

Chain of Responsibility

Padrão de projeto para tratar um pedido ao longo de um cadeia.

O que é

Nessa aula você irá aprender um padrão de projetos comportamental, que possibilita fazer processamento de um pedido por uma sequência de processamento que lidam ou manipulam esse pedido.

Problema motivacional

Imagine que você trabalha em uma loja de eletrônicos, e que você tem um pedido de um cliente. E este pedido é de um produto raro que acabou de lançar e ainda não tem no estoque, este cliente pede para conversar com seu gerente para tratar de fazer uma encomenda e assim você o encaminha para a sala do gerente.

Chegando lá ele mostra o produto que ele deseja comprar e o gerente verifica que o produto é de uma marca internacional, e dentro da empresa ele só tem acesso aos fornecedores brasileiros, então o gerente instruí o cliente para fazer uma ligação para a central estadual.

Então o cliente faz como foi instruído e liga na central e conversa com o representante da marca no estado sobre a possibilidade de fazer este pedido raro internacional para ele, vendo que o produto é de um departamento diferente do dele, o representante encaminha a ligação do cliente para o representante nacional...

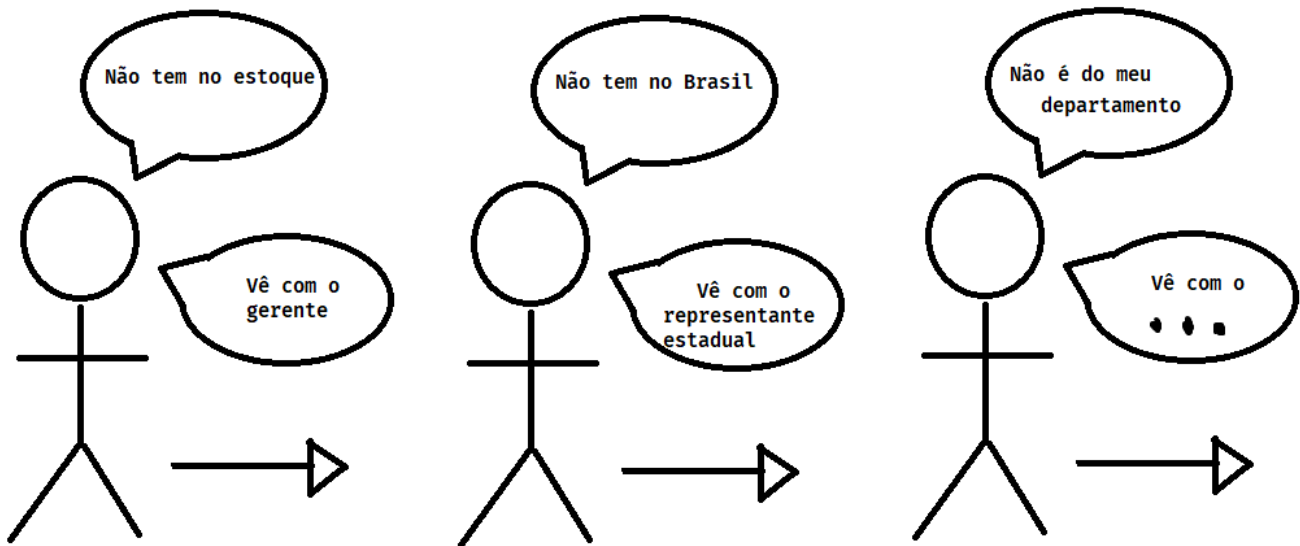
Acredito que você já pegou a ideia...

Como isso poderia ser representado em código?

```
estoque = ["Item 1", "Item 2", "Item 3"]
produtos_nacionais = ["Item 1", "Item 2", "Item 3"]
produtos_internacionais_departamento1 = ["Item 1", "Item 2", "Item 3"]
class Atendente:
    def lida_pedido(produto):
        if produto is in estoque:
            return "Pedido aceito"
        else:
            gerente = Gerente()
            gerente.lida_pedido(produto)

class Gerente:
    def lida_pedido(produto):
        if produto is in produtos_nacionais:
            return "Pedido aceito"
        else:
            representante = Representante()
            representante.lida_pedido(produto)
```

```
class RepresentanteEstadual:
    def lida_pedido(produto):
        if produto in produtos_internacionais_departamento1:
            return "Pedido aceito"
        else:
            # Continua para o representante nacional
```



