

# Discutindo principios S.O.L.I.D. com exemplos em Python

Junho 2020



# Sumario

Apresentacao	3
Introducao	4
Principio da Responsabilidade Unica	6
Princípio Aberto-Fechado	11
Princípio da Substituição de Liskov	19
Princípio da Segregação da Interface	24
Princípio da inversão da dependência	31
Conclusao	35



# Apresentacao

- Engenheiro de Software na Loadsmart




Troque uma ideia comigo!

 [github.com/gillianomenezes](https://github.com/gillianomenezes)

 [@gillianomenezes](https://t.me/gillianomenezes)

 [@gillianomenezes](https://twitter.com/gillianomenezes)

 [linkedin.com/in/gillianomenezes/](https://linkedin.com/in/gillianomenezes/)

 [gillianomenezes@gmail.com](mailto:gillianomenezes@gmail.com)



# Introducao

- Em marco de 1995, Robert C. Martin (a.k.a. Uncle Bob), escreveu um artigo sobre um conjunto de princípios para o design de projetos orientados a objetos
- Esses principios foram a base para o S.O.L.I.D.:

<b>SRP</b>	Principio da Responsabilidade Única	Uma classe deve ter um e apenas um motivo para mudar.
<b>OCP</b>	Princípio Aberto-Fechado	Você deve poder estender um comportamento de classe, sem modificá-la.
<b>LSP</b>	Princípio da Substituição de Liskov	Classes derivadas devem ser substituíveis por suas classes base.
<b>ISP</b>	Princípio da Segregação da Interface	Muitas interfaces específicas são melhores do que uma interface única.
<b>DIP</b>	Princípio da inversão da dependência	Dependa de uma abstração e não de uma implementação.



# Motivacao

S.O.L.I.D. são princípios de design destinados a tornar o código de um software mais:

