

Patrones de Diseño:

POO y SOLID

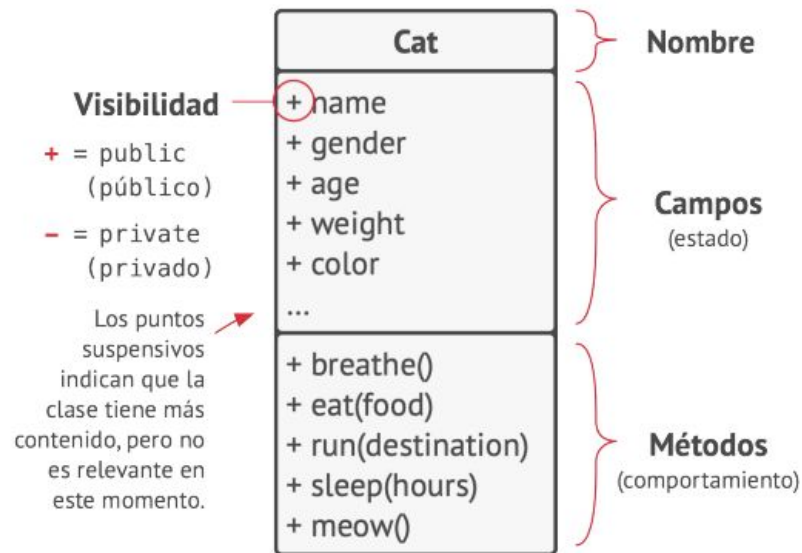
Programación Orientada a Objetos



Abstracción

Clases / Objetos

```
[8] 1 class Cat:
2     def __init__(self, name, gender, age, weight, color, texture):
3         self.name = name
4         self.gender = gender
5         self.age = age
6         self.weight = weight
7         self.color = color
8         self.texture = texture
9         # ...
10
11     def breathe(self):
12         pass
13
14     def eat(self, food):
15         pass
16
17     def run(self, destination):
18         pass
19
20     def sleep(self, hours):
21         pass
22
23     def meow():
24         pass
25
```



```
1 oscar = Cat(  
2     name="Óscar",  
3     gender="macho",  
4     age=3,  
5     weight=7,  
6     color="marrón",  
7     texture="rayada"  
8 )  
9  
10 luna = Cat(  
11     name="Luna",  
12     gender="hembra",  
13     age=2,  
14     weight=5,  
15     color="gris",  
16     texture="lisa"  
17 )  
18 oscar, luna
```

(<__main__.Cat at 0x7f905282bc10>, <__main__.Cat at 0x7f905282bc50>)



Óscar: Cat

name = "Óscar"
sex = "macho"
age = 3
weight = 7
color = marrón
texture = rayada



Luna: Cat

name = "Luna"
sex = "hembra"
age = 2
weight = 5
color = gris
texture = lisa

Encapsulación

```
1 class EjemploEncapsulacion(object):
2     atributo_publico = "Soy un atributo público"
3     _atributo_protegido = "Soy un atributo alcanzable desde una subclase."
4     __atributo_privado = "Soy un atributo inalcanzable desde fuera."
5
6     def metodo_publico(self):
7         print(self.__atributo_privado)
8
9     def _metodo_protegido(self):
10        print(self._atributo_protegido)
11
12    def __metodo_privado(self):
13        print(self.atributo_publico)
14
15 ee = EjemploEncapsulacion()
```

```
[30] 1 print(ee.atributo_publico)
      2 print(ee.metodo_publico())
```

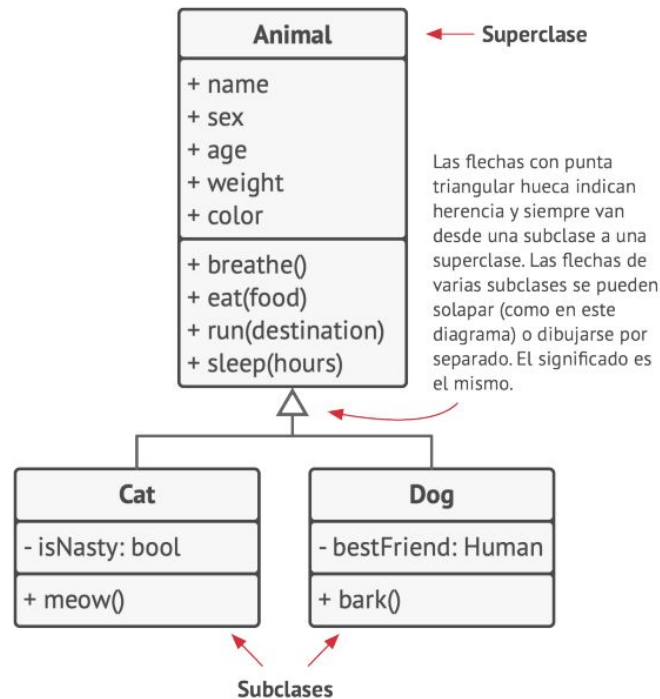
```
Soy un atributo público
Soy un atributo inalcanzable desde fuera.
None
```

```
[32] 1 # Nos levantara una excepción al intentar
      2 # llamar un atributo o método privado
      3 print(ee.__atributo_privado)
      4 print(ee.__metodo_privado())
      5
      6 # Nos levantara una excepción al intentar
      7 # llamar un atributo o método protegido
      8 print(ee._atributo_protegido)
      9 print(ee._metodo_protegido())
     10
```

