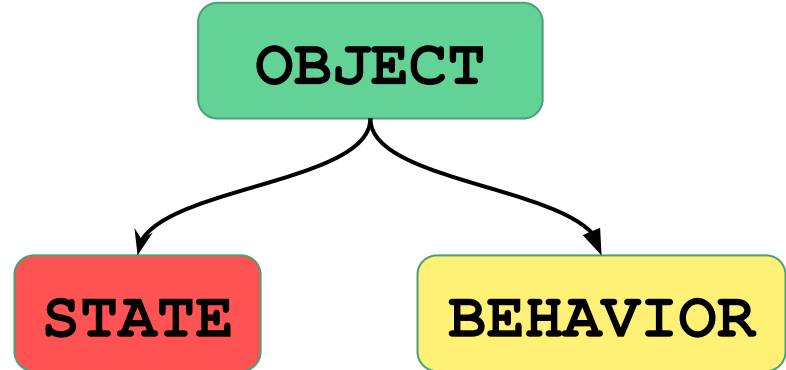
What is it?

“A programming paradigm based on the concept of objects”

We need to understand what “object” means

An objects has two characteristics:

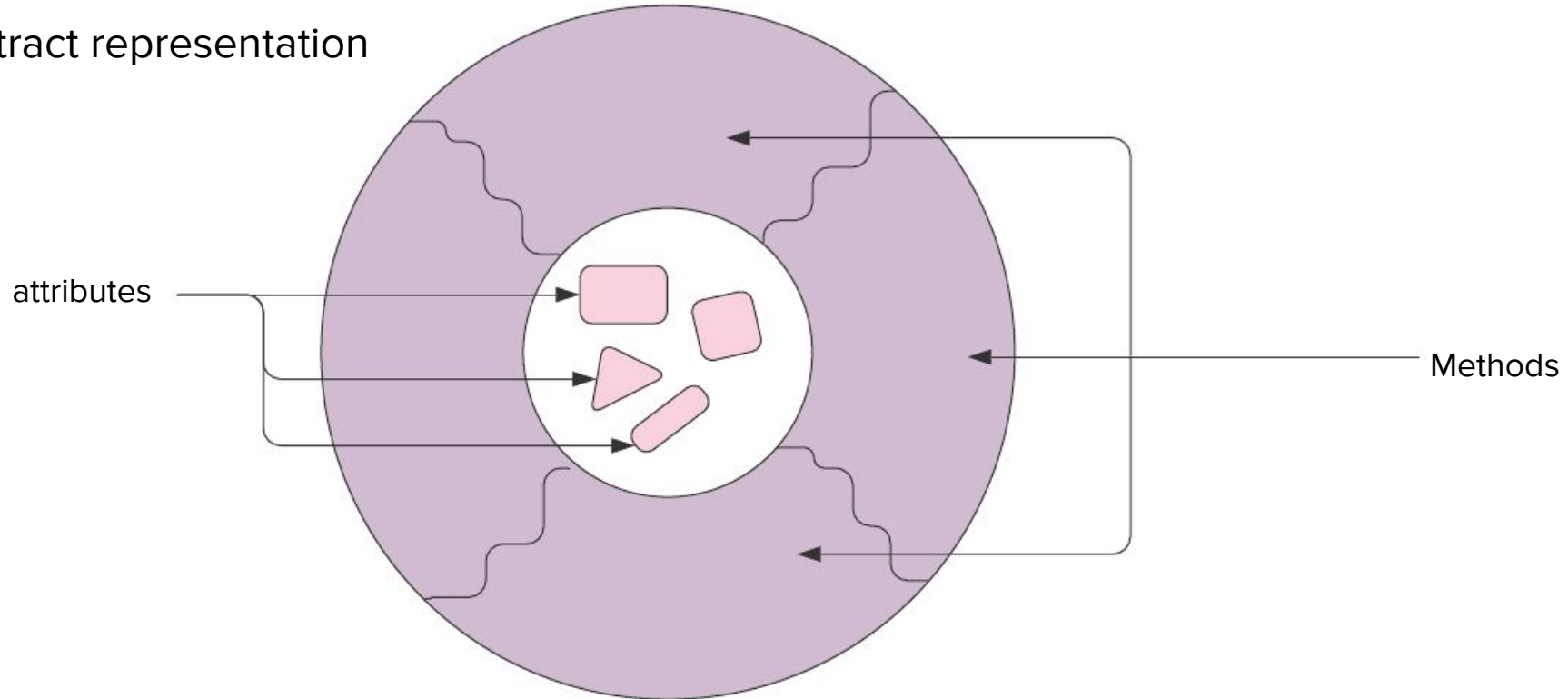
- They all have state
- They all have behavior