- Compreensível
- Flexível
- Facil manutenção
- Reutilizável

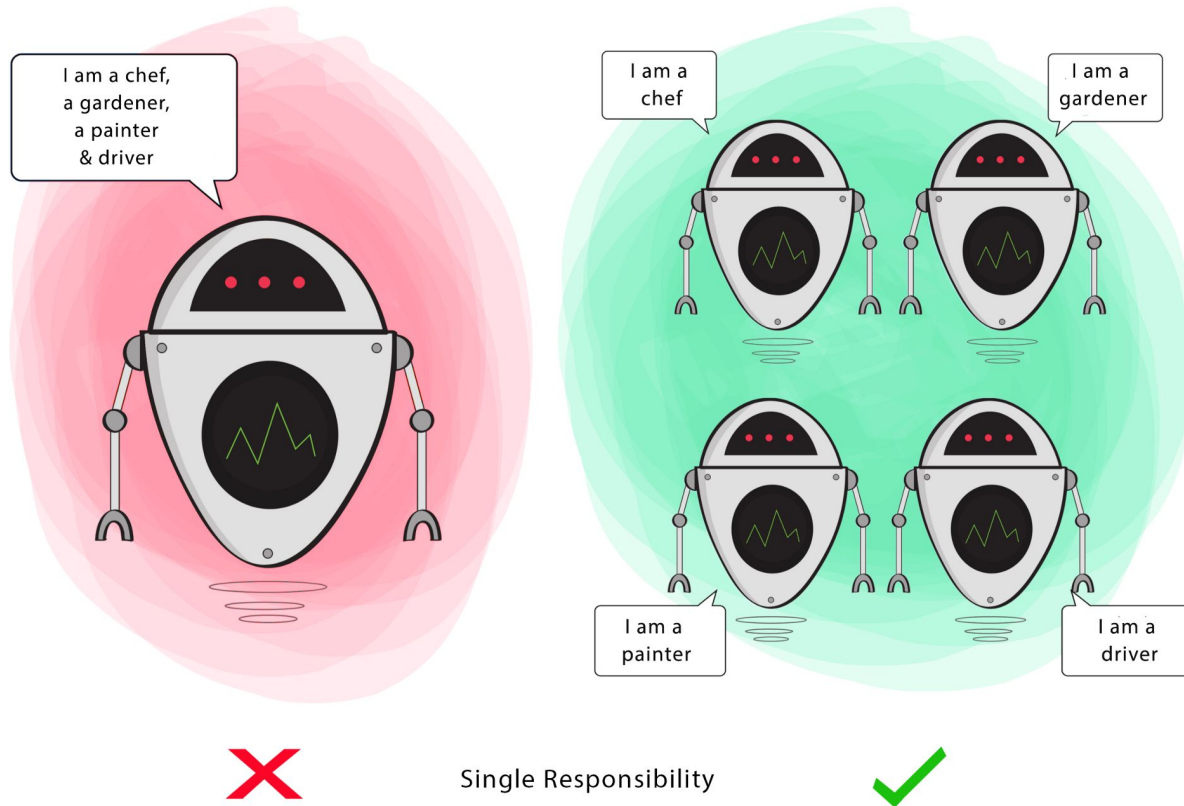


# 1. Princípio da Responsabilidade Unica



Uma classe deveria ter uma, e somente uma, razão para mudar.

# 1. Principio da Responsabilidade Unica





# 1. Princípio da Responsabilidade Unica

A classe de exemplo “Shipper” viola o Princípio da Responsabilidade Única

Como isso viola o SRP?

```
class Shipper:
    def __init__(self, name: str):
        self.name = name

    def get_name(self):
        pass

    def save(self, shipper: Shipper):
        pass
```







# 1. Princípio da Responsabilidade Única

A classe de exemplo “Shipper” viola o Princípio da Responsabilidade Única

Como esse design causará  
problemas no futuro?

```
class Shipper:
    def __init__(self, name: str):
        self.name = name

    def get_name(self):
        pass

    def save(self, shipper: Shipper):
        pass
```



# 1. Princípio da Responsabilidade Unica

## Solucao

1. Crie uma nova classe para lidar apenas com a lógica de negócios
2. Crie uma nova classe para lidar apenas com a persistência

```
class Shipper:
    def __init__(self, name: str):
        self.name = name
        self.db = ShipperDB()

    def get_name(self):
        return self.name

    def get(self, id):
        return self.db.get_shipper(id)

    def save(self):
        self.db.save(shipper=self)

class ShipperDB:
    def get_shipper(self, id):
        pass

    def save(self, shipper: Shipper):
        pass
```