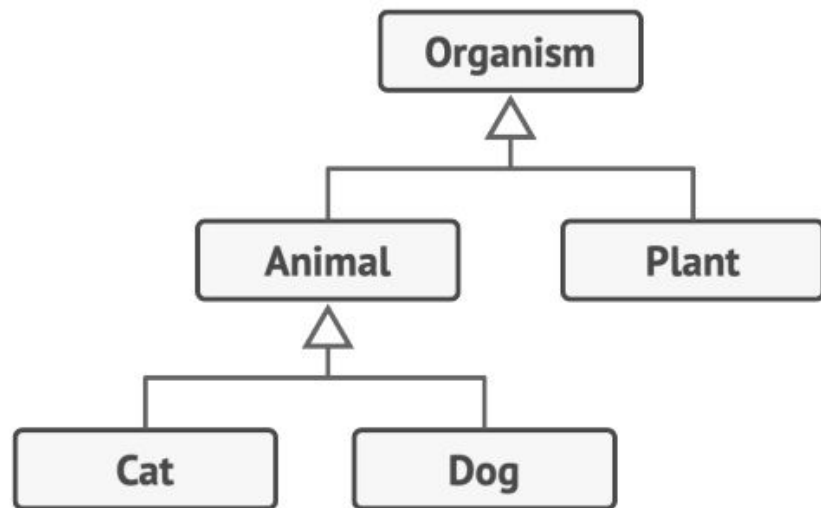


Herencia

```
1 class Animal(object):
2     def __init__(self, name, sex, age, weight, color):
3         self.name = name
4         self.sex = sex
5         self.age = age
6         self.weight = weight
7         self.color = color
8     def breathe(self):
9         pass
10    def eat(self, food):
11        pass
12    def run(self, destination):
13        pass
14    def sleep(self, hours):
15        pass
16
17
18 class Cat(Animal):
19     def __init__(self, name, sex, age, weight, color, is_nasty):
20         super().__init__(name, sex, age, weight, color)
21         self.is_nasty = is_nasty
22     def meow(self):
23         pass
24
25
26 class Dog(Animal):
27     def __init__(self, name, sex, age, weight, color, best_friend):
28         super().__init__(name, sex, age, weight, color)
29         self.best_friend = best_friend
30     def bark(self):
31         pass
```



```
1 class Organism(object):  
2     pass  
3  
4  
5 class Plant(Organism):  
6     pass  
7  
8  
9 class Animal(Organism):  
10     pass  
11  
12  
13 class Cat(Animal):  
14     pass  
15  
16  
17 class Dog(Animal):  
18     pass
```



Polimorfismo

Es la capacidad de detectar la verdadera clase de un objeto e invocar su implementación.

Incluso aunque su tipo real sea desconocido en el contexto actual.

```
// Código Java
class Animal{
    public Animal() {}
}

class Perro extends Animal {
    public Perro() {}
}

class Gato extends Animal {
    public Gato() {}
}
```

```
// Código Java
Animal[] animales = new Animal[10];
animales[0] = new Perro();
animales[1] = new Gato();
```

```
// Código Java
class OtraClase {
    public OtraClase() {}
}

Animal a = new OtraClase();
animales[0] = new OtraClase();
```




```
1 from abc import abstractmethod
2 from abc import ABC
3
4 class Animal(ABC):
5     # ...
6     @abstractmethod
7     def make_sound(self):
8         pass
9
10
11 class Cat(Animal):
12     def make_sound(self):
13         print("Maullar!")
14
15
16 class Dog(Animal):
17     def make_sound(self):
18         print("Ladrar!")
19
20
21 class RubberDuck:
22     def make_sound(self):
23         print("Cuack!")
24
```

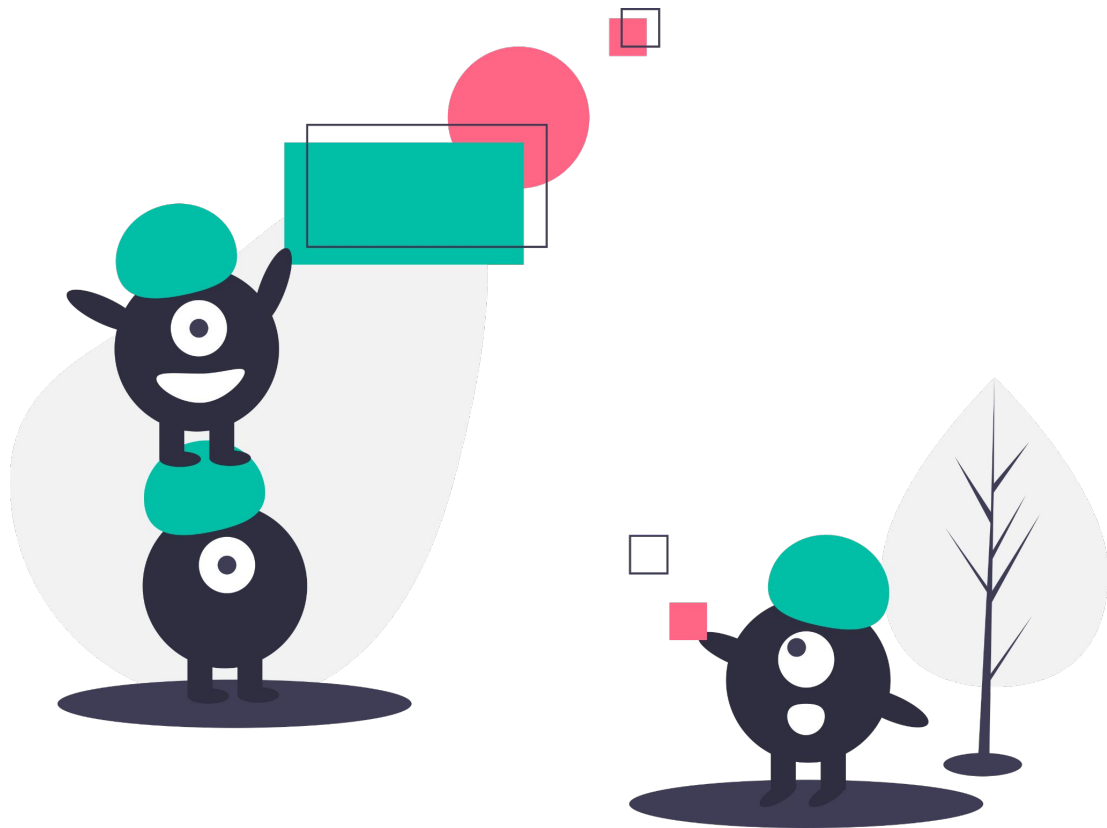
```
[52] 1 bag = [Cat(), Dog(), RubberDuck()]
      2 for element in bag:
      3     print(isinstance(element, Animal), )
      4     print(element.make_sound())
      5     print()
```

