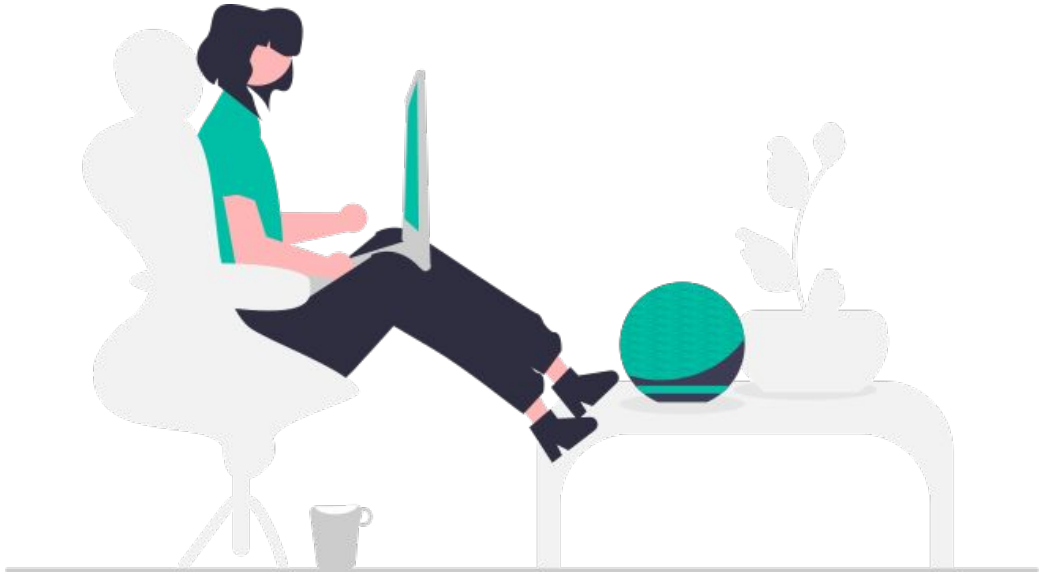


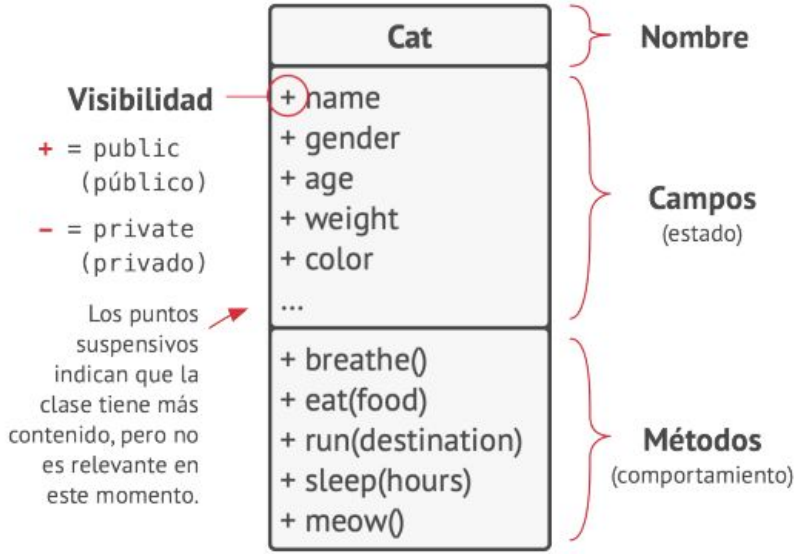
Patrones de Diseño:

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
```



```python
oscar = Cat(
    name="Óscar",
    gender="macho",
    age=3,
    weight=7,
    color="marrón",
    texture="rayada"
)

luna = Cat(
    name="Luna",
    gender="hembra",
    age=2,
    weight=5,
    color="gris",
    texture="lisa"
)
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

```python
class EjemploEncapsulacion(object):
    atributo_publico = "Soy un atributo público"
    _atributo_protegido = "Soy un atributo alcanzable desde una subclase."
    __atributo_privado = "Soy un atributo inalcanzable desde fuera."

    def metodo_publico(self):
        print(self.__atributo_privado)

    def _metodo_protegido(self):
        print(self._atributo_protegido)

    def __metodo_privado(self):
        print(self.atributo_publico)

ee = EjemploEncapsulacion()
```

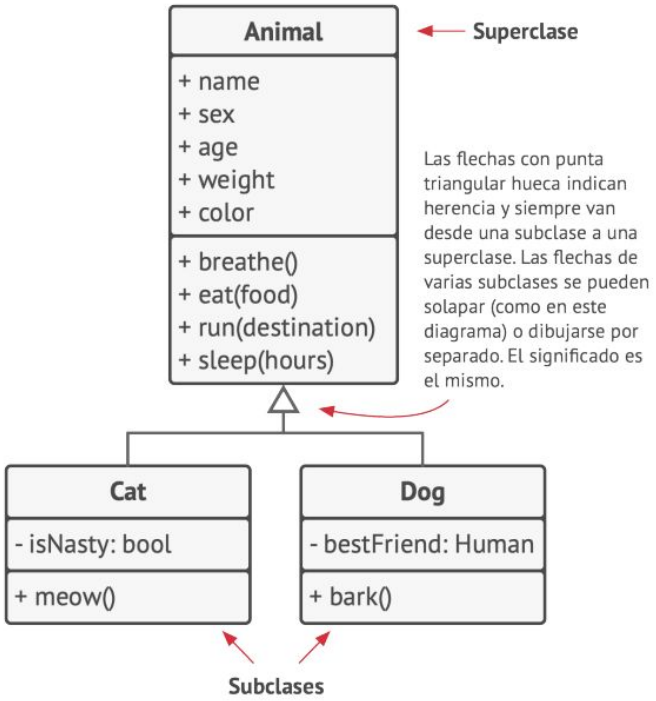
[30]
```python
print(ee.atributo_publico)
print(ee.metodo_publico())
```