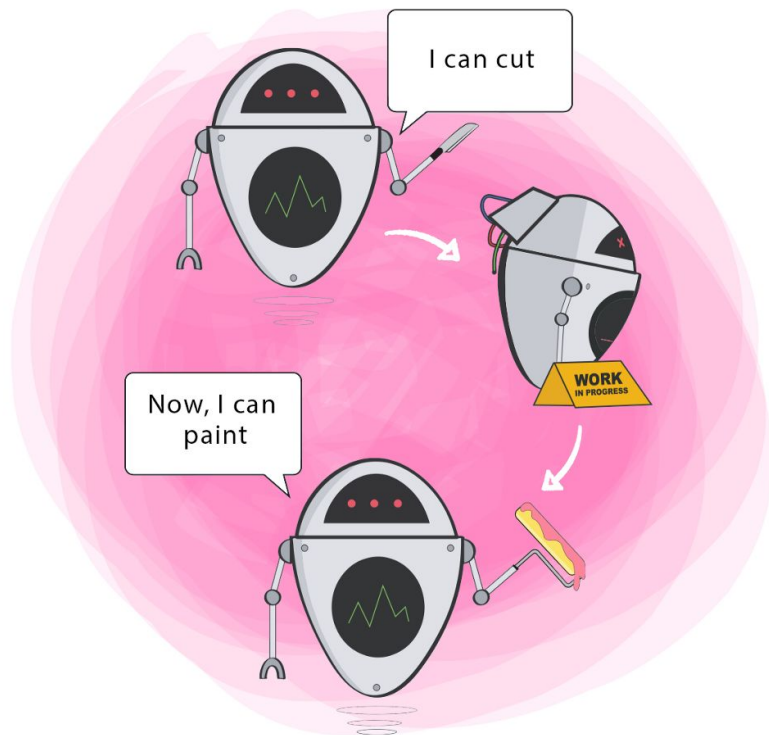


## 2. Princípio Aberto-Fechado

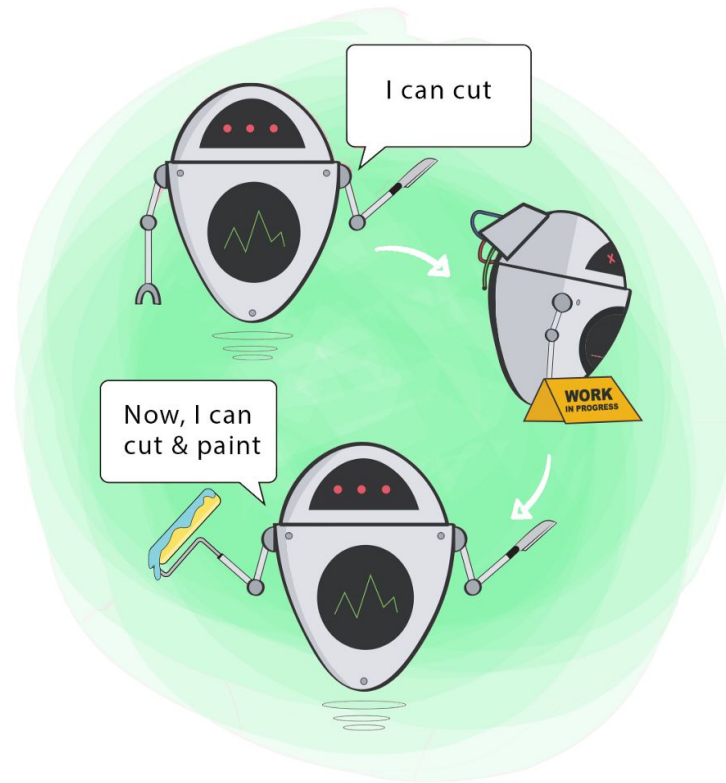


Você deve poder estender um comportamento de classe, sem modificá-la.

## 2. Princípio Aberto-Fechado



Open-Closed





## 2. Princípio Aberto-Fechado

A classe “Animal” violata o principio Aberto-Fechado

Como o OCP é violado?

```
class Animal:
    def __init__(self, name: str):
        self.name = name
    def get_name(self):
        pass

animals = [Animal('lion'), Animal('mouse')]

def animal_sound(animals: list):
    for animal in animals:
        if animal.name == 'lion':
            print('roar')
        elif animal.name == 'mouse':
            print('squeak')

animal_sound(animals)
```





## 2. Princípio Aberto-Fechado

A classe “Animal” violata o principio Aberto-Fechado

E se quisermos adicionar um novo animal?

```
class Animal:
    def __init__(self, name: str):
        self.name = name
    def get_name(self):
        pass

animals = [Animal('lion'), Animal('mouse'), Animal('snake')]

def animal_sound(animals: list):
    for animal in animals:
        if animal.name == 'lion':
            print('roar')
        elif animal.name == 'mouse':
            print('squeak')
        elif animal.name == 'snake':
            print('hiss')

animal_sound(animals)
```





# 2. Princípio Aberto-Fechado

## Solucao

```
class Animal:
    def __init__(self, name: str):
        self.name = name
    def get_name(self):
        pass
    def make_sound(self):
        Pass

def animal_sound(animals: list):
    for animal in animals:
        print(animal.make_sound())

animal_sound(animals)
```

```
class Lion(Animal):
    def make_sound(self):
        return 'roar'

class Mouse(Animal):
    def make_sound(self):
        return 'squeak'

class Snake(Animal):
    def make_sound(self):
        return 'hiss'
```





## 2. Princípio Aberto-Fechado

Outro exemplo: classe “Discount” viola o Princípio Aberto-Fechado

Como isso viola o OCP?

```
class Discount:
    def __init__(self, customer, price):
        self.customer = customer
        self.price = price

    def give_discount(self):
        if self.customer == 'fav':
            return self.price * 0.2
```







## 2. Princípio Aberto-Fechado

Outro exemplo: classe “Discount” viola o Princípio Aberto-Fechado

Como isso viola o OCP?

```
class Discount:
    def __init__(self, customer, price):
        self.customer = customer
        self.price = price

    def give_discount(self):
        if self.customer == 'fav':
            return self.price * 0.2
        if self.customer == 'vip':
            return self.price * 0.4
```