Objects

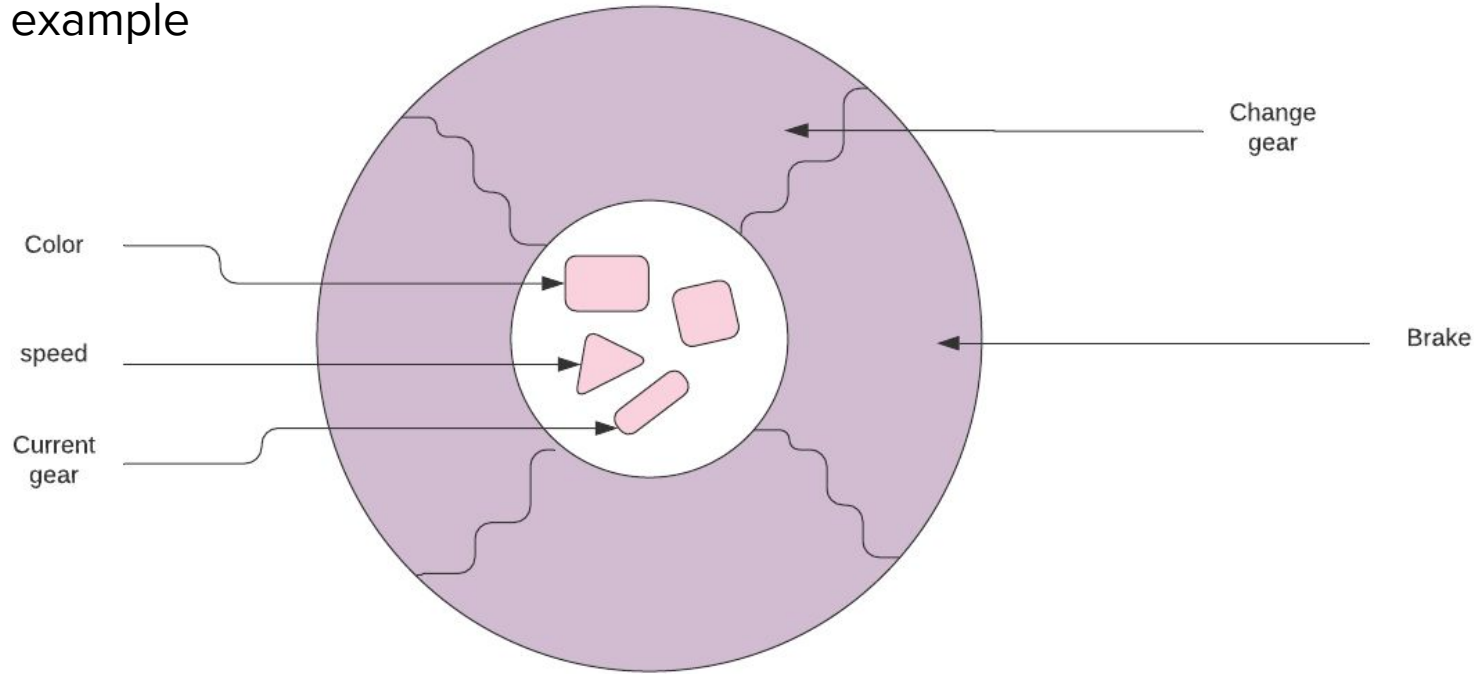
Abstract representation





Objects

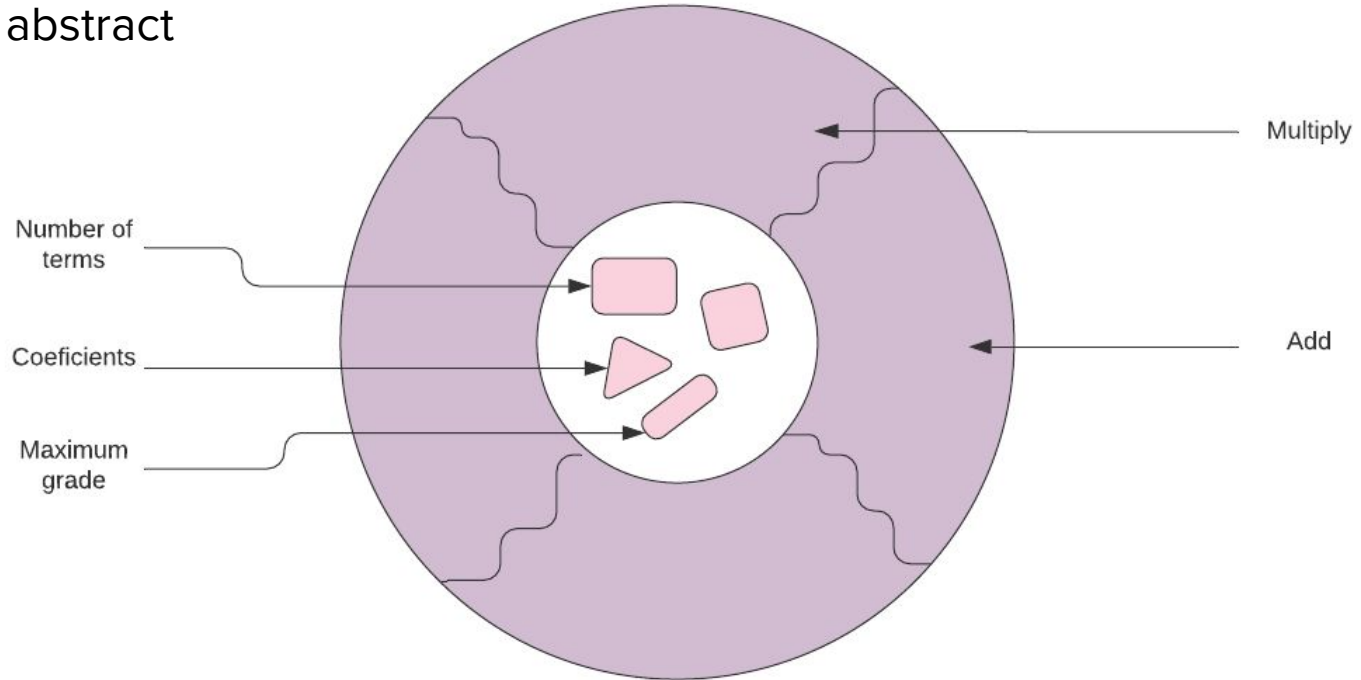
Bike, an object example





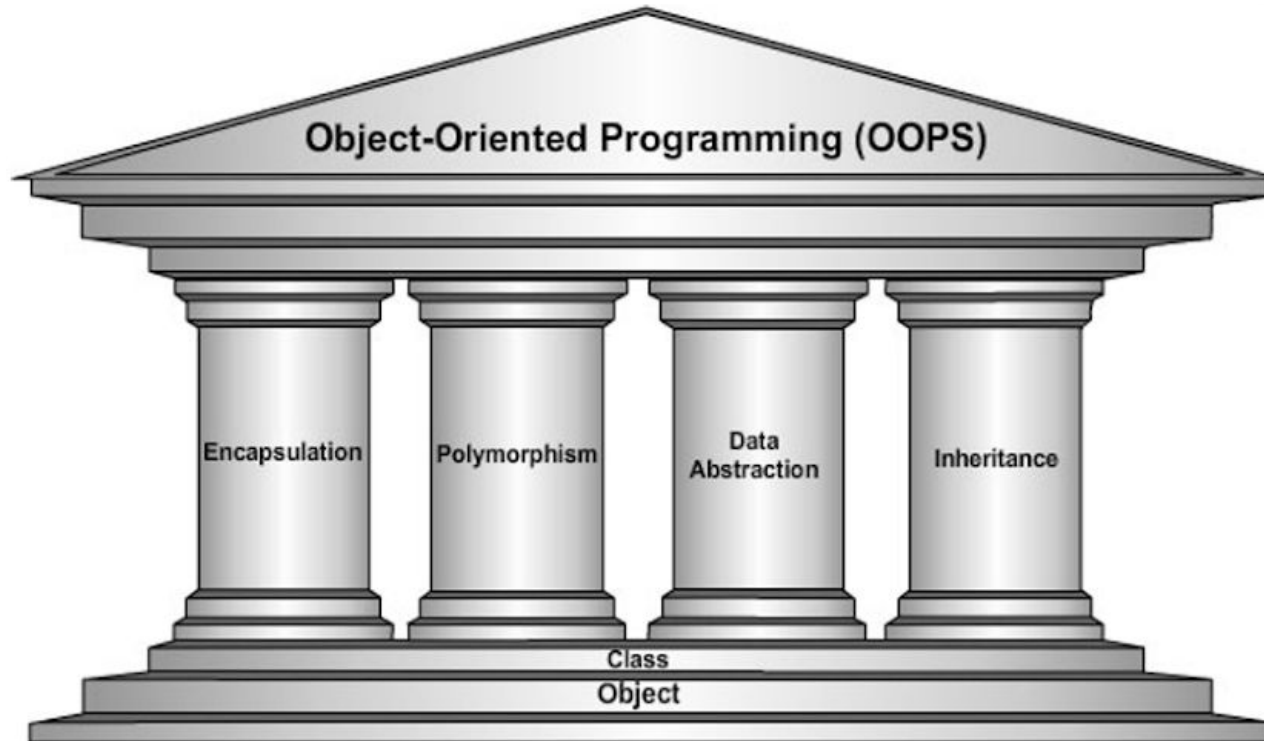
Objects

Polynomial, an abstract
example





OOP principles





Abstraction

- Definition

“The process of removing physical, spatial, or temporal details or attributes in the study of objects or systems to focus attention on details of greater importance”



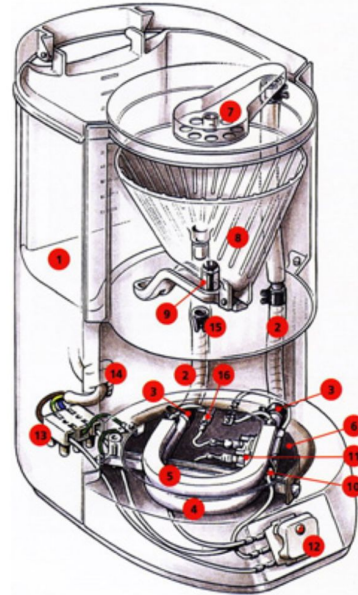
You focus on...

- Provide water
- Provide coffee
- Select the kind of coffee you want
- Switch it on

You abstract from...

- Temperature
- Amount of coffee to use
- How the water is heated
- etc...

Abstraction





Abstraction

What does abstraction offer us?

- Manage complexity
- Implement more complex logic

Abstraction in OOP is exactly the same!