True
Maullar!
None

True
Ladrar!
None

False
Cuack!
None

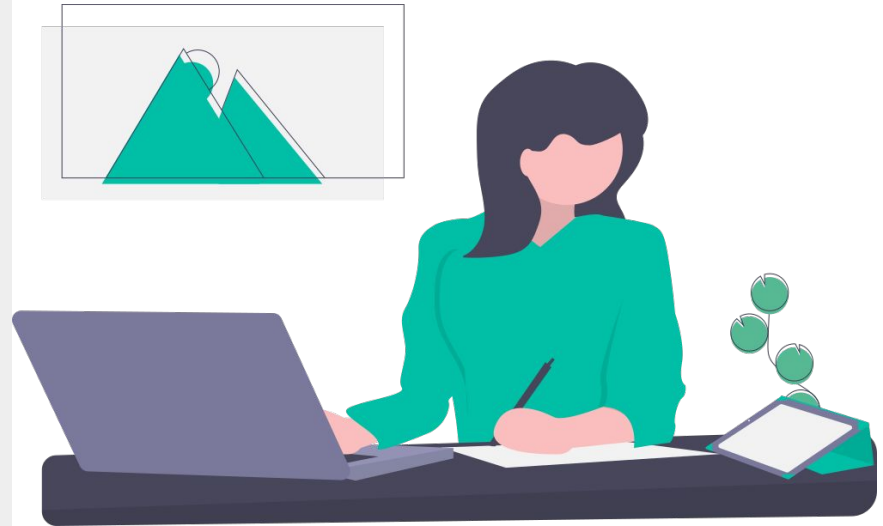
Principios SOLID

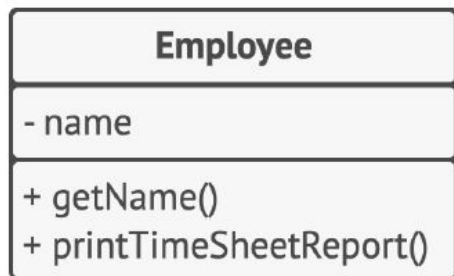


Single Responsibility Principle

Principio de responsabilidad única

- ****Reducir la complejidad.**
- Cada clase es responsable de una única parte de la funcionalidad esa responsabilidad
- Las clases crecen y requieren cambios constantes. Ya no recuerdas los detalles.
- Una clase con varias funcionalidades, es más susceptible a cambios continuos. Te arriesgas a descomponer otras partes de la clase que no pretendías cambiar

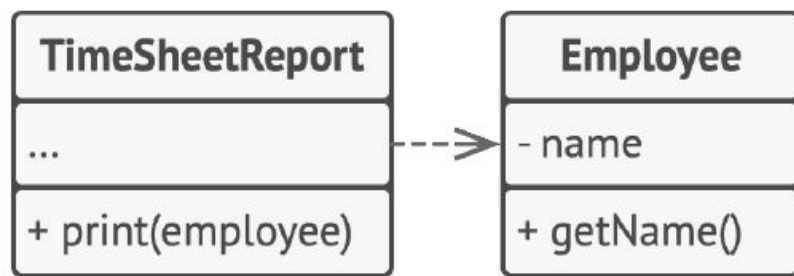
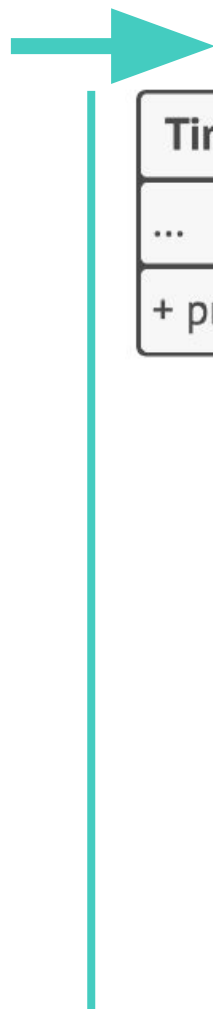




```

1 class Employee:
2     def __init__(self, name):
3         self.__name = name
4
5     def get_name(self):
6         return self.__name
7
8     def print_time_sheet_report(self):
9         # Lógica compleja
10        # ...
11        result = self.__name + " - 0001"
12        print(result)
13
14 e = Employee("Fernando Pérez")
15 e.print_time_sheet_report()
  
```

Fernando Pérez- 0001



```

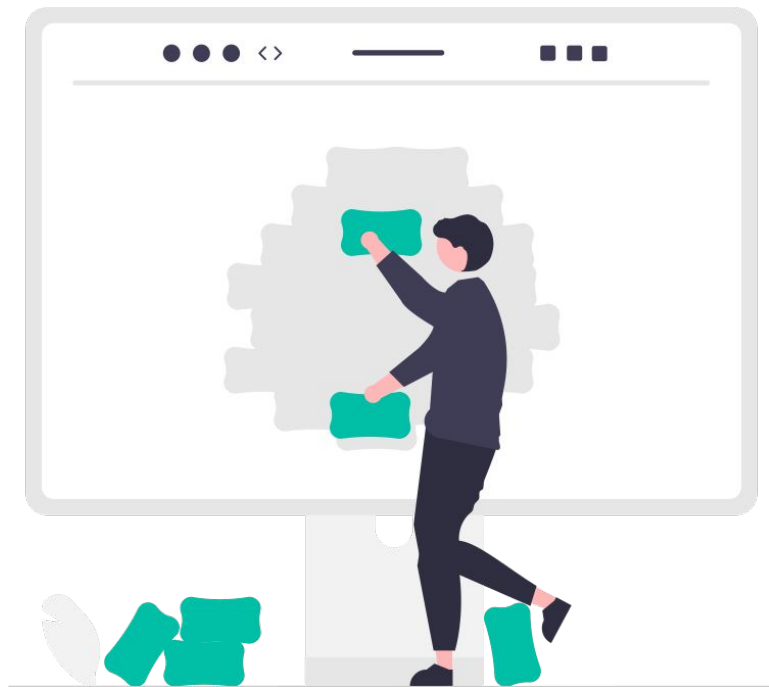
1 class Employee:
2     def __init__(self, name):
3         self.__name = name
4
5     def get_name(self):
6         return self.__name
7
8
9 class TimeSheetReport:
10    def print(self, employee: Employee):
11        # Lógica compleja
12        # ...
13        result = employee.get_name() + " - 0001"
14        print(result)
15
16 e = Employee("Fernando Pérez")
17
18 tsr = TimeSheetReport()
19 tsr.print(e)
  
```

Fernando Pérez - 0001

Open/Closed Principle

Principio de abierto/cerrado

- **** Evitar descomponer funcionalidades existentes en nuevas implementaciones.**
- **Abiertas**, para extenderlas en subclase (añadir atributos y métodos o sobrescribir).
- **Cerradas**, la clase está completa para que otras clases la utilicen; interfaz claramente definida y no se cambiará en el futuro.
- Las subclases permite sobrescribir partes de la clase original para que se comporten de otra manera.
- *Los hijos no deben cargar con problemas de los padres.*