# 2. Princípio Aberto-Fechado

## Solucao

```
class Discount:
    def __init__(self, customer, price):
        self.customer = customer
        self.price = price
    def get_discount(self):
        return self.price * 0.2
```

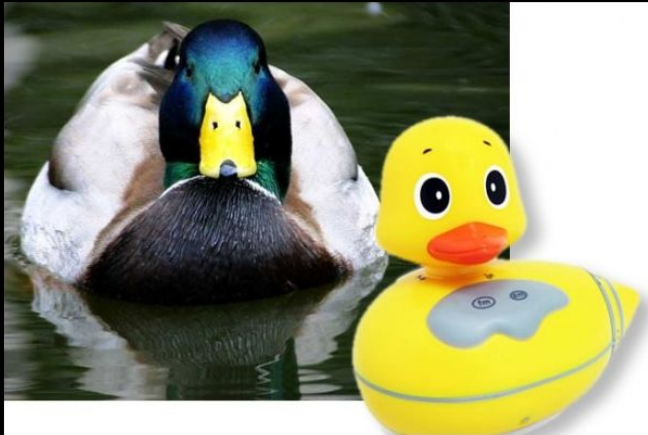
```
class VIPDiscount(Discount):
    def get_discount(self):
        return super().get_discount() * 2

class SuperVIPDiscount(VIPDiscount):
    def get_discount(self):
        return super().get_discount() * 2
```





# 3. Princípio da Substituição de Liskov

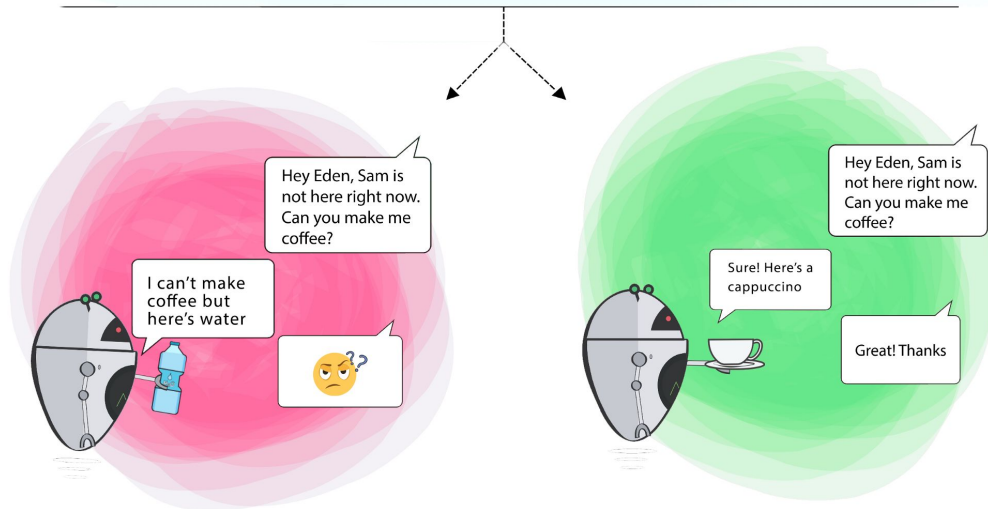
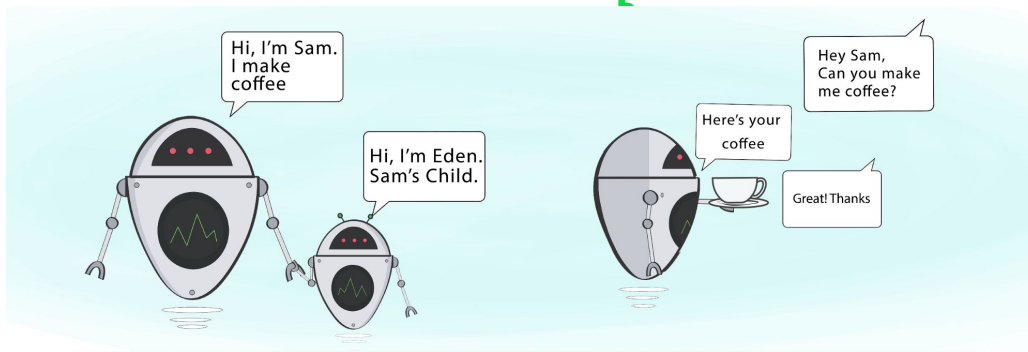


## LSP - PRINCÍPIO DE SUBSTITUIÇÃO DE LISKOV

Se parece com um pato, faz som de pato, nada como pato, mas precisa de baterias, certamente você tem uma abstração problemática

Classes derivadas devem ser substituíveis por suas classes base.