Interface

“ Shared boundary by which two systems (human or computer) can communicate and exchange information “

“ Limited sets of functions or bindings that provide useful functionality at a more abstract level, hiding their precise implementation ”



Abstraction

Summary:

Working with objects through their external and visible part (interface) allows us to forget and worry about their internal details, reducing the complexity of the system (abstraction)



An example of abstraction

```
let firstPoint = {x:1, y:1};
let secondPoint = {x:5, y:3};
let thirdPoint = {x:7, y:1};
let triangle = [firstPoint, secondPoint, thirdPoint];

// Traslation of 5 in X
for (let pointIter = 0; pointIter < triangle.length;
pointIter++) {
    triangle[pointIter].x += 5;
}
// Traslation of 2 in Y
for (let pointIter = 0; pointIter < triangle.length;
pointIter++) {
    triangle[pointIter].y += 2;
}
```

An example of abstraction

```
// 90 degree rotation
for (let pointIter = 0; pointIter < triangle.length;
pointIter++) {
  let newPoint = {};
  newPoint.x = -(triangle[pointIter].y)
  newPoint.y = triangle[pointIter].x;
  triangle[pointIter] = newPoint;
}
```



An example of abstraction

```
class Triangle {  
  constructor(points) {  
    this.points = points;  
  }  
  
  traslationX (distance) {  
    for (let pointIter = 0; pointIter <  
      this.points.length; pointIter++) {  
      this.points[pointIter].x += distance;  
    }  
  }  
  
  traslationY (distance) {  
    for (let pointIter = 0; pointIter <  
this.points.length;  
      pointIter++) {  
      this.points[pointIter].y += distance;  
    }  
  }  
}
```

```
rotation() {  
  let newPoint = {};  
  newPoint.x = -(this.points[pointIter].y)  
  newPoint.y = this.points[pointIter].x;  
  this.points[pointIter] = newPoint;  
  }  
}
```



An example of abstraction

```
let firstPoint = {x:1, y:1};  
let secondPoint = {x:5, y:3};  
let thirdPoint = {x:7, y:1};  
let points = [firstPoint, secondPoint,  
thirdPoint];  
let triangle = new Triangle(points);  
  
triangle.traslationX(5);  
triangle.traslationY(2);  
triangle.rotation();
```

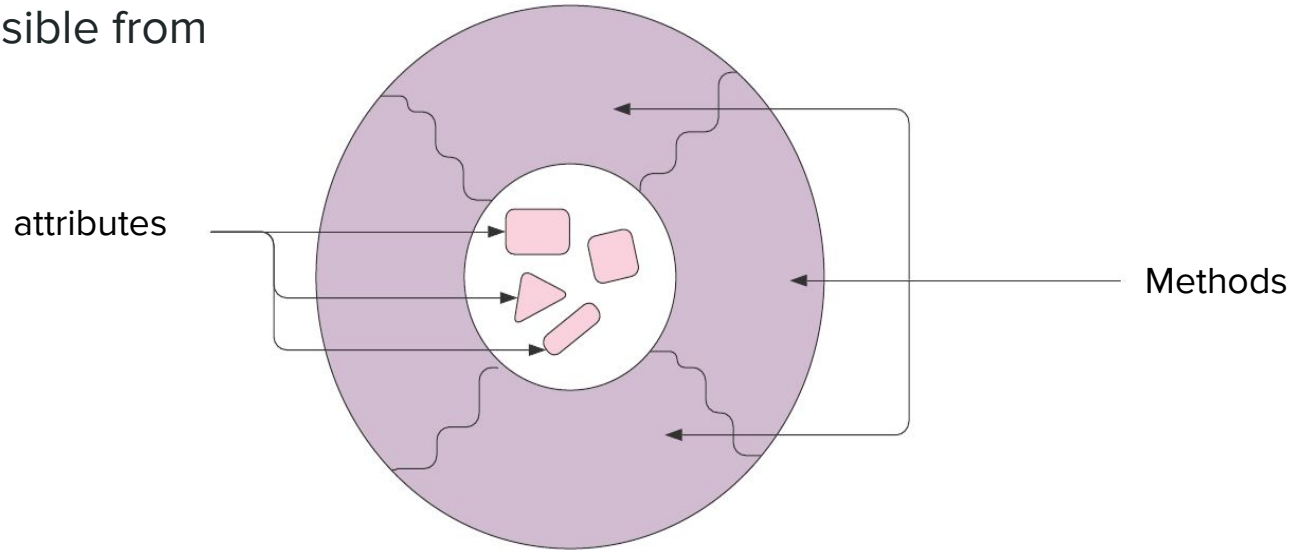
This allows us to build more complex code in our abstraction level



Encapsulation

Encapsulation is used to hide the values or state of a structured data object inside of it

Attributes are NOT accessible from the outside





Encapsulation

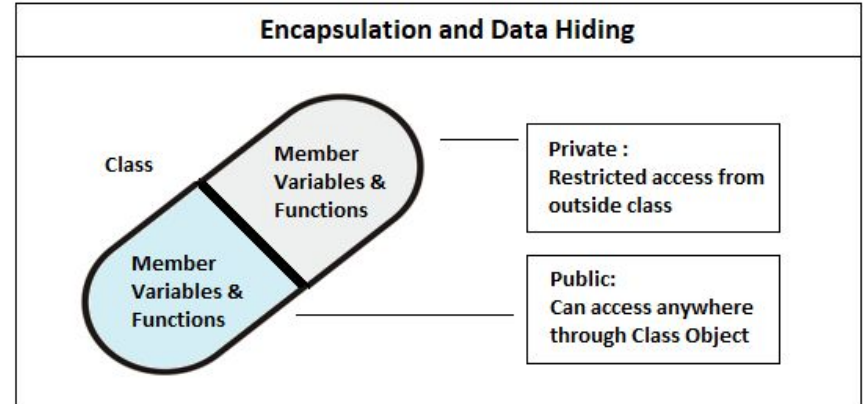




Encapsulation

Benefits?

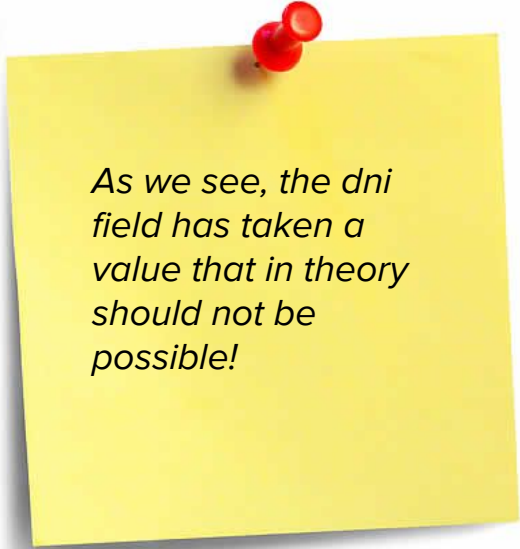
- Protects the integrity of the data
- The ability to modify your code, without affecting the code of others who used your code
- Helps us to make flexible code





Encapsulation example - Integrity

```
class Person {  
    public:  
        std::string name;  
        int dni;  
  
        person(std::string, int);  
        ~person();  
};
```



As we see, the dni field has taken a value that in theory should not be possible!

```
person panadero(Juan, 51100000);  
panadero.dni = -1;
```



Encapsulation example - compatibility

```
class Person {  
    public:  
        std::string name;  
        int dni;  
  
        person(std::string, int);  
        ~person();  
};
```

```
class Person {  
    public:  
        std::string name;  
        std::vector<int> dni;  
  
        person(std::string, int);  
        ~person();  
};
```

```
Person panadero(Juan, 51100000);  
panadero.dni = 51100001;
```



Encapsulation example - compatibility

```
class Person {  
    private:  
        std::string name;  
        int dni;  
  
    public:  
        person(std::string, int);  
        ~person();  
  
        void setName();  
        void setDni(int);  
        std::string getName();  
        int getDni();  
};
```

```
void Person::setDni(int newDni) {  
    if (newDni < 0) {  
        throw "DNI must be greater than 0";  
    }  
    else {  
        dni = newDni;  
    }  
}
```

```
Person panadero(Juan, 51100000);  
panadero.setDni(-1);          // Error  
panadero.setDni(511000000)  
    // Funciona con cualquier implementación
```