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

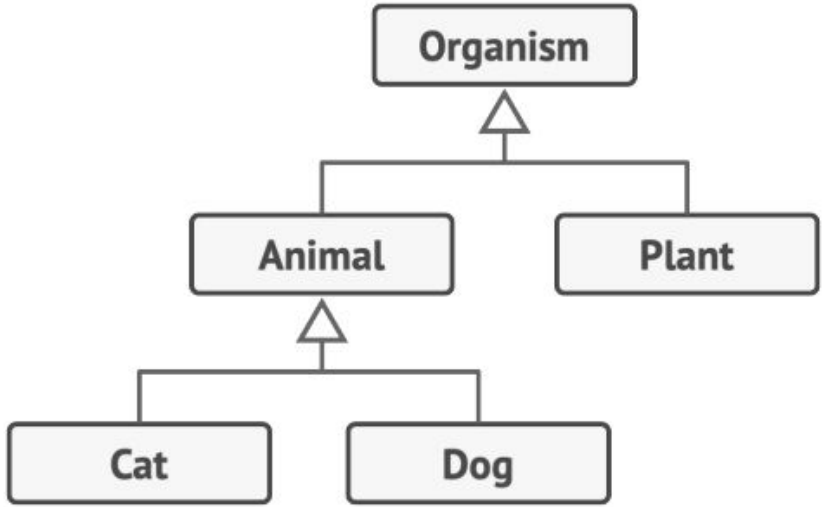
[32]
```python
# Nos levantara una excepción al intentar
#  llamar un atributo o método privado
print(ee.__atributo_privado)
print(ee.__metodo_privado())

# Nos levantara una excepción al intentar
#  llamar un atributo o método protegido
print(ee._atributo_protegido)
print(ee._metodo_protegido())

```

Herencia

```python
class Animal(object):
    def __init__(self, name, sex, age, weight, color):
        self.name = name
        self.sex = sex
        self.age = age
        self.weight = weight
        self.color = color
    def breathe(self):
        pass
    def eat(self, foot):
        pass
    def run(self, destination):
        pass
    def sleep(self, hours):
        pass


class Cat(Animal):
    def __init__(self, name, sex, age, weight, color, is_nasty):
        super().__init__(name, sex, age, weight, color)
        self.is_nasty = is_nasty
    def meow(self):
        pass


class Dog(Animal):
    def __init__(self, name, sex, age, weight, color, best_friend):
        super().__init__(name, sex, age, weight, color)
        self.best_friend = best_friend
    def bark(self):
        pass
```



```python
class Organism(object):
    pass


class Plant(Organism):
    pass


class Animal(Organism):
    pass


class Cat(Animal):
    pass

class Dog(Animal):
    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.

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

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

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

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

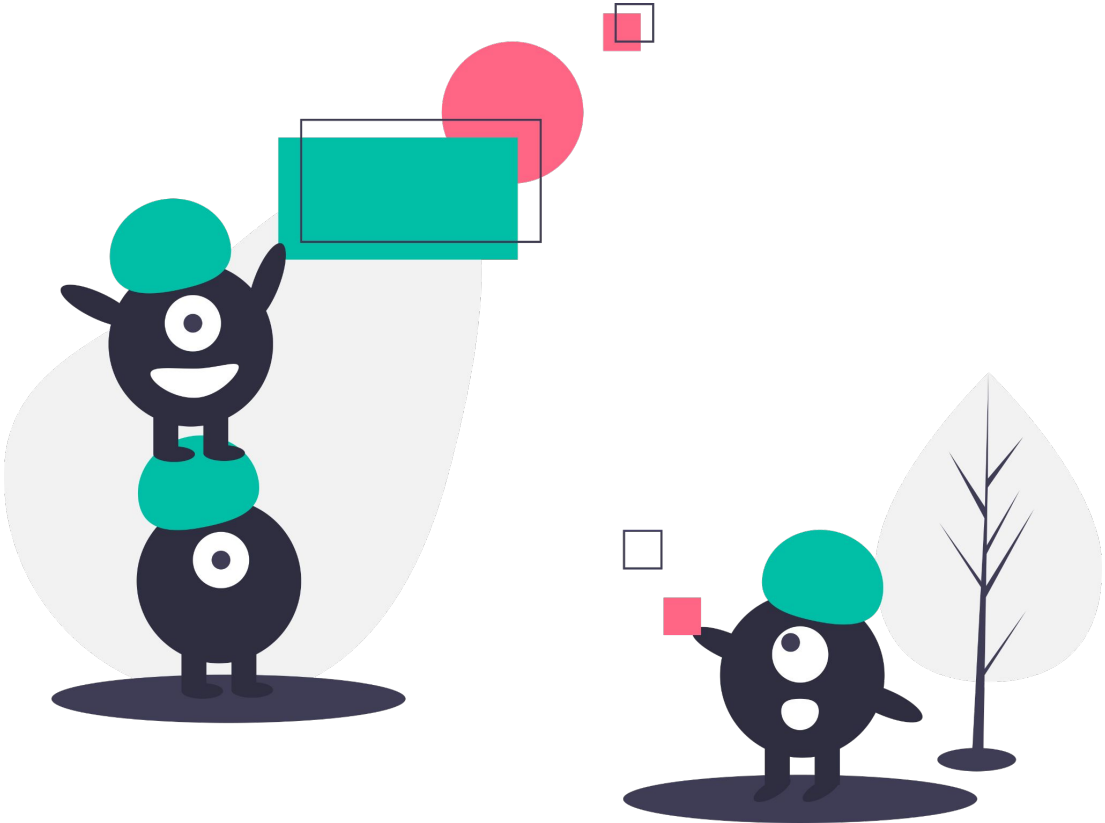
```java
// Código Java
class OtraClase {
    public OtraClase() {}
}
Animal a = new OtraClase();
animales[0] = new OtraClase();
```

```python
from abc import abstractmethod
from abc import ABC

class Animal(ABC):
    # ...
    @abstractmethod
    def make_sound(self):
        pass


class Cat(Animal):
    def make_sound(self):
        print("Maullar!")


class Dog(Animal):
    def make_sound(self):
        print("Ladrar!")


class RubberDuck:
    def make_sound(self):
        print("Cuack!")

```

```
[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



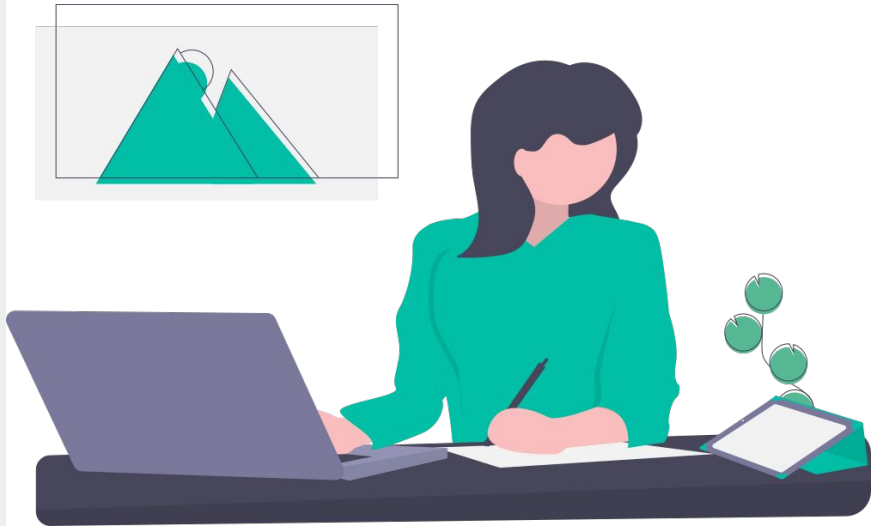
Single Responsibility Principle

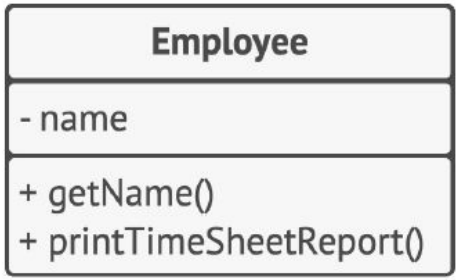
Principio de responsabilidad única

- ****Reducir la complejidad.**

- Cada clase es responsable de una única parte de la funcionalidad esa responsabilidad