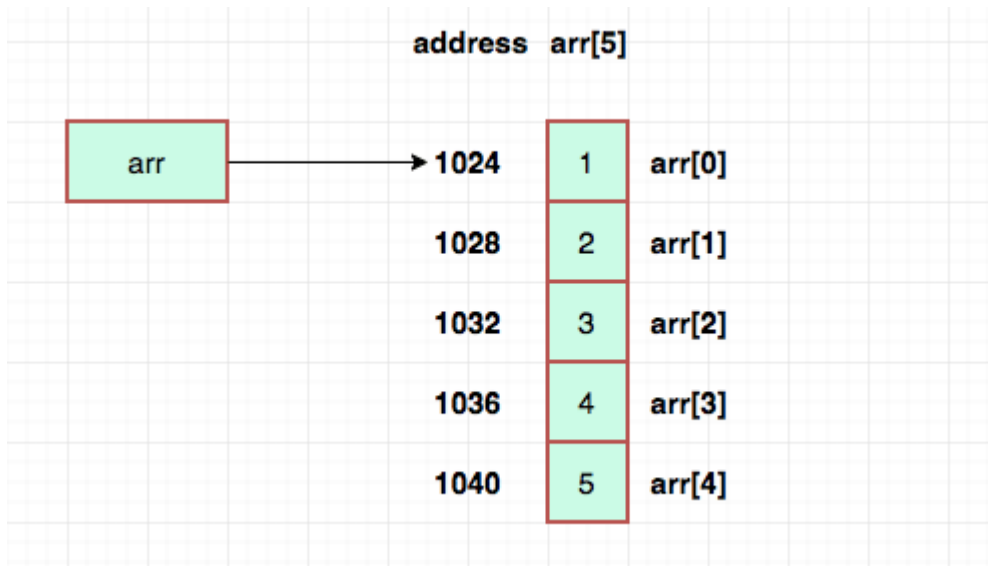
O que acabamos de criar aqui foi uma cadeia de responsabilidades, em que os funcionários dessa empresa tem como responsabilidade receber um pedido, tentar resolver o problema e entregar o pedido para o cliente, e em caso de não conseguirem resolver o problema, o funcionário passa para o próximo funcionário na cadeia.

Estrutura de dados por trás do padrão

O padrão Chain of Responsibility, é baseado na ideia de uma estrutura de dados bem famosa, a **lista encadeada**, em que nela temos um item, normalmente chamado de **nó**, que contem um dado, e a informação de onde encontrar o próximo item na memória.

Essa estrutura foi criada pois linguagens "antigas" e de **baixo nível** como o C, dão ao programador a possibilidade de criar apenas "arrays", que são blocos de memória de tamanho fixo, em que esse tamanho é definido pelo programador ao criar o array e não pode ser alterado.

```
int arr[5];
```



Inclusive essa estrutura de dados está por trás da lista que criamos em linguagens mais modernas e de alto nível, como o Python e o Javascript, que tem tamanho *dinâmico*, ou seja, podemos inserir e remover itens a qualquer momento.

```
lista = ["Item 1", "Item 2", "Item 3"]
lista.append("Item 4")
lista.remove("Item 2")
```

Material complementar:

Nesta aula não iremos aprofundar na estrutura de dados **lista encadeada**, mas é recomendado que você se aprofunde no conteúdo usando os links abaixo.

- [Introdução ao conteúdo em inglês com exemplo em várias linguagens](#)
- [Introdução ao conteúdo em português com exemplos em C](#)
- [Listas e suas operações em Python](#)
- [Diferentes tipos de listas](#)

Lista de processamento

Agora que você entendeu a estrutura de dados, ficou fácil de entender o que está acontecendo, troque o dado que é armazenado na lista por uma função que recebe como parâmetro algo, faz processamento nesse item e passa para frente (ou não) o item para continuar a ser tratado pela cadeia, e já temos tudo que precisamos para ter uma cadeia de responsabilidades.

Organização do código

O objetivo de um padrão de projeto é ser uma solução que **torna uma implementação mais fácil de entender e manter e reutilizar**, então vamos mudar o código seguindo alguns princípios de design de orientação a objetos chamado **S.O.L.I.D**:

Primeiro é possível identificar que todas as classes do exemplo eram algum tipo de funcionário, e todos funcionários têm algum tipo de forma de lidar com o pedido, então podemos criar uma

classe que represente o funcionário geral, o pai das outras classes, e que possuir um **método abstrato** para lidar com o pedido, e uma informação de quem é o seu superior, que lidará com o pedido caso esse funcionário não consiga lidar.

```
from abc import ABC, abstractmethod
# biblioteca que permite que classes sejam abstratas

class Funcionario(ABC):
    _superior = None

    @abstractmethod
    def lida_pedido(self, pedido):
        pass

    def define_superior(self, superior):
        self._superior = superior
        return superior
    # Esse