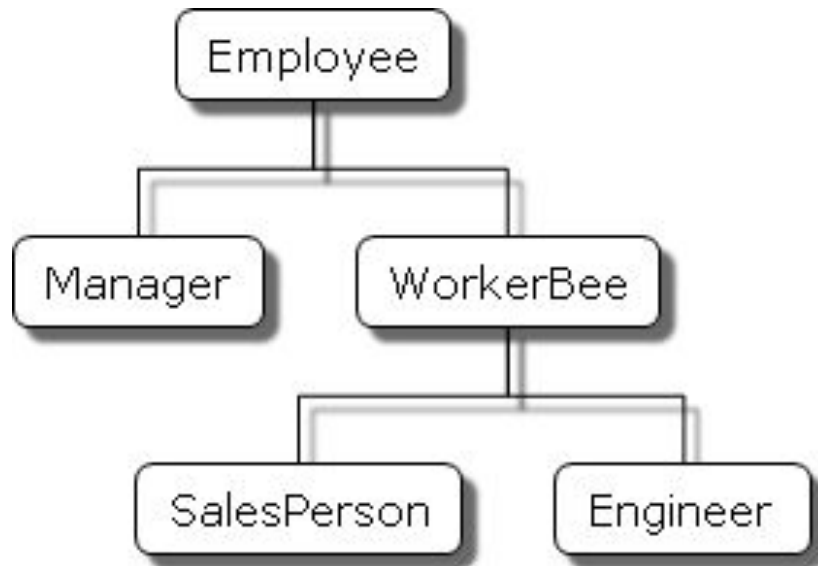


Inheritance

We already know what inheritance is

It has different benefits:

- Code reuse
- Changes propagation



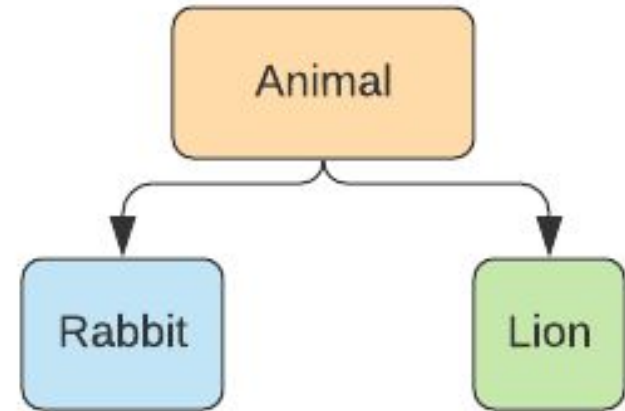


Inheritance example

We want to create some classes which represents animals

Each animal do different things (hide, bite, etc...) but they all do some things the same (run, make some noise, for example)

So we can design the following class hierarchy





Inheritance example

This is the father class.

It has 2 methods

- run
- stop

Now we want to implement
two new animals
Rabbit and Lion

```
class Animal {  
  constructor (name) {  
    this.speed = 0;  
    this.name = name;  
  }  
  run (speed) {  
    this.speed = speed;  
    console.log(`${this.name} runs with speed  
    ${this.speed}.`);  
  }  
  stop () {  
    this.speed = 0;  
    console.log(`${this.name} stands still.`);  
  }  
}
```



Inheritance example

```
class Rabbit {  
  constructor (name) {  
    this.speed = 0;  
    this.name = name;  
  }  
  run (speed) {  
    this.speed = speed;  
    console.log(`${this.name} runs with  
speed ${this.speed}.`);  
  }  
  stop () {  
    this.speed = 0;  
    console.log(`${this.name} stands  
still.`);  
  }  
  hide () {  
    console.log(`${this.name} hides!`);  
  }  
}
```

```
class Lion {  
  constructor (name) {  
    this.speed = 0;  
    this.name = name;  
  }  
  run (speed) {  
    this.speed = speed;  
    console.log(`${this.name} runs with  
speed ${this.speed}.`);  
  }  
  stop () {  
    this.speed = 0;  
    console.log(`${this.name} stands  
still.`);  
  }  
  bite () {  
    console.log(`${this.name} has bitten  
you!!`);  
  }  
}
```



Inheritance example

What if we use inheritance?

```
class Rabbit extends Animal {  
  hide(name) {  
    console.log(`${this.name}  
      hides!`);  
  }  
}
```

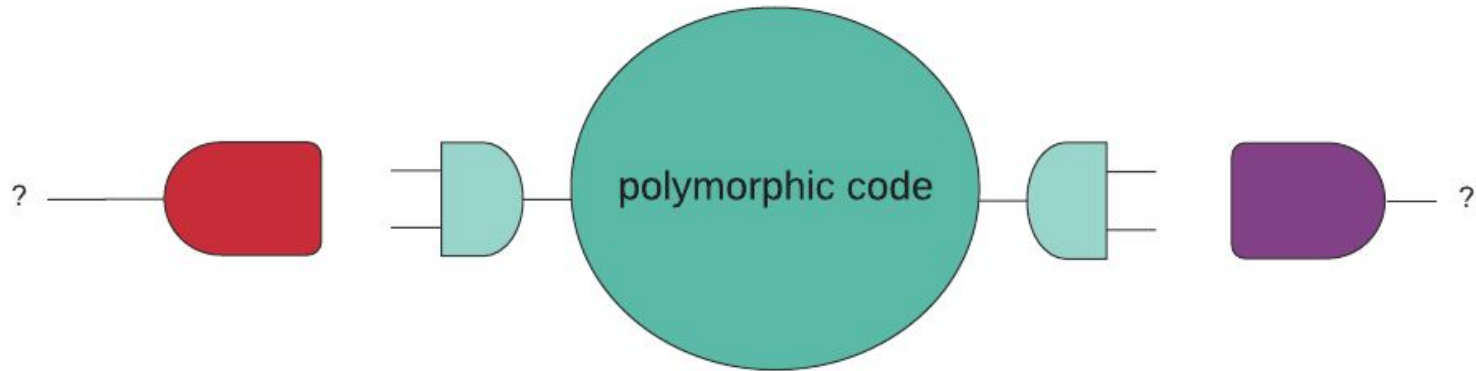
```
class Lion extends Animal {  
  bite(name) {  
    console.log(`${this.name} has bitten you!!`);  
  }  
}
```

The amount of code reused is obvious



Polymorphism

Objects of different types can be accessed through the same interface if they both have it





Shapes example

```
const PI = 3.1416;

class Circle {
  constructor(radius) {
    this.radius = radius;
  }

  area() {
    return (PI *
Math.pow(this.radius, 2));
  }

  perimeter() {
    return (2 * PI * this.radius);
  }
}
```

```
class Square {
  constructor(side) {
    this.side = side;
  }

  area() {
    return
(Math.pow(this.side, 2));
  }

  perimeter() {
    return (this.side * 4);
  }
}
```



Shapes example

They have the same interface, so...

```
function shapeData(shape) {  
  console.log(shape.area());  
  console.log(shape.perimeter());  
}  
  
let testSquare = new Square(5); // 25 20  
let testCircle = new Circle(3); // 28.27 18.84  
shapeData(testSquare);  
shapeData(testCircle);
```

The *shapeData* function works with both of them!



Inheritance is really that useful?

It helps us in a lot of ways, but it is more controversial than encapsulation or polymorphism

Encapsulation and polymorphism - used to separate pieces of code from each other

Inheritance - ties classes together



DRY

Don't write duplicated code

If you have a block of code in more than two places (WET) consider making it a separate method



DRY
Don't Repeat Yourself

DRY

Why is not recommended to duplicate code?

- Maintainability
- Readability
- Better testing

These points become a cost reduction





DRY example

```
function myFunctionA (arr, factor)
{
  let len = arr.length;
  let min = Infinity;

  while (len--) {
    if (arr[len] < min) {
      min = arr[len];
    }
  }
  return (Math.cos(min) * factor);
}
```

```
function myFunctionB (arr) {
  let len = arr.length;
  let min = Infinity;

  while (len--) {
    if (arr[len] < min) {
      min = arr[len];
    }
  }
  let shortestPath = new distance(min);
  shortestPath.draw();
  return (shortestPath.time);
}
```



DRY example

```
function minimum (arr) {  
  let len = arr.length;  
  let min = Infinity;  
  
  while (len--) {  
    if (arr[len] < min) {  
      min = arr[len];  
    }  
  }  
  return (min);  
}
```

```
function myFunctionA (arr, factor)  
{  
  let min = min(arr);  
  
  return (Math.cos(min) * factor);  
}
```

```
function myFunctionB (arr) {  
  
  let min = min(arr);  
  
  let shortestPath = new distance(min);  
  shortestPath.draw();  
  return (shortestPath.time);  
}
```



Encapsulate what varies

The most important principle?