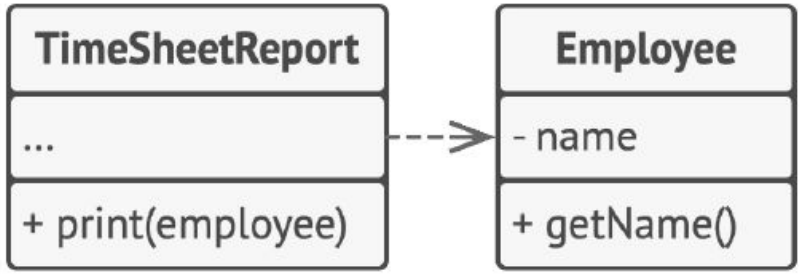
- Las clases crecen y requieren cambios constantes. Ya no recuerdas los detalles.
- Una clase con varias funcionalidades, es más susceptible a cambios continuos. Te arriesgas a descomponer otras partes de la clase que no pretendías cambiar






```python
class Employee:
    def __init__(self, name):
        self.__name = name

    def get_name(self):
        return self.__name

    def print_time_sheet_report(self):
        # Lógica compleja
        # ...
        result = self.__name + " - 0001"
        print(result)

e = Employee("Fernando Pérez")
e.print_time_sheet_report()
```

```
Fernando Pérez- 0001
```




```python
class Employee:
    def __init__(self, name):
        self.__name = name

    def get_name(self):
        return self.__name


class TimeSheetReport:
    def print(self, employee: Employee):
        # Lógica compleja
        # ...
        result = employee.get_name() + " - 0001"
        print(result)

e = Employee("Fernando Pérez")

tsr = TimeSheetReport()
tsr.print(e)
```

```
Fernando Pérez - 0001
```

Open/Closed Principle

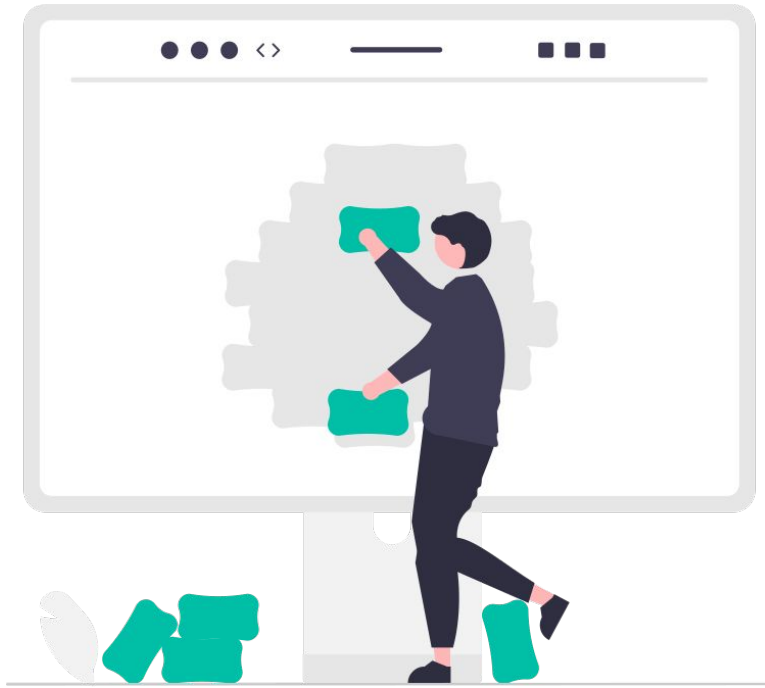
Principio de abierto/cerrado

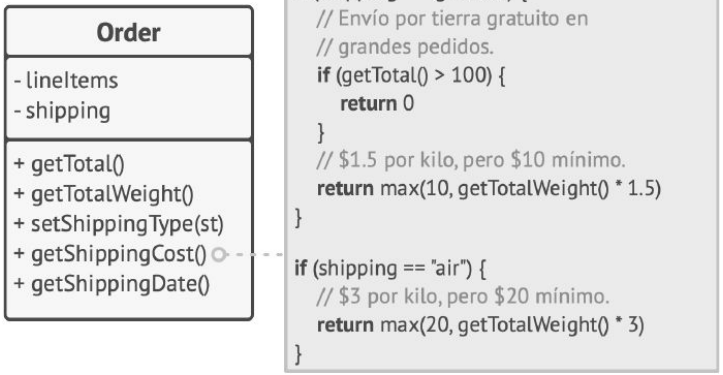
- ** **Evitar descomponer funcionalidades existentes en nuevas implementaciones.**

- **Abiertas**, para extenderlas en subclase (añadir atributos y métodos o sobreescribir).
- **Cerradas**, la clase está completa para que otras clases la utilicen; interfaz claramente definida y no se cambiará en el futuro.