| Order |
|---|
| - lineItems - shipping |
| + getTotal() + getTotalWeight() + setShippingType(st) + getShippingCost() - + getShippingDate() |

```

if (shipping == "ground") {
    // Envío por tierra gratuito en
    // grandes pedidos.
    if (getTotal() > 100) {
        return 0
    }
    // $1.5 por kilo, pero $10 mínimo.
    return max(10, getTotalWeight() * 1.5)
}

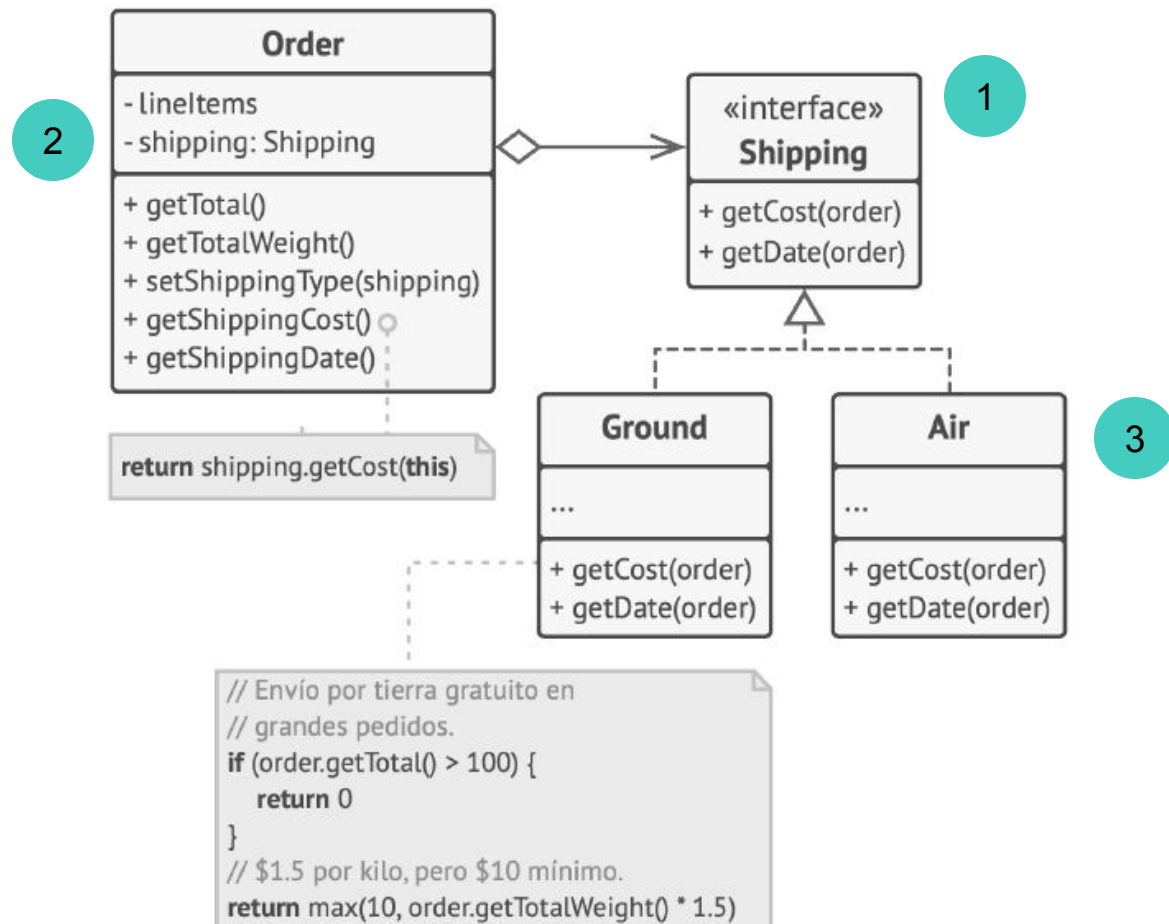
if (shipping == "air") {
    // $3 por kilo, pero $20 mínimo.
    return max(20, getTotalWeight() * 3)
}

```

```

1 class Order:
2     def __init__(self, line_items, shipping):
3         self.__line_items = line_items
4         self.__shipping = shipping
5
6     def get_total(self):
7         pass
8
9     def get_total_weight(self):
10        pass
11
12    def set_shipping_type(self, shipping:str):
13        self.__shipping = shipping
14
15    def get_shipping_cost(self):
16        # Envío por tierra
17        if self.shipping == "ground":
18            # Envío por tierra gratuito en grandes pedidos
19            if self.get_total > 100:
20                return 0
21            # $1.5 por kilo, pero $10 minimo
22            t = self.get_total_weight() * 1.5
23            return t if t > 10 else 10
24
25        # Envío por aire
26        if self.shipping == "air":
27            # $3 por kilo, pero $20 minimo
28            t = self.get_total_weight() * 3
29            return t if t > 20 else 20
30
31        # XXX: Agregar aqui mas envios ...
32
33    def get_shipping_date(self):
34        pass

```



```

1 class Order:
2     def __init__(self, line_items, shipping):
3         self.__line_items = line_items
4         self.__shipping = shipping
5
6     def get_total(self):
7         pass
8
9     def get_total_weight(self):
10        pass
11
12    def set_shipping_type(self, shipping:Shipping):
13        self.__shipping = shipping
14
15    def get_shipping_cost(self):
16        # Aplicando polimorfismo y con la interface podemos
17        # obtener el costo dependiente de la instancia en shipping
18        self.__shipping.get_cost()
19
20    def get_shipping_date(self):
21        pass

```

2

```

1 from abc import abstractmethod
2 from abc import ABC
3
4 class Shipping(ABC):
5     @abstractmethod
6     def get_cost(self, order):
7         pass
8
9     @abstractmethod
10    def get_date(self, order):
11        pass

```

1

```

1 class Ground(Shipping):
2     def get_date(self):
3         pass
4
5     def get_cost(self, order: Order):
6         # Envio gratuito en grandes pedidos
7         if self.get_total > 100:
8             return 0
9         # $1.5 por kilo, pero $10 minimo
10        t = self.get_total_weight() * 1.5
11        return t if t > 10 else 10