“the only constant is change”

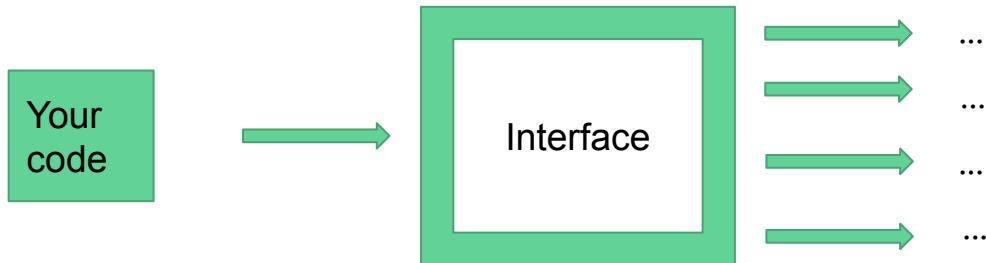
Summary:

Hide the potential variation behind an interface.

Your
code



Your
code





Composition over inheritance

Both define relationships between classes

is indifferent which one we use?

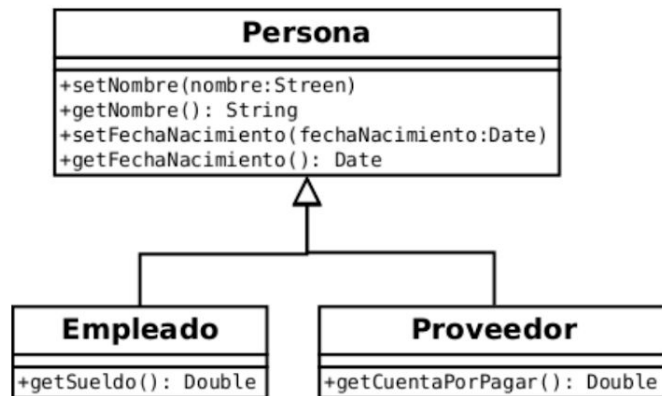
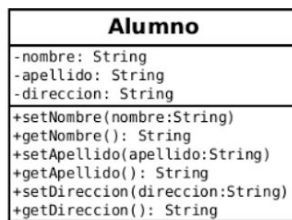
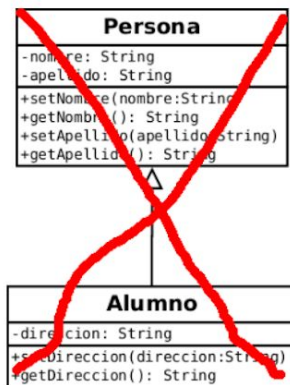
Inheritance produce...

- dependencies in the code
- Less readability, scalability and debugging.



Composition over inheritance

Examples of common cases where is better composition

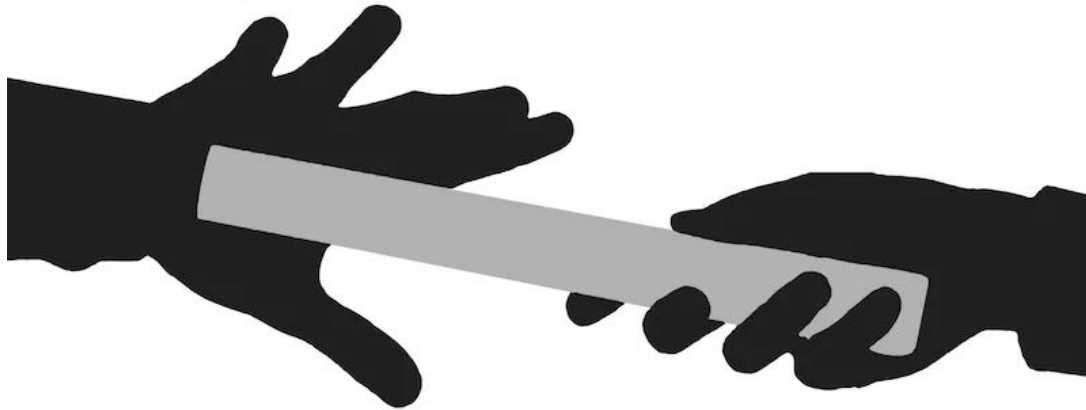




Delegation

Important principle that we already know

"Delegate part of your responsibility to another object that specialises in a part of what you need to accomplish."





SOLID

- ¿What is SOLID?





The Principles of SOLID - Class Design

The first five principles are principles of class design are:

- **SRP** (Single Responsibility Principle)
- **OCP** (Open Closed Principle)
- **LSP** (Liskov Substitution Principle)
- **ISP** (Interface Segregation Principle)
- **DIP** (Dependency Inversion Principle)



The Principles of SOLID - SRP



The *Single Responsibility Principle* is based on the fact that each of the classes in a program must focus on having a single task or responsibility



The Principles of SOLID - SRP





The Principles of SOLID - SRP

```
class UserSettings {  
    constructor(user) {  
        this.user = user;  
    }  
  
    changeSettings(settings) {  
        if (this.verifyCredentials()) {  
            // ...  
        }  
    }  
  
    verifyCredentials() {  
        // ...  
    }  
}
```



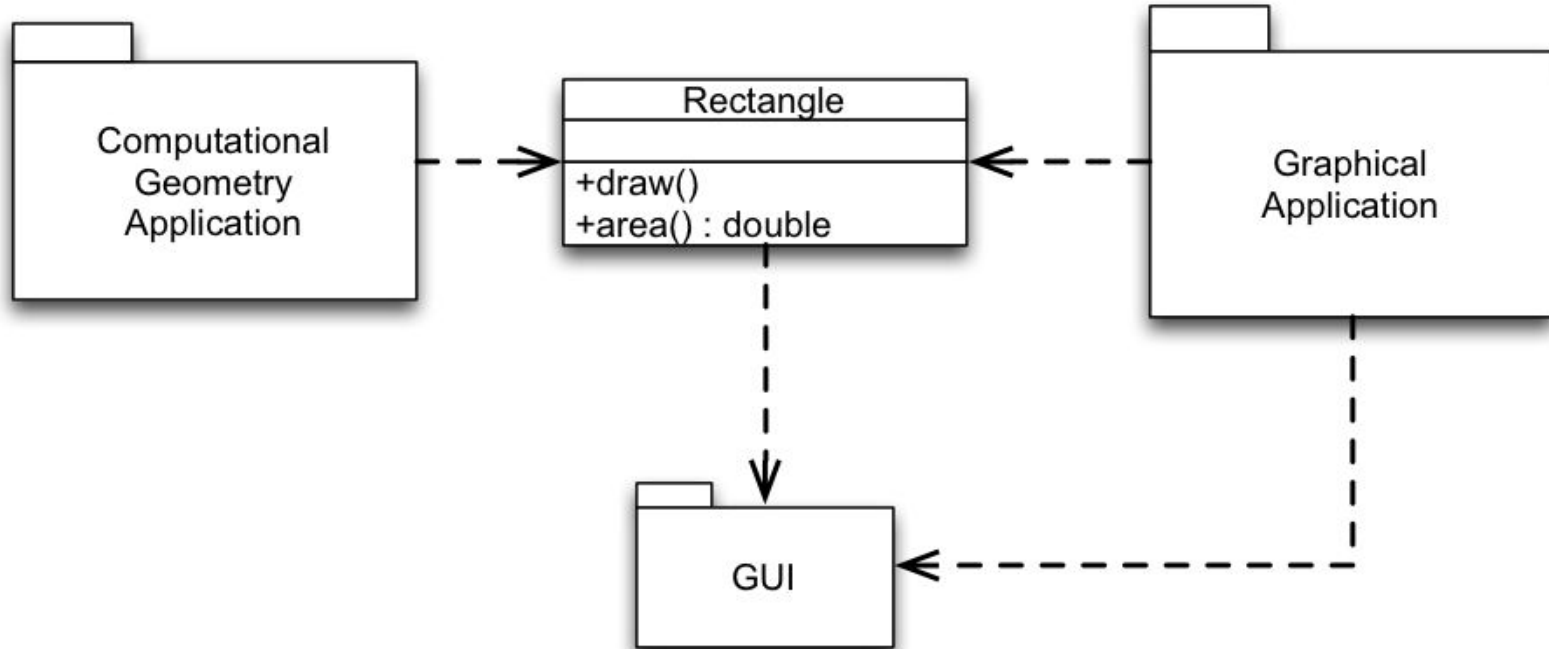
The Principles of SOLID - SRP

```
class UserAuth {  
    constructor(user) {  
        this.user = user;  
    }  
  
    verifyCredentials() {  
        // ...  
    }  
}
```

```
class UserSettings {  
    constructor(user) {  
        this.user = user;  
        this.auth = new UserAuth(user);  
    }  
  
    changeSettings(settings) {  
        if (this.auth.verifyCredentials()) {  
            // ...  
        }  
    }  
}
```