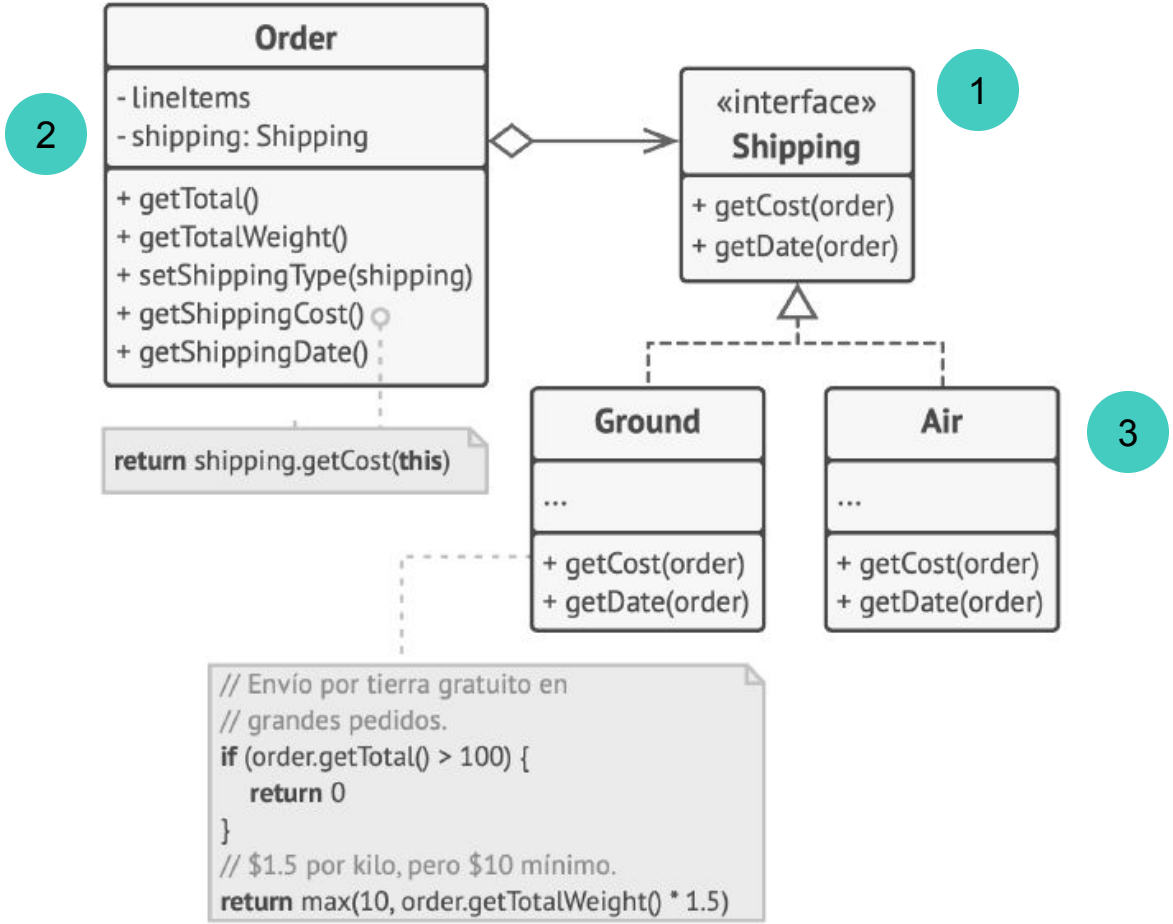
- Las subclases permite sobreescribir partes de la clase original para que se comporten de otra manera.

- *Los hijos no deben cargar con problemas de los padres.*






```python
class Order:
    def __init__(self, line_items, shipping):
        self.__line_items = line_items
        self.__shipping = shipping

    def get_total(self):
        pass

    def get_total_weight(self):
        pass

    def set_shipping_type(self, shipping:str):
        self.__shipping = shipping

    def get_shipping_cost(self):
        # Envio por tierra
        if self.shipping == "ground":
            # Envio por tierra gratuito en grandes pedidos
            if self.get_total > 100:
                return 0
            # $1.5 por kilo, pero $10 minimo
            t = self.get_total_weight() * 1.5
            return t if t > 10 else 10

        # Envio por aire
        if self.shipping == "air":
            # $3 por kilo, pero $20 minimo
            t = self.get_total_weight() * 3
            return t if t > 20 else 20

        # XXX: Agregar aqui mas envios ...

    def get_shipping_date(self):
        pass
```


Order
- lineItems
- shipping: Shipping
+ getTotal()
+ getTotalWeight()
+ setShippingType(shipping)
+ getShippingCost()
+ getShippingDate()
«interface»
Shipping
+ getCost(order)
+ getDate(order)
Ground
...
+ getCost(order)
+ getDate(order)
Air
...
+ getCost(order)
+ getDate(order)
return shipping.getCost(this)
// Envío por tierra gratuito en
// grandes pedidos.
if (order.getTotal() > 100) {
return 0
}
// $1.5 por kilo, pero $10 mínimo.
return max(10, order.getTotalWeight() * 1.5)
1
2
3

```python
from abc import abstractmethod
from abc import ABC

class Shipping(ABC):
    @abstractmethod
    def get_cost(self, order):
        pass

    @abstractmethod
    def get_date(self, order):
        pass
```

1

```python
class Order:
    def __init__(self, line_items, shipping):
        self.__line_items = line_items
        self.__shipping = shipping

    def get_total(self):
        pass

    def get_total_weight(self):
        pass

    def set_shipping_type(self, shipping:Shipping):
        self.__shipping = shipping

    def get_shipping_cost(self):
        # Aplicando polimorfismo y con la interface podemos
        #  obtener el costo dependiento de la instancia en shippping
        self.__shipping.get_cost()

    def get_shipping_date(self):
        pass
```

2

```python
class Ground(Shipping):
    def get_date(self):
        pass

    def get_cost(self, order: Order):
        # Envio gratuito en grandes pedidos
        if self.get_total > 100:
            return 0
        # $1.5 por kilo, pero $10 minimo
        t = self.get_total_weight() * 1.5
        return t if t > 10 else 10
```

3

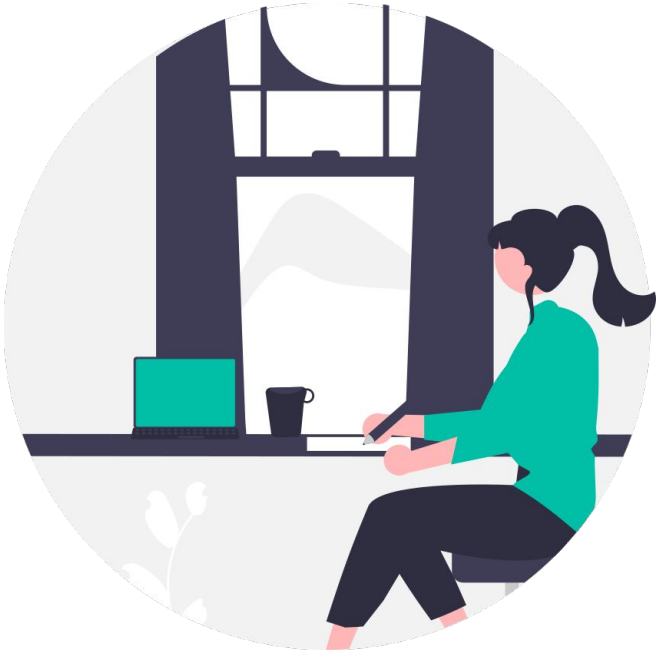
```python
class Air(Shipping):
    def get_date(self):
        pass

    def get_cost(self, order: Order):
        # $3 por kilo, pero $20 minimo
        t = self.get_total_weight() * 3
        return t if t > 20 else 20
```