# 3. Princípio da Substituição de Liskov



Liskov Substitution





# 3. Princípio da Substituição de Liskov

A classe de exemplo "Truck" viola o Princípio da Substituição de Liskov

Como isso viola o LSP?

```
def truck_freight_cost(trucks: list):  
    for truck in trucks:  
        if isinstance(truck, Refeer):  
            print(refeer_freight_cost(truck))  
        elif isinstance(truck, Flatbed):  
            print(flatbed_freight_cost(truck))  
        elif isinstance(truck, Dryvan):  
            print(dryvan_freight_cost(truck))  
  
freight_cost(trucks)
```





# 3. Princípio da Substituição de Liskov

## Solucao

```
class Truck:
    def freigth_cost(self):
        pass
```

```
class Reefer(Truck):
    def freigth_cost(self):
        pass
```

```
class Flatbed(Truck):
    def freigth_cost(self):
        pass
```

```
class Dryvan(Truck):
    def freigth_cost(self):
        pass
```

```
def truck_freight_cost(trucks: list):
    for truck in trucks:
        print(truck.freigth_cost())

truck_freight_cost(trucks)
```





# 3. Princípio da Substituição de Liskov

## Observações

- Polimorfismo
- Princípios relacionados: LSP contribui para o OCP

# 4. Princípio da Segregação da Interface



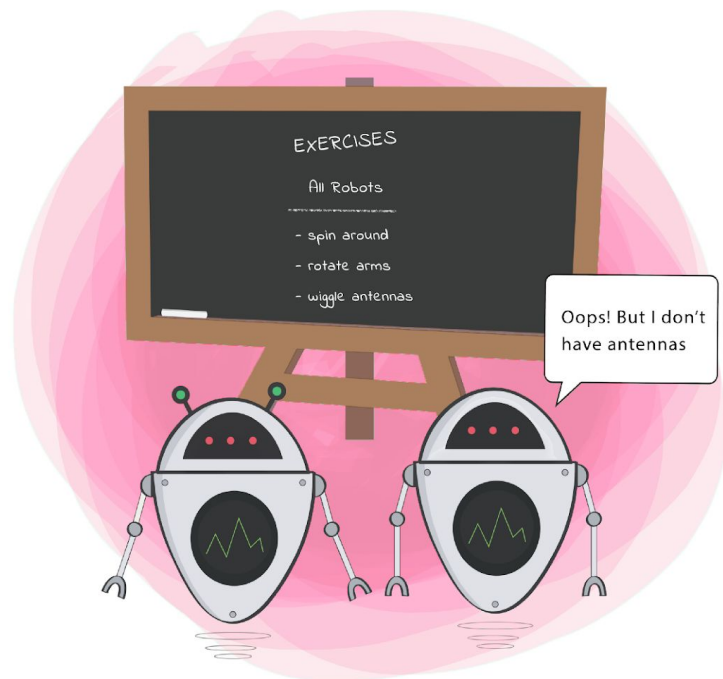
INTERFACE SEGREGATION PRINCIPLE

You Want Me To Plug This In, Where?

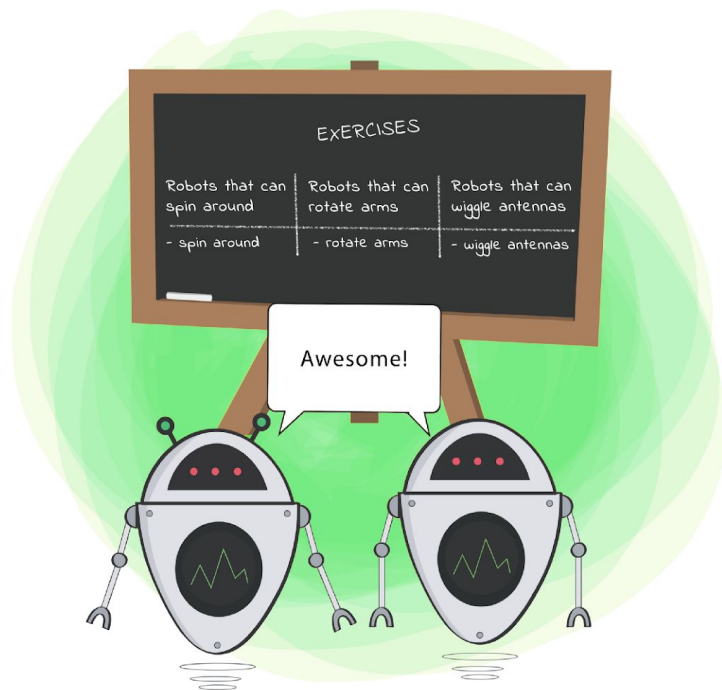
Muitas interfaces específicas são melhores do que uma interface única.



# 4. Princípio da Segregação da Interface



Interface Segregation





# 4. Princípio da Segregação da Interface

A classe de exemplo "IShape" viola o Princípio da Segregação da Interface

Como isso viola o ISP?

```
class IShape:
    def draw_square(self):
        raise NotImplementedError

    def draw_rectangle(self):
        raise NotImplementedError

    def draw_circle(self):
        raise NotImplementedError
```





# 4. Princípio da Segregação da Interface

A classe de exemplo "IShape" viola o Princípio da Segregação da Interface

```
class Circle(IShape):
    def draw_square(self):
        pass

    def draw_rectangle(self):
        pass

    def draw_circle(self):
        pass

class Square(IShape):
    def draw_square(self):
        pass

    def draw_rectangle(self):
        pass

    def draw_circle(self):
        pass
```

```
class Rectangle(IShape):
    def draw_square(self):
        pass

    def draw_rectangle(self):
        pass

    def draw_circle(self):
        pass
```





# 4. Princípio da Segregação da Interface

A classe de exemplo "IShape" viola o Princípio da Segregação da Interface

E se precisarmos adicionar  
uma classe Triangle?

```
class IShape:
    def draw_square(self):
        raise NotImplementedError

    def draw_rectangle(self):
        raise NotImplementedError

    def draw_circle(self):
        raise NotImplementedError

    def draw_triangle(self):
        raise NotImplementedError
```





# 4. Princípio da Segregação da Interface

A classe de exemplo "IShape" viola o Princípio da Segregação da Interface

```
class Circle(IShape):  
    def draw_square(self):  
        pass  
    def draw_rectangle(self):  
        pass  
    def draw_circle(self):  
        pass  
    def draw_triangle(self):  
        pass
```

```
class Square(IShape):  
    def draw_square(self):  
        pass  
    def draw_rectangle(self):  
        pass  
    def draw_circle(self):  
        pass  
    def draw_triangle(self):  
        pass
```

```
class Rectangle(IShape):  
    def draw_square(self):  
        pass  
    def draw_rectangle(self):  
        pass  
    def draw_circle(self):  
        pass  
    def draw_triangle(self):  
        pass
```

```
class Triangle(IShape):  
    def draw_square(self):  
        pass  
    def draw_rectangle(self):  
        pass  
    def draw_circle(self):  
        pass  
    def draw_triangle(self):  
        pass
```





# 4. Princípio da Segregação da Interface

A classe de exemplo "IShape" viola o Princípio da Segregação da Interface

```
class IShape:
    def draw(self):
        raise NotImplementedError
```

```
class Circle(IShape):
    def draw(self):
        pass
```

```
class Square(IShape):
    def draw(self):
        pass
```

```
class Rectangle(IShape):
    def draw(self):
        pass
```

```
class Triangle(IShape):
    def draw(self):
        pass
```



# 5. Princípio da inversão da dependência

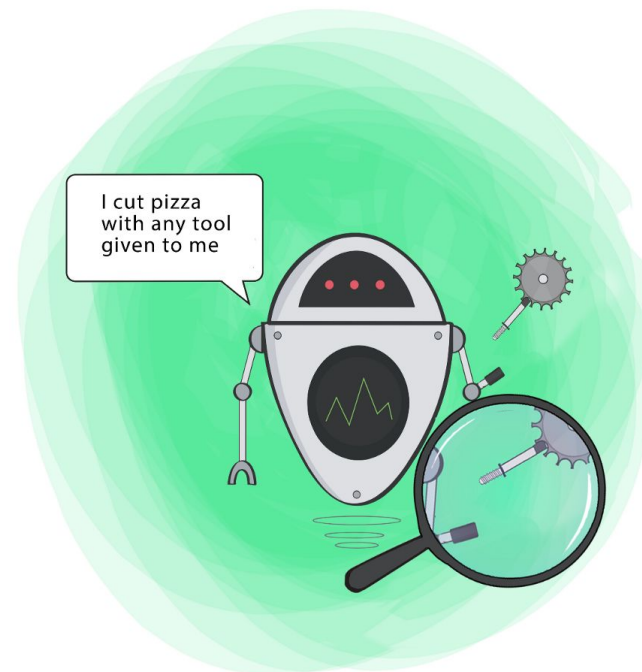
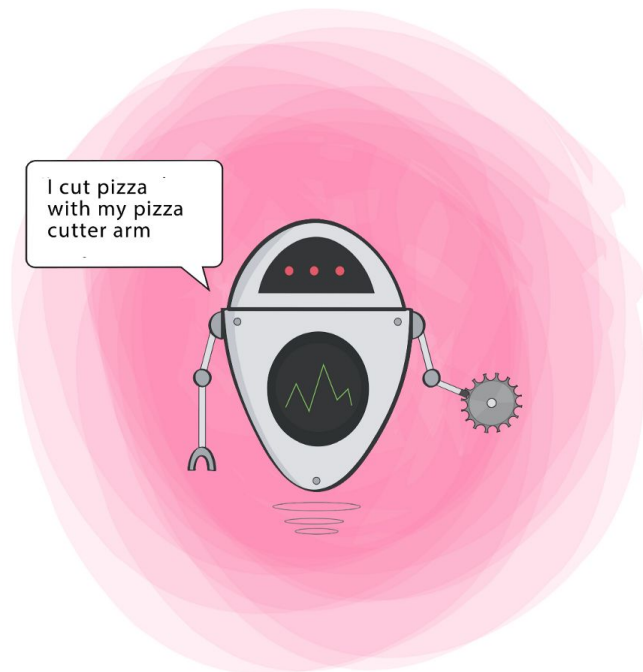


## Dependency Inversion Principle

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

Dependa de uma abstração e não de  
uma implementação.

# 5. Princípio da inversão da dependência



Dependency Inversion





# 5. Princípio da inversão da dependência

As classes de exemplo violam o Princípio da inversão da dependência

Como esse design viola o DIP?

```
class XMLHttpRequestService(XMLHttpRequestService):  
    pass  
  
class Http:  
    def __init__(self, xml_http_service:  
XMLHttpRequestService):  
        self.xml_http_service = xml_http_service  
  
    def get(self, url: str, options: dict):  
        self.xml_http_service.request(url, 'GET')  
  
    def post(self, url, options: dict):  
        self.xml_http_service.request(url, 'POST')
```





# 5. Princípio da inversão da dependência

## Solucao

```
class Connection:
    def request(self, url: str, options: dict):
        raise NotImplementedError

class Http:
    def __init__(self, http_connection: Connection):
        self.http_connection = http_connection

    def get(self, url: str, options: dict):
        self.http_connection.request(url, 'GET')

    def post(self, url, options: dict):
        self.http_connection.request(url, 'POST')
```

```
class XMLHttpRequestService(Connection):
    xhr = XMLHttpRequest()

    def request(self, url: str, options: dict):
        self.xhr.open()
        self.xhr.send()

class NodeHttpRequestService(Connection):
    def request(self, url: str, options: dict):
        pass

class MockHttpRequestService(Connection):
    def request(self, url: str, options: dict):
        pass
```





# Conclusao



- Os princípios S.O.L.I.D. são obrigatórios para um bom design de programação OO.
- Proporciona um código menos rígido e frágil sendo mais fácil de manter e extender.



- Lembre-se desses princípios ao escrever um novo código.
- Refatore o código existente para aderir ao princípios S.O.L.I.D.

# Perguntas?

# Obrigado!