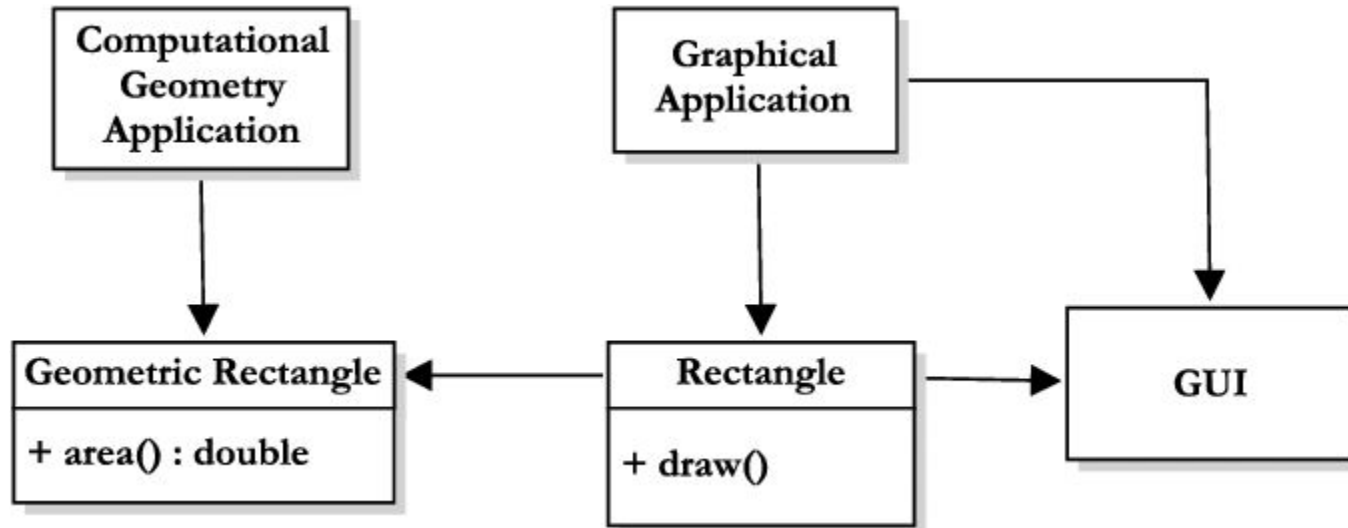


The Principles of SOLID - SRP

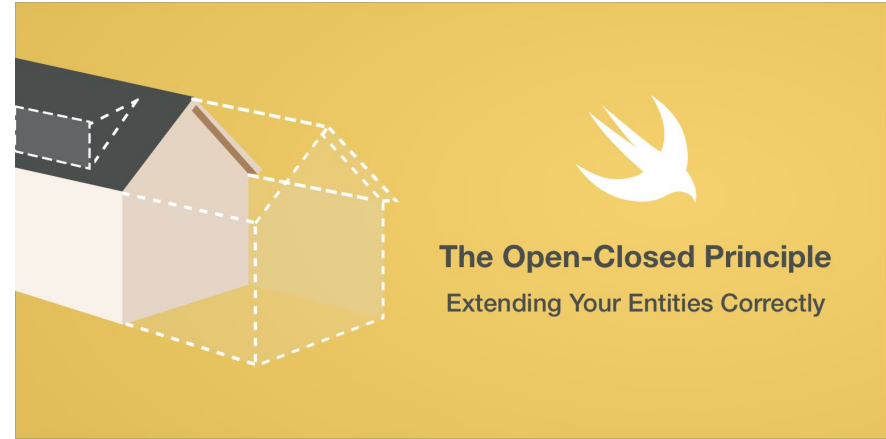
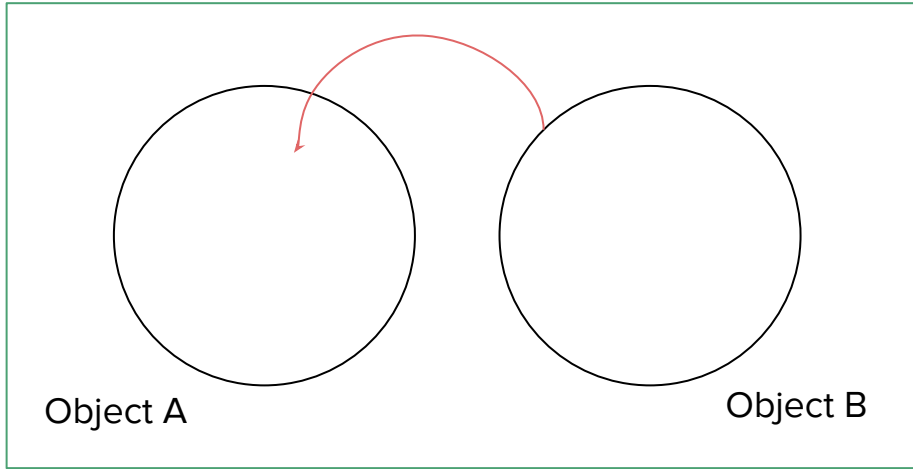




The Principles of SOLID - SRP



The Principles of SOLID - OCP



The *Open Closed Principle* is where the software elements(classes, modules, functions,etc.) should be open for extension, but closed for modification



The Principles of SOLID - OCP

```
class Pdf {  
    constructor(name, size) {  
        this.name = name;  
        this.size = size;  
    }  
  
    // ...  
}
```

```
class Png {  
    constructor(name) {  
        this.name = name;  
    }  
  
    // ...  
}
```

```
class Printer {  
    printFile(file) {  
        if (file instanceof Pdf) {  
            // Print Pdf...  
        } else if (file instanceof Png) {  
            // Print Png...  
        }  
    }  
}
```



The Principles of SOLID - OCP

```
class Printable {  
    print() {  
        // ...  
    }  
}
```

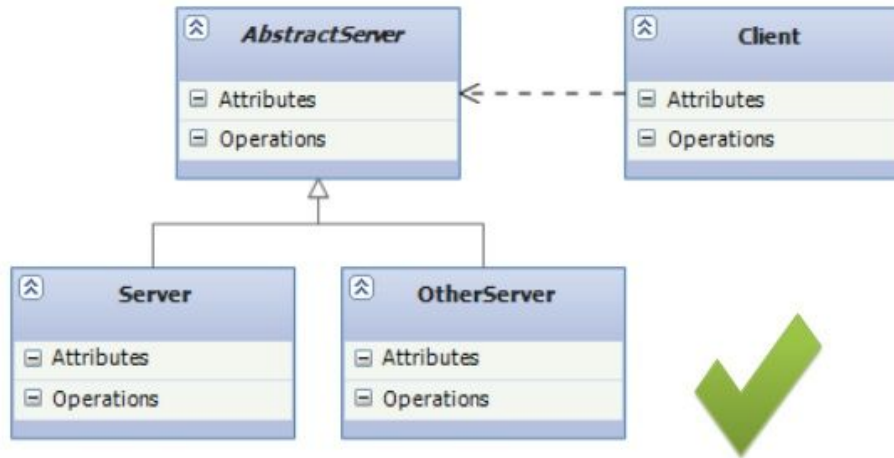
```
class Printer {  
    printFile(file)  
    {  
        file.print();  
    }  
}
```

```
class Pdf extends  
Printable {  
    constructor(name, size)  
    {  
        super();  
        this.name = name;  
        this.size = size;  
    }  
  
    // Override  
    print() {  
        // ...  
    }  
}
```

```
class Png extends  
Printable {  
    constructor(name) {  
        super();  
        this.name = name;  
    }  
  
    // Override  
    print() {  
        // ...  
    }  
}
```