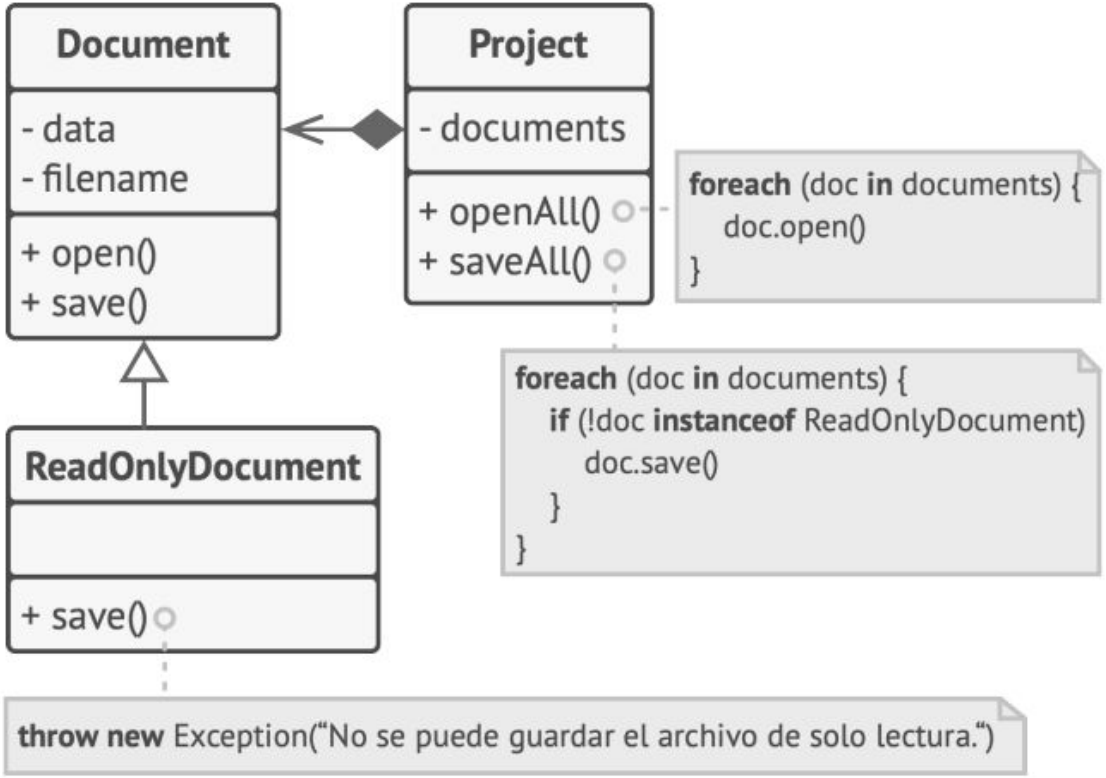
Liskov Substitution Principle

Principio de sustitución de Liskov

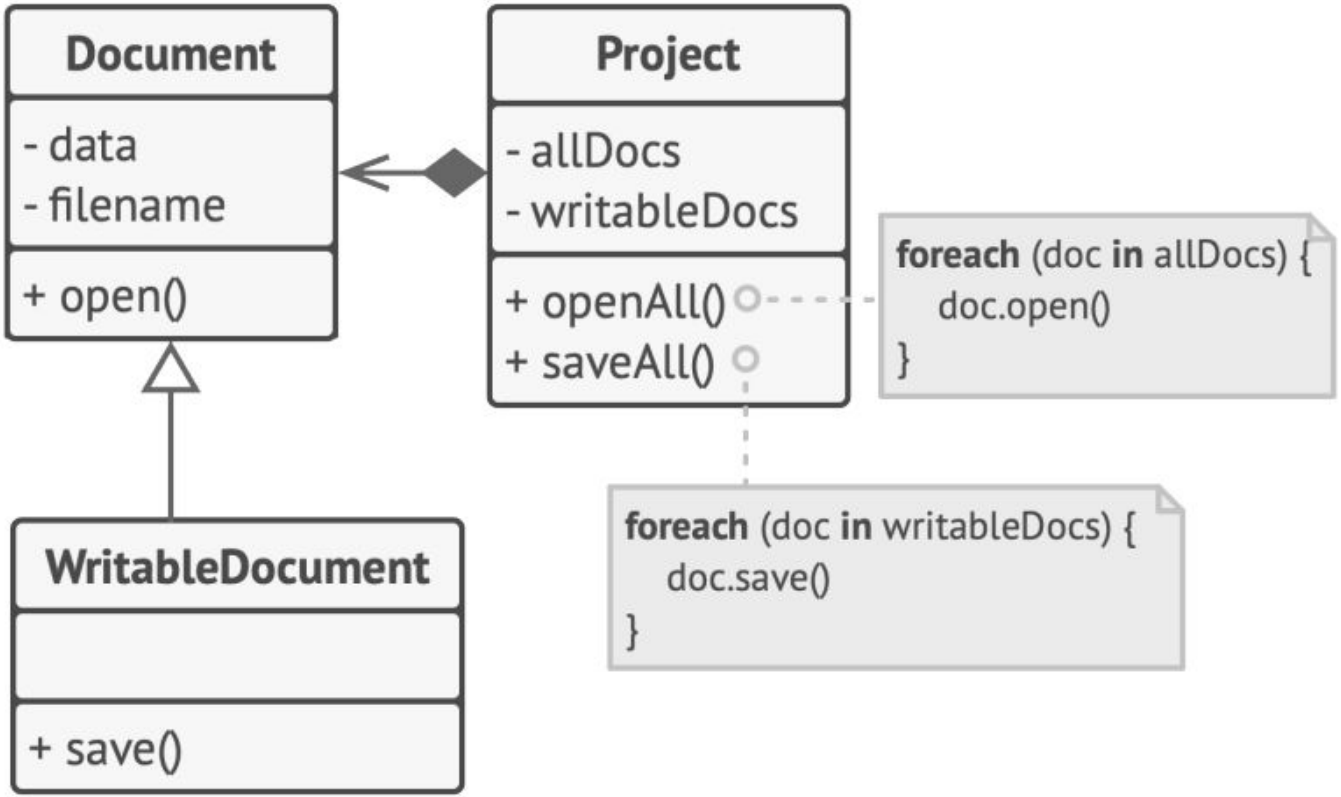
- Una subclase debe permanecer compatible con el comportamiento de la superclase.

- Debemos extender el comportamiento base, no cambiarlo por algo muy distinto.





Project se vuelve dependiente


Document
- data
- filename
+ open()
Project
- allDocs
- writableDocs
+ openAll()
+ saveAll()
foreach (doc in allDocs) {
doc.open()
}
foreach (doc in writableDocs) {
doc.save()
}
WritableDocument
+ save()

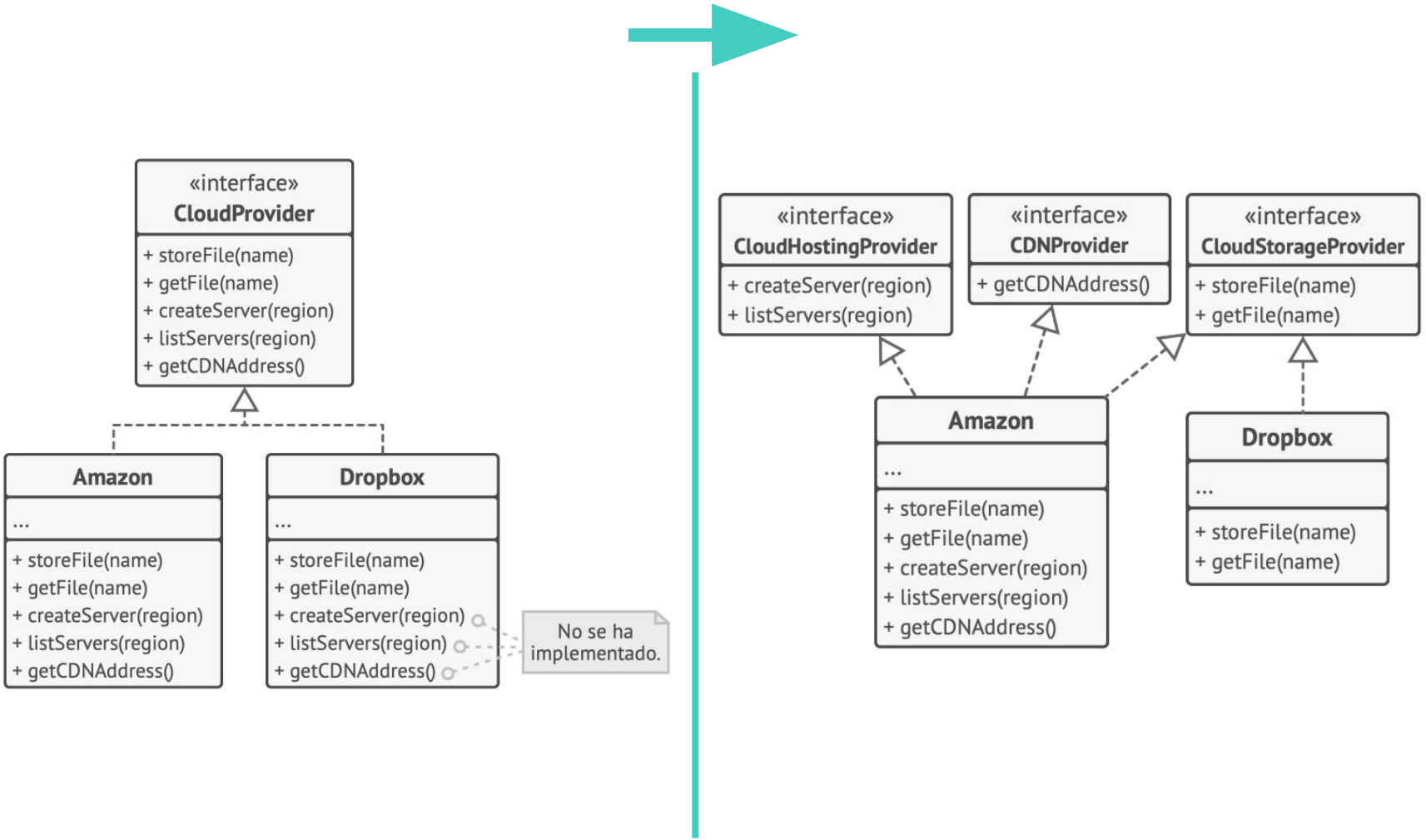
Interface Segregation Principle

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

```
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    }
```


«interface»
CloudProvider
+ storeFile(name)
+ getFile(name)
+ createServer(region)
+ listServers(region)
+ getCDNAddress()
Amazon
...
+ storeFile(name)
+ getFile(name)
+ createServer(region)
+ listServers(region)
+ getCDNAddress()
Dropbox
...
+ storeFile(name)
+ getFile(name)
+ createServer(region)
+ listServers(region)
+ getCDNAddress()
No se ha implementado.
«interface»
CloudHostingProvider
+ createServer(region)
+ listServers(region)
«interface»
CDNProvider
+ getCDNAddress()
«interface»
CloudStorageProvider
+ storeFile(name)
+ getFile(name)
Amazon
...
+ storeFile(name)
+ getFile(name)
+ createServer(region)
+ listServers(region)
+ getCDNAddress()
Dropbox
...
+ storeFile(name)
+ getFile(name)