```

3

```

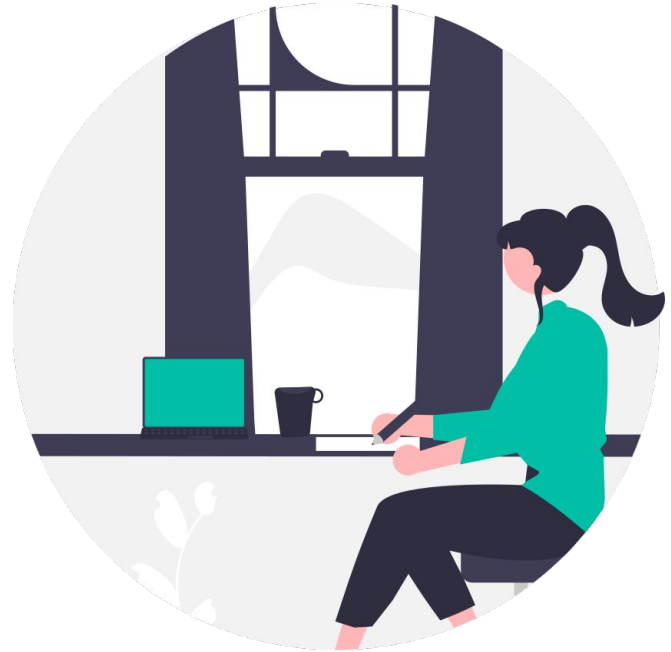
14 class Air(Shipping):
15     def get_date(self):
16         pass
17
18     def get_cost(self, order: Order):
19         # $3 por kilo, pero $20 minimo
20         t = self.get_total_weight() * 3
21         return t if t > 20 else 20
22

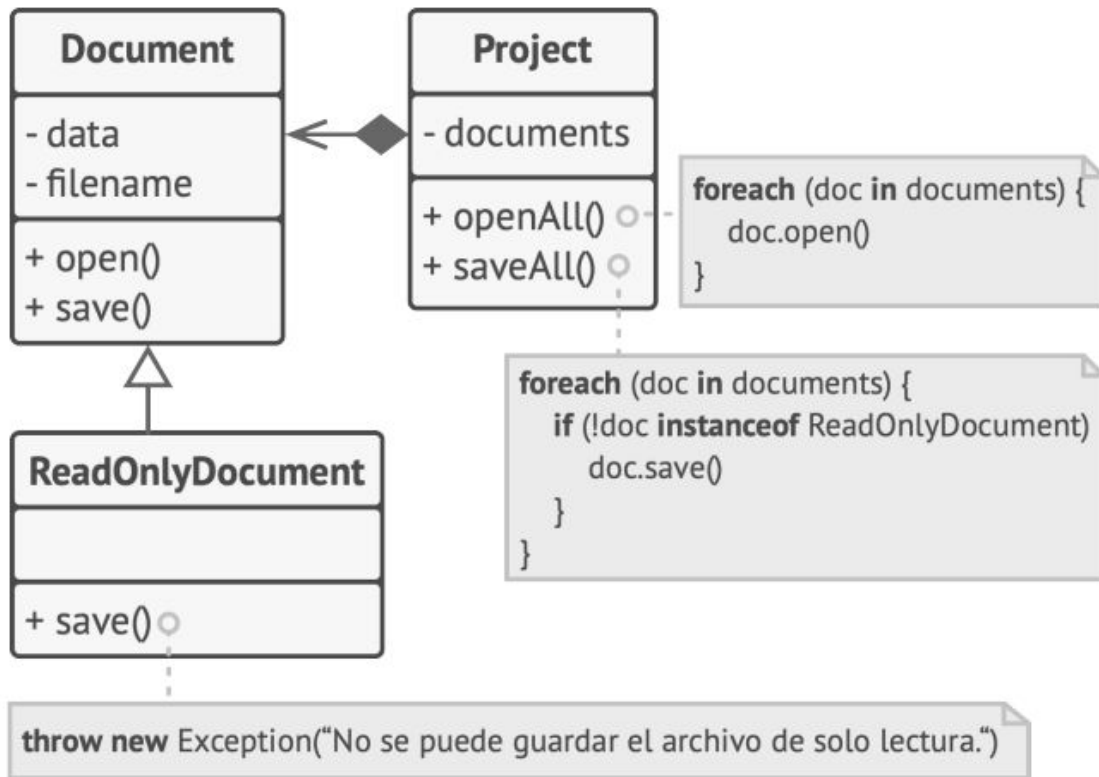
```


Liskov Substitution Principle

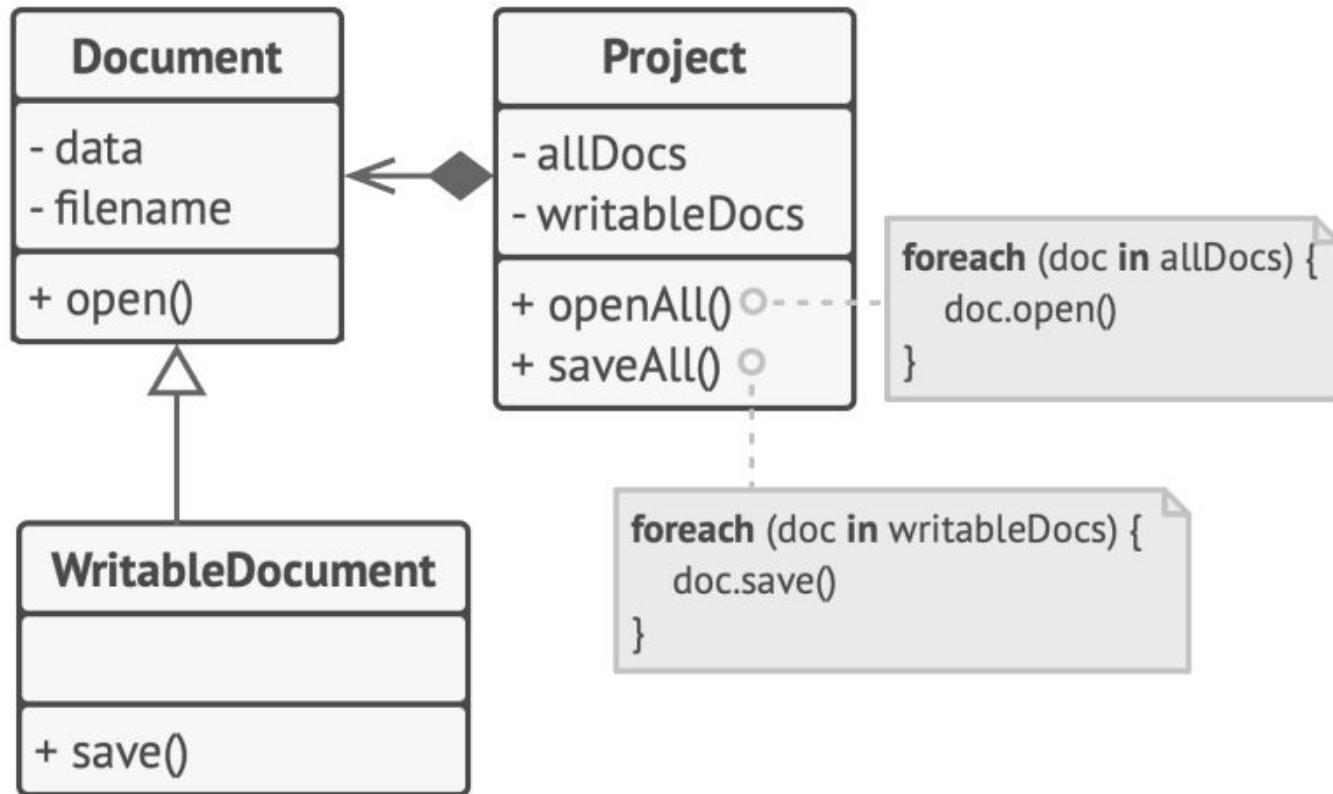
Principio de sustitución de Liskov

- Una subclase debe permanecer compatible con el comportamiento de la superclase.
- Debemos extender el comportamiento base, no cambiarlo por algo muy distinto.





Project se vuelve dependiente



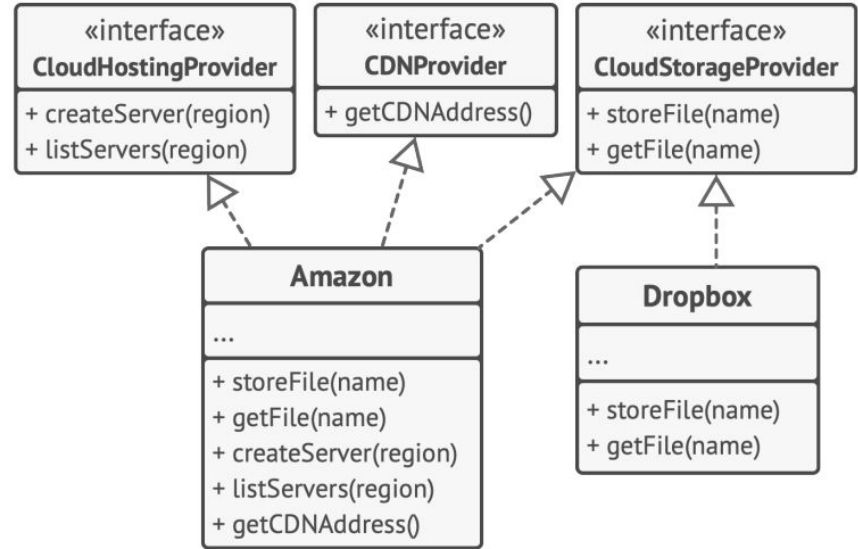
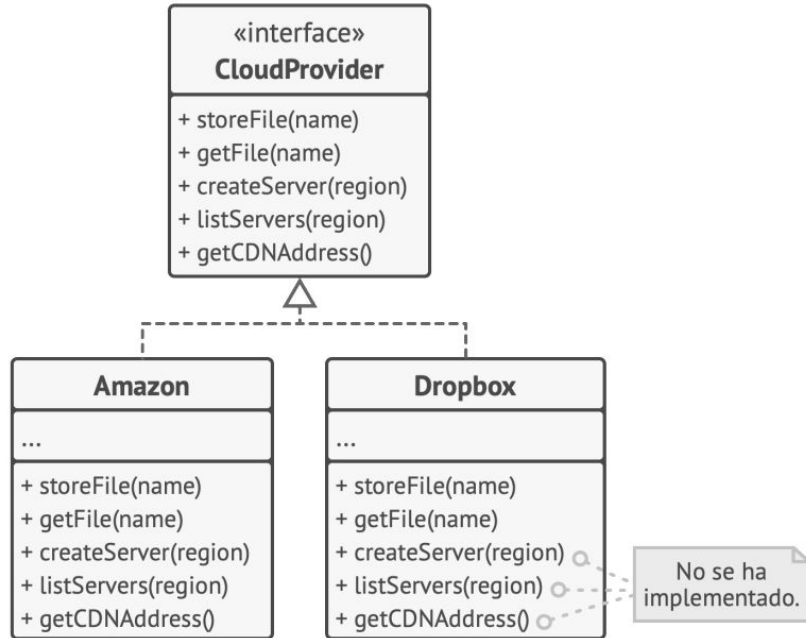
Interface Segregation Principle

Principio de segregación de la interfaz

- Desintegrar interfaces “grandes” hasta crear otras más detalladas y específicas
- No se debe forzar a depender de métodos innecesarios.
- 1 superclase, n interfaces.
- Bien, pero NO te conviertas en adicto a las interfaces. Equilibrio ante todo.

after-content.component.ts X

```
19 export class AfterContentComponent implements AfterContentChecked, AfterContentInit {  
20   private prevHero = '';  
21   comment = '';  
22  
23   // Query for a CONTENT child of type `ChildComponent`  
24   @ContentChild(ChildComponent) contentChild!: ChildComponent;  
25  
26   constructor(private logger: LoggerService) {  
27     this.logIt('AfterContent constructor');  
28   }  
29  
30   ngAfterContentInit() {  
31     // contentChild is set after the content has been initialized  
32     this.logIt('AfterContentInit');  
33     this.doSomething();  
34   }  
35  
36   ngAfterContentChecked() {  
37     // contentChild is updated after the content has been checked  
38     if (this.prevHero === this.contentChild.hero) {  
39       this.logIt('AfterContentChecked (no change)');  
40     } else {  
41       this.prevHero = this.contentChild.hero;  
42       this.logIt('AfterContentChecked');  
43       this.doSomething();  
44     }  
45   }  
46 }
```



Dependency Inversion Principle

Principio de inversión de la dependencia

- Clases de bajo nivel. Operaciones básicas (trabajar con disco, transferir datos por red, conexiones a base de datos)
- Clases de alto nivel. Lógica de negocio compleja y usan las de bajo nivel.
- Comúnmente diseñan primero las clases de bajo nivel y luego comienzan con las de alto nivel.

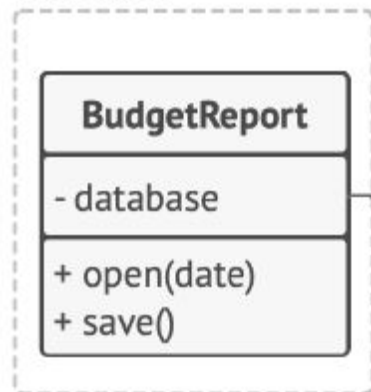
No estás seguro de lo que es posible a alto nivel, porque el contenido de bajo nivel aún no está implementado o claro.

Con este sistema, las clases de la lógica de negocio tienden a hacerse dependientes de clases primarias de bajo nivel.

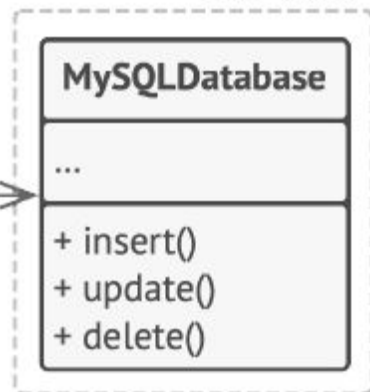
-



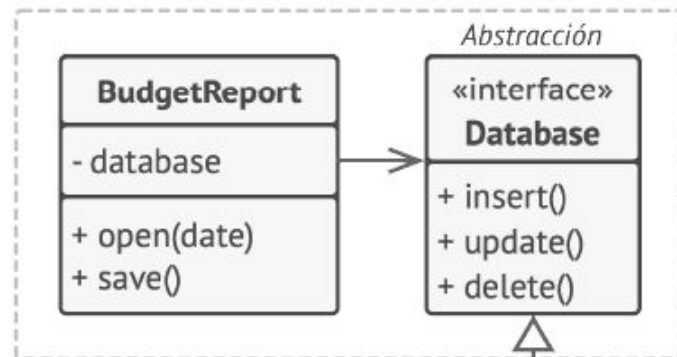
Alto nivel



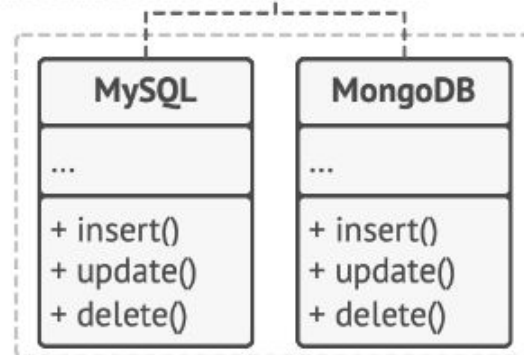
Bajo nivel



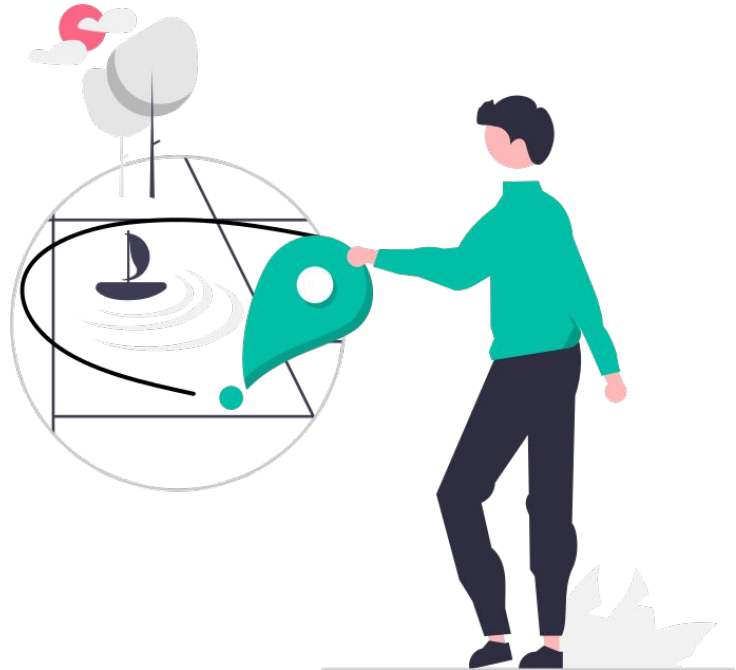
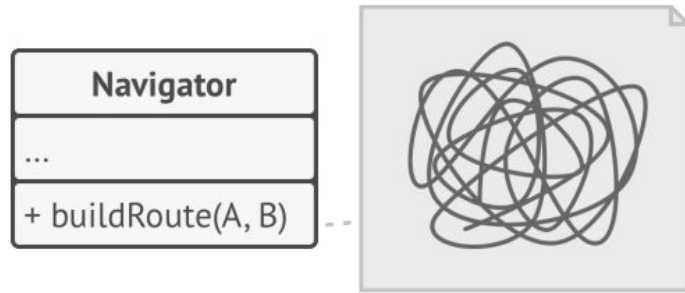
Alto nivel

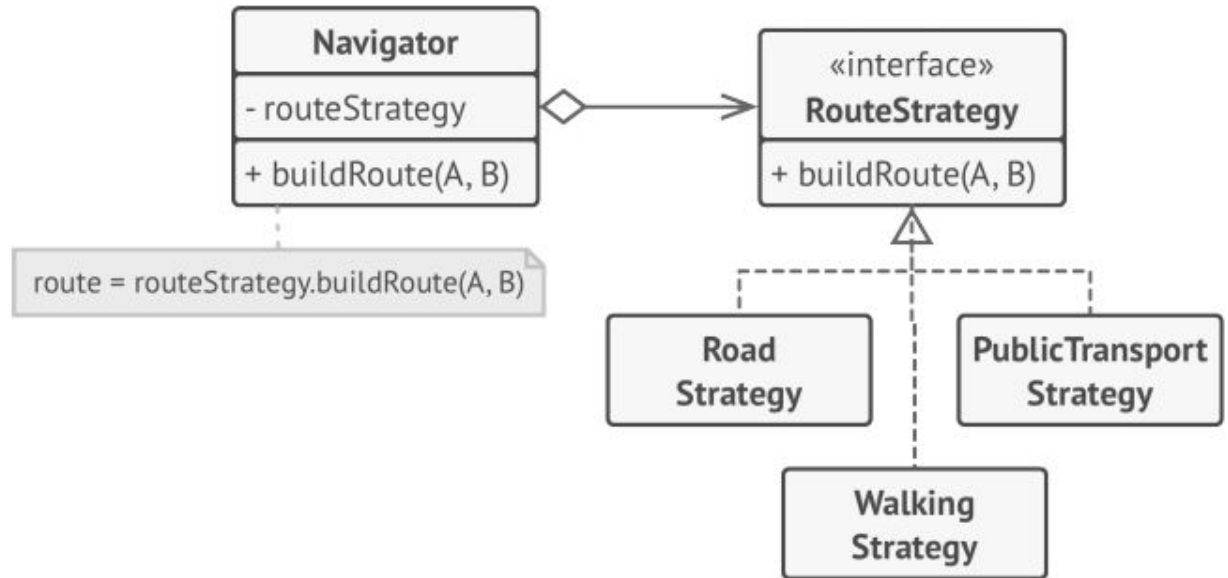


Bajo nivel

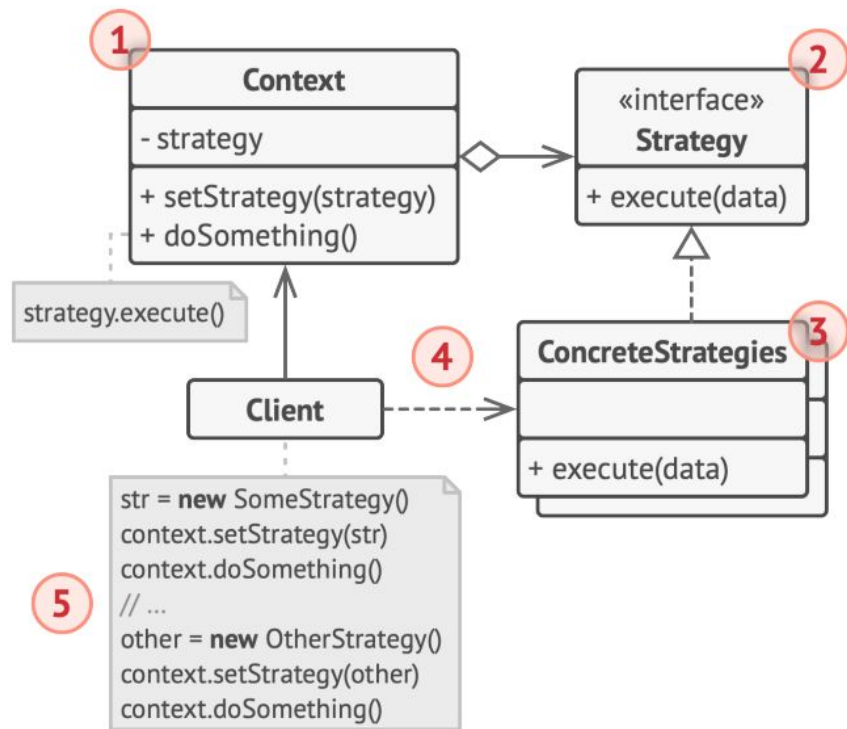
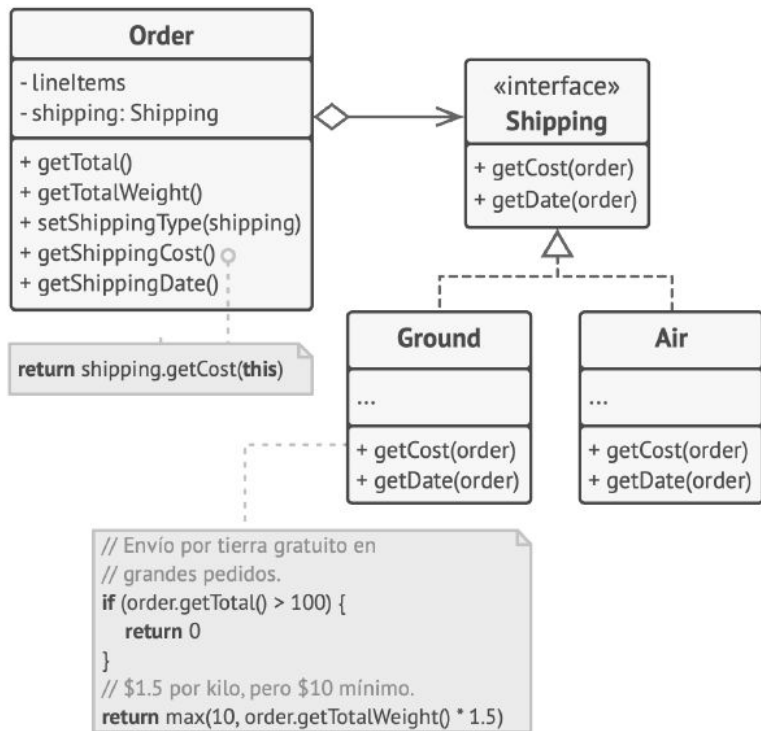


Estrategia



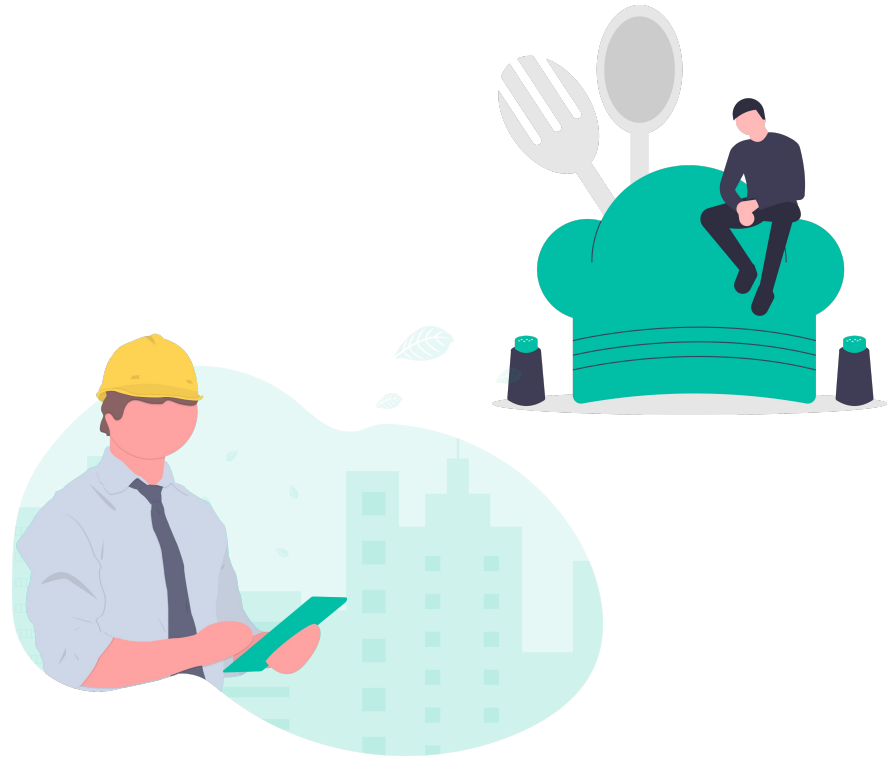


Patrón Estrategia permite definir una familia de algoritmos, extrae cada uno de ellos en una clase separada “Strategia” y hacer sus objetos intercambiables.



¿Qué son los patrones de diseño?

- Soluciones habituales a problemas que ocurren con frecuencia en el diseño de software.
- Son un concepto general para resolver un problema particular.
No es una porción específica de código (función o bibliotecas),
- Es muy diferente a algoritmos aunque ambos describen soluciones a problemas conocidos.
Son como planos prefabricados que se pueden personalizar para resolver un problema de diseño recurrente en tu código.



Gracias