The Principles of SOLID - OCP



The Principles of SOLID - LSP

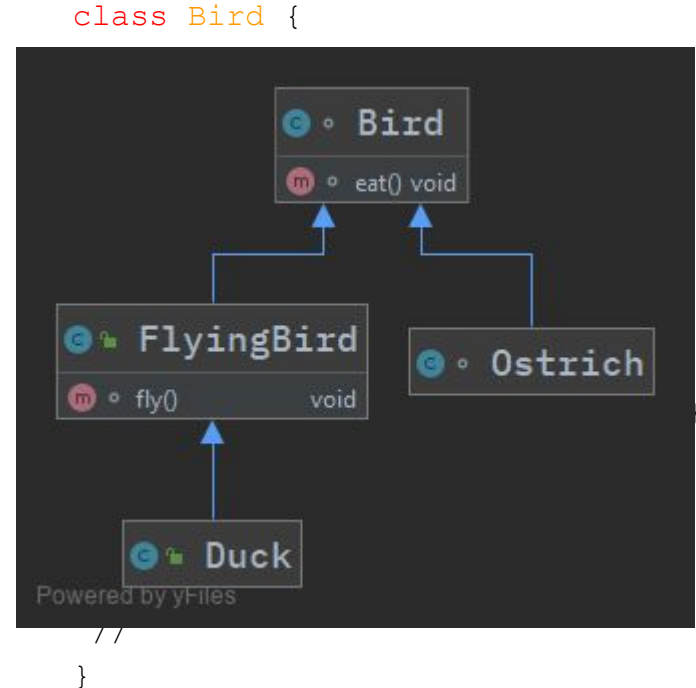


The *Liskov Substitution Principle* is the ability to replace any instance of a parent class with an instance of one of its child classes without negative side effects



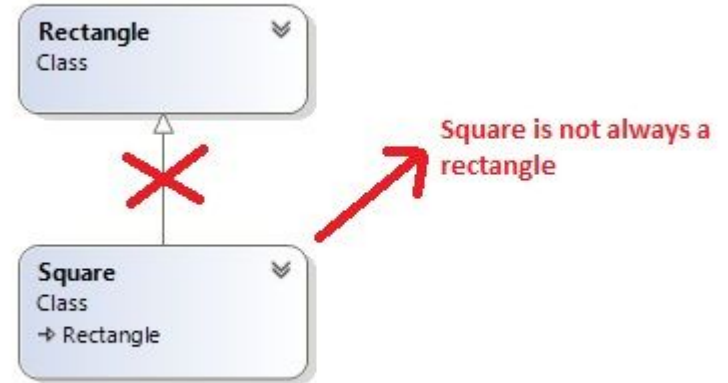
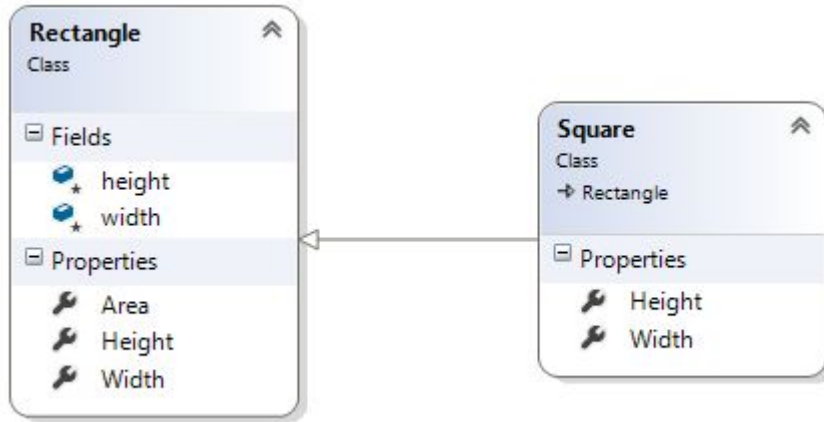
The Principles of SOLID - LSP

```
class Bird {  
    fly() {  
        //  
    }  
}  
class Eagle extends Bird {  
    //  
}  
class Ostrich extends Bird {  
    //  
}
```



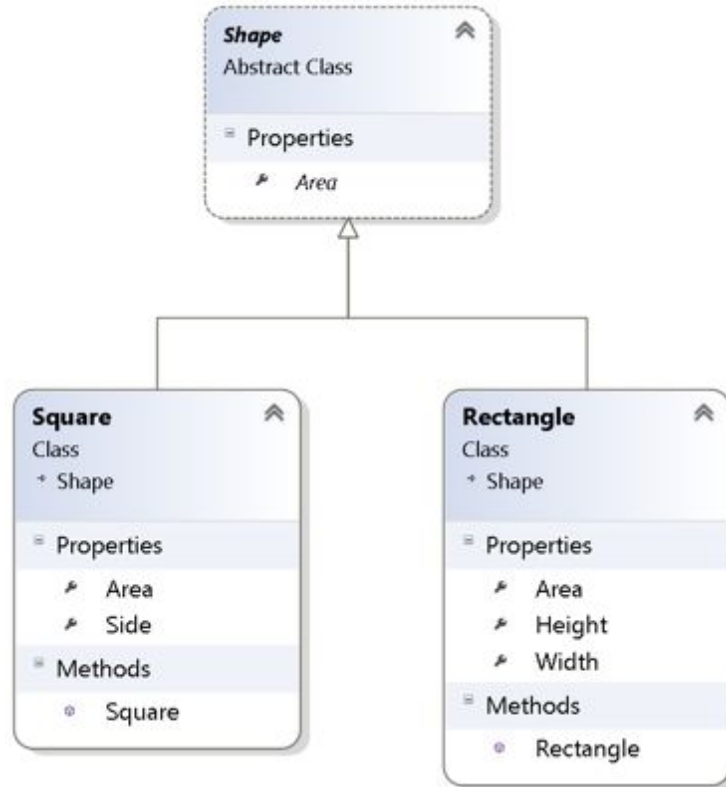


The Principles of SOLID - LSP



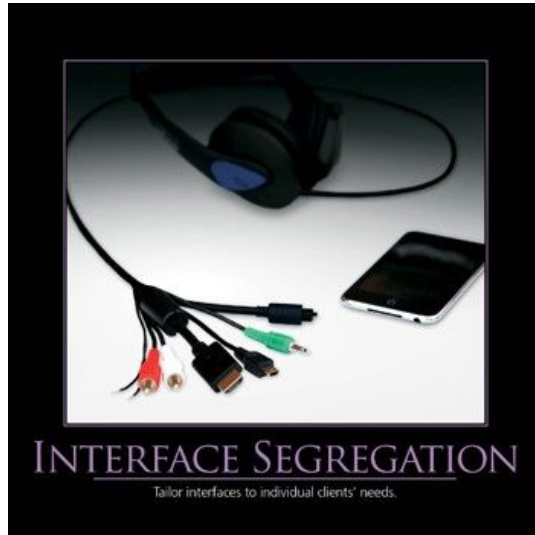


The Principles of SOLID - LSP





The Principles of SOLID - ISP



The *Interface Segregation Principle*, the code should not be forced to depend on methods that it doesn't use



The Principles of SOLID - ISP

```
class IColors {  
    paintRed();  
    paintYellow();  
    paintBlue();  
}
```

```
class Red extends IColors{  
    paintRed() {  
        // implementation details to paint red  
    }  
    paintYellow() {  
        // this class doesn't need this method, but  
        still needs an empty implementation.  
    }  
    paintBlue() {  
        // this class doesn't need this method, but  
        still needs an empty implementation.  
    }  
}
```



The Principles of SOLID - ISP

```
class IColors {  
    paint();  
}  
class IRed {  
    paintRed();  
}  
class IYellow {  
    paintYellow();  
}  
class IBlue {  
    paintBlue();  
}
```

```
class Red extends IRed {  
    paintRed() {  
        //...  
    }  
}  
class Yellow extends IYellow {  
    paintYellow() {  
        //...  
    }  
}
```



The Principles of SOLID - ISP

```
class Blue extends IBlue {  
    paintBlue() {  
        //...  
    }  
}  
class CustomColor extends IColors {  
    paint() {  
        //...  
    }  
}
```



The Principles of SOLID - DIP



Dependency Inversion Principle

Would you solder a lamp directly to the electrical wiring in a wall?