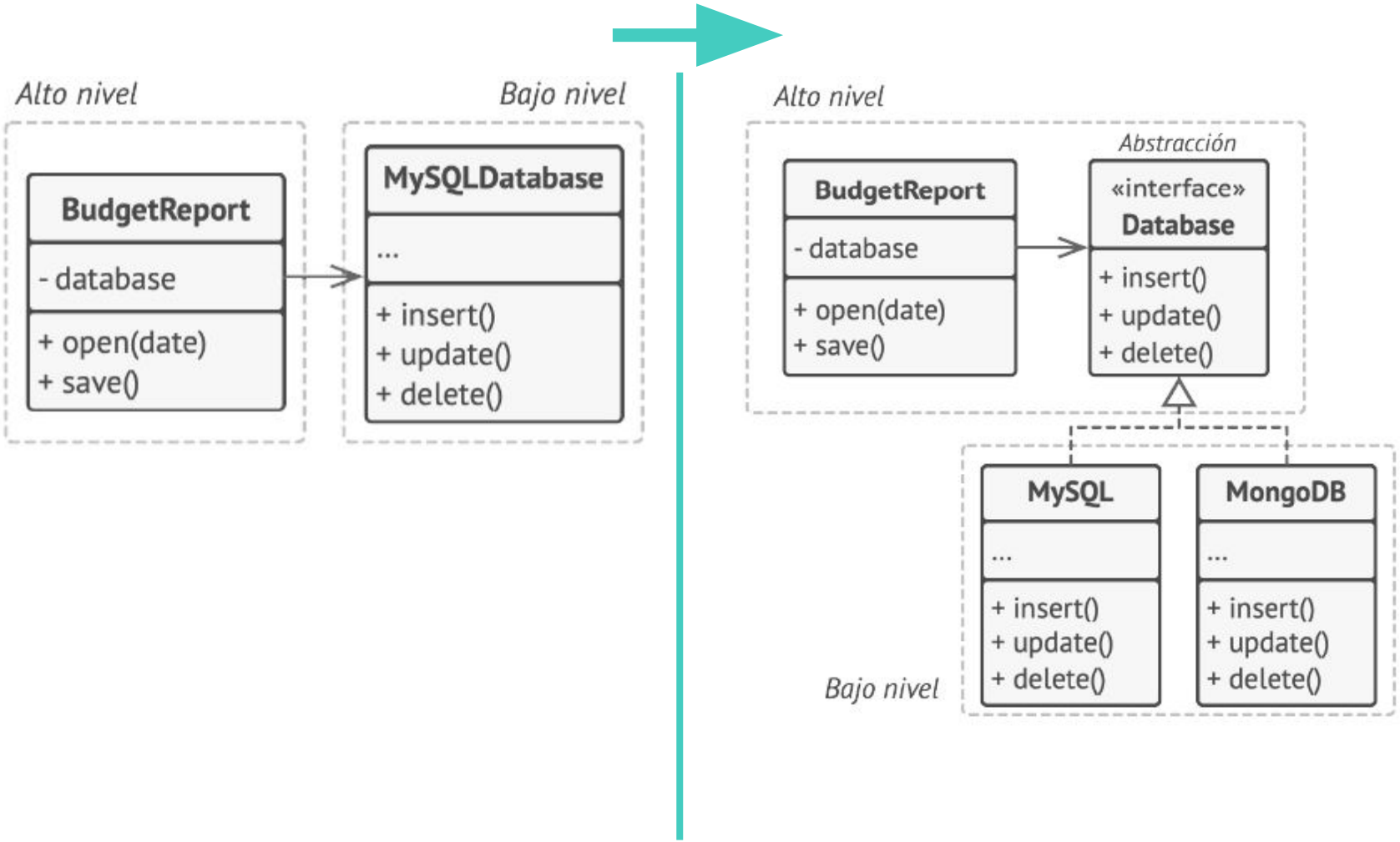
Dependency Inversion Principle

Principio de inversión de la dependencia

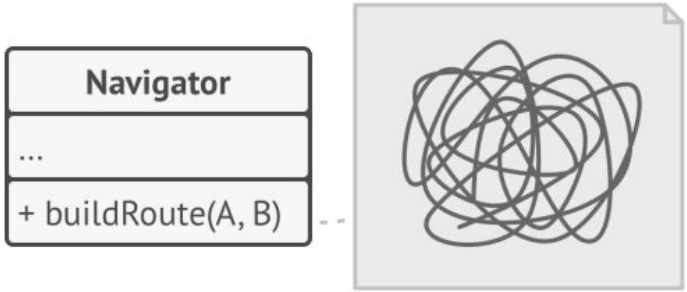
- Clases de bajo nivel. Operaciones básicas (trabajar con disco, transferir datos por red, conexiones a base de datos)
- Clases de alto nivel. Lógica de negocio compleja y usan las de bajo nivel.
- Comúnmente diseñan primero las clases de bajo nivel y luego comienzan con las de alto nivel.

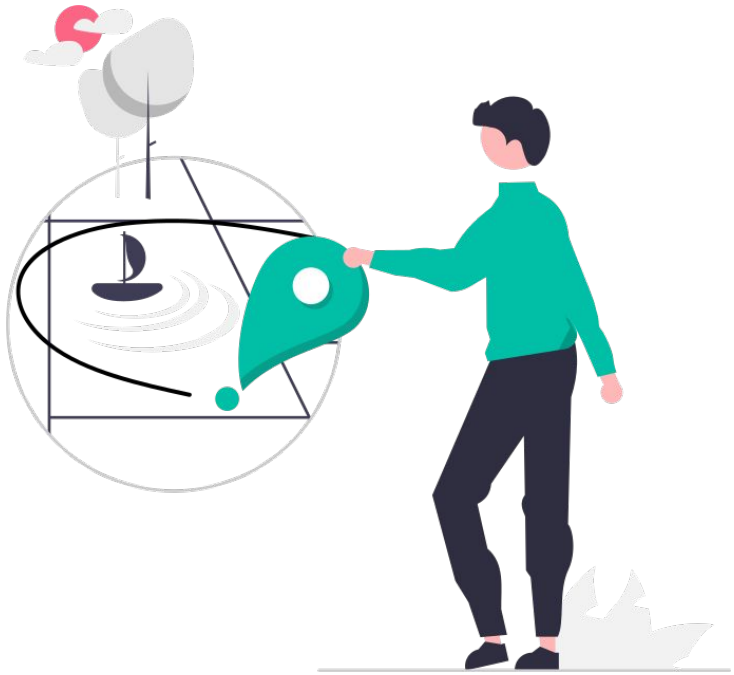
  No estás seguro de lo que es posible a alto nivel, porque el contenido de bajo nivel aún no está implementado o claro.

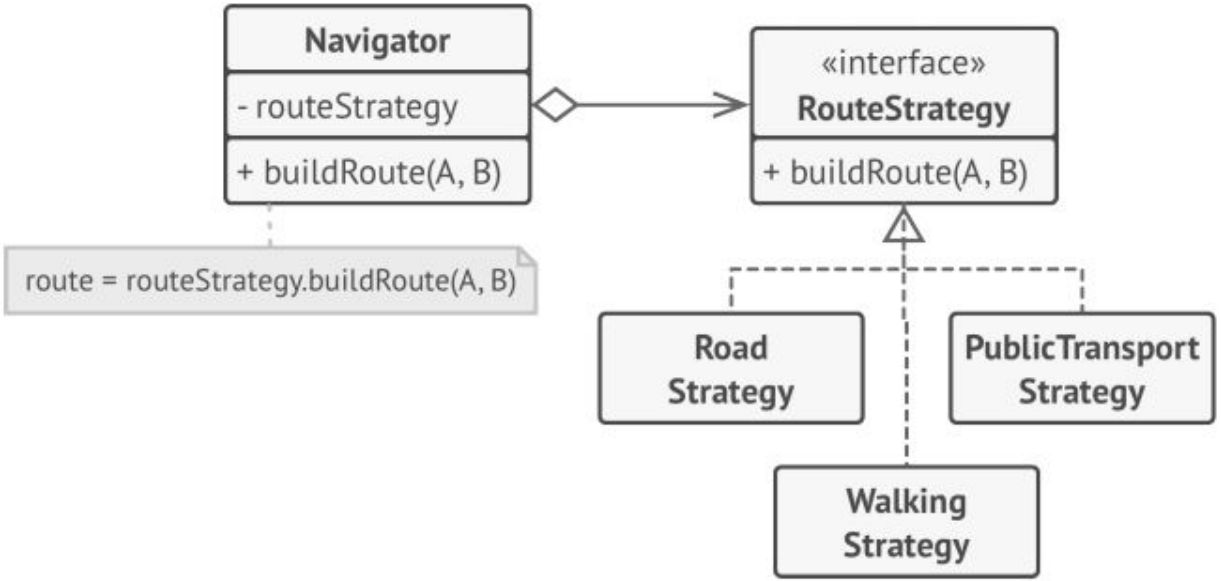
  Con este sistema, las clases de la lógica de negocio tienden a hacerse dependientes de clases primarias de bajo nivel.
-


Alto nivel
Bajo nivel
BudgetReport
- database
+ open(date)
+ save()
MySQLDatabase
...
+ insert()
+ update()
+ delete()
Alto nivel
Abstracción
BudgetReport
- database
+ open(date)
+ save()
«interface»
Database
+ insert()
+ update()
+ delete()
MySQL
...
+ insert()
+ update()
+ delete()
MongoDB
...
+ insert()
+ update()
+ delete()
Bajo nivel

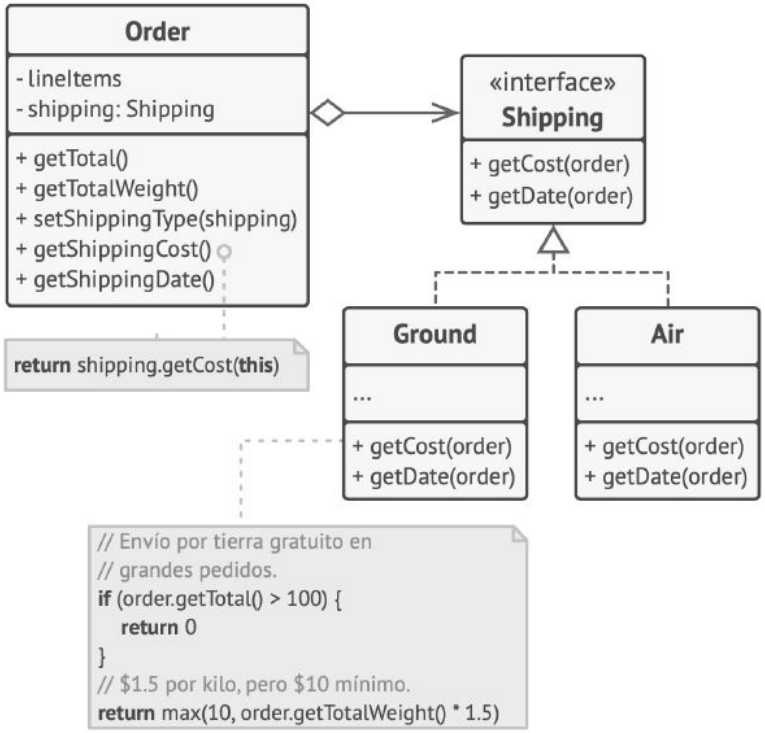
Estrategia

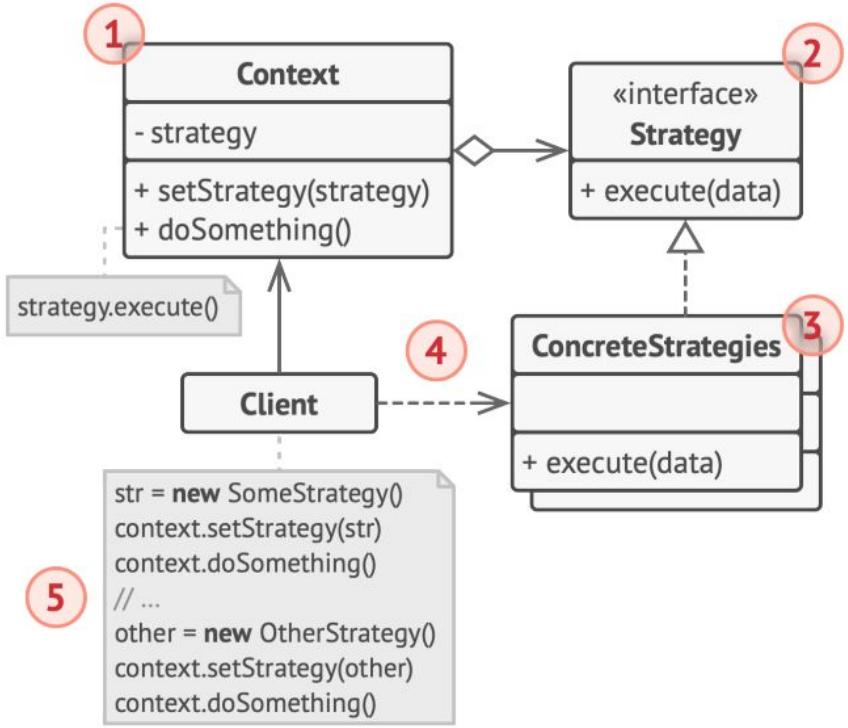






Patrón Estrategia permite definir una familia de algoritmos, extrae cada uno de ellos en una clase separada “Strategia” y hacer sus objetos intercambiables.


Order
- lineItems
- shipping: Shipping
+ getTotal()
+ getTotalWeight()
+ setShippingType(shipping)
+ getShippingCost()
+ getShippingDate()
«interface»
Shipping
+ getCost(order)
+ getDate(order)
Ground
...
+ getCost(order)
+ getDate(order)
Air
...
+ getCost(order)
+ getDate(order)
return shipping.getCost(this)
// Envío por tierra gratuito en
// grandes pedidos.
if (order.getTotal() > 100) {
return 0
}
// $1.5 por kilo, pero $10 mínimo.
return max(10, order.getTotalWeight() * 1.5)


1
Context
- strategy
+ setStrategy(strategy)
+ doSomething()
2
«interface»
Strategy
+ execute(data)
strategy.execute()
4
Client
3
ConcreteStrategies
+ execute(data)
5
str = new SomeStrategy()
context.setStrategy(str)
context.doSomething()
// ...
other = new OtherStrategy()
context.setStrategy(other)
context.doSomething()

¿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

fernando.perez@wisphub.net

fernandoprzgmz@gmail.com

fernandoprzgmz