The *Dependency Inversion Principle* said that the high level objects should not depend on low level implementations

The Principles of SOLID - DIP

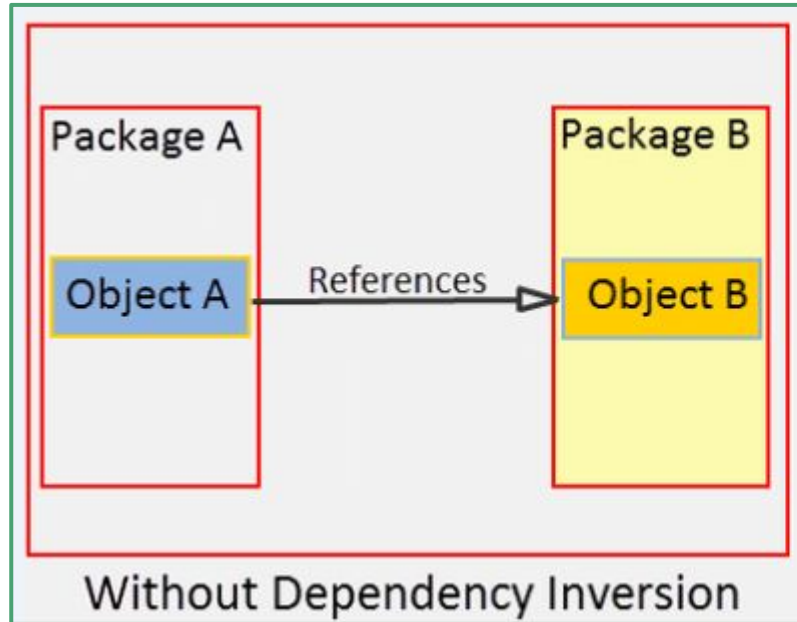


The Principles of SOLID - DIP



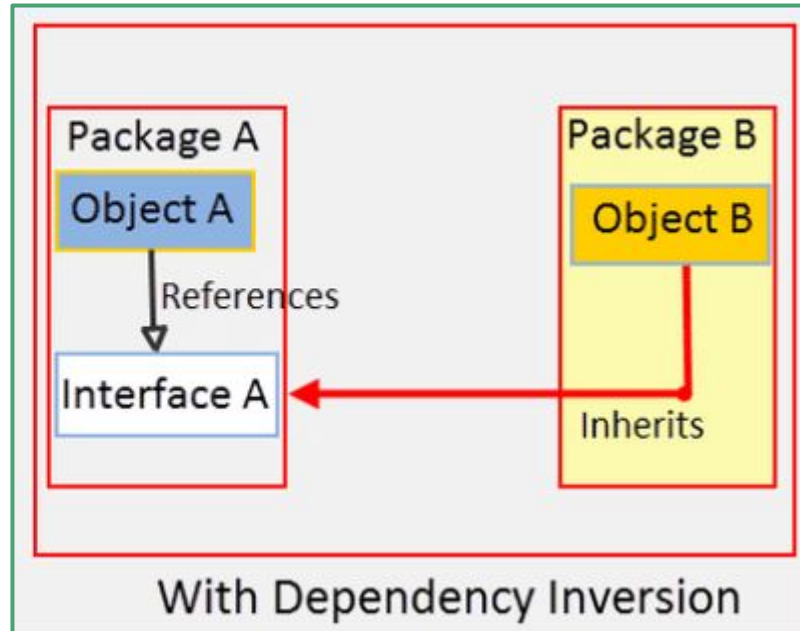


The Principles of SOLID - DIP



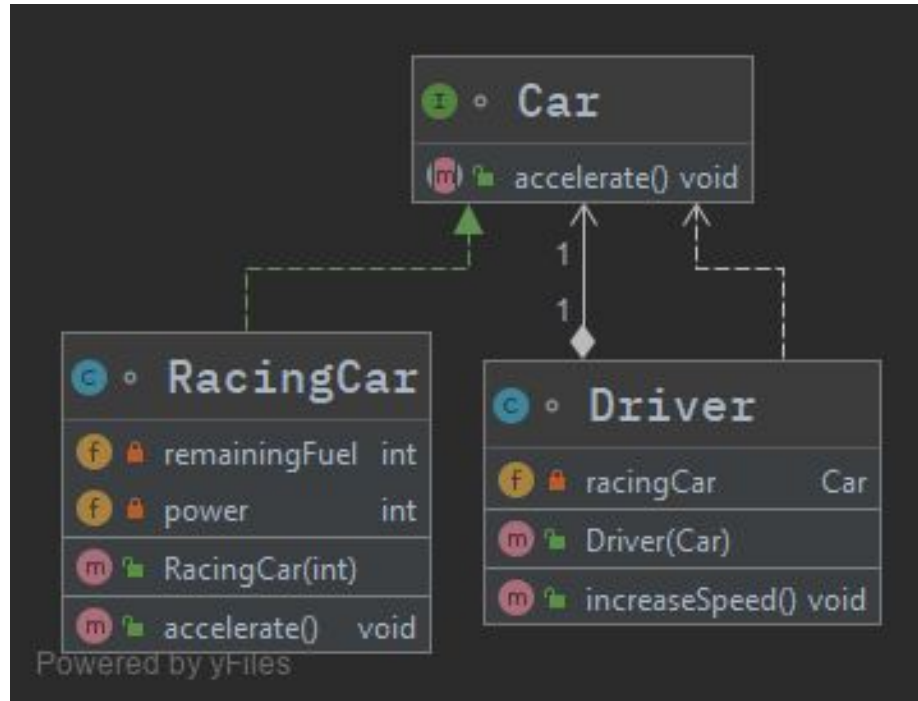


The Principles of SOLID - DIP





The Principles of SOLID - DIP





The Principles of SOLID - DIP

```
class Milk {  
    // implementation  
}  
  
class Coffee {  
    constructor(milk) {  
        this.milk = milk  
    }  
  
    cost() {  
        // calculate the cost  
    }  
}
```

```
class Ingredient {  
    // implementation  
}  
  
class Milk extends Ingredient {  
    // implementation  
}
```



The Principles of SOLID - DIP

```
class Beverage {  
    constructor(ingredient) {  
        this.ingredient = ingredient  
    }  
    cost() {  
        // calculate the cost  
    }  
}  
class Coffee extends Beverage {  
    cost() {  
        // calculate the cost  
    }  
}
```

¿Preguntas?



<https://github.com/ULL-ESIT-INF-PAI-2019-2020/2019-2020-pai-trabajo-oopp-solid-grupo-solid>

GRACIAS POR SU ATENCIÓN

References

- Eloquent Javascript - chapter 06 (https://eloquentjavascript.net/06_object.html)
- Javascript.info - (<https://javascript.info/object-basics>)
- Different websites:
 - <https://courses.cs.washington.edu/courses/cse341/99wi/java/tutorial/java/objects/object.html>
 - [https://en.wikipedia.org/wiki/Polymorphism_\(computer_science\)](https://en.wikipedia.org/wiki/Polymorphism_(computer_science))
 - <https://lostechies.com/chadmyers/2008/03/08/pablo-s-topic-of-the-month-march-solid-principles/>
 - <https://programmingwithmosh.com/javascript/solid-5-principles-of-object-oriented-design-every-developer-must-learn/>
 - <https://blog.raulmoya.es/principios-s-o-l-i-d-en-javascript-14041e6a3c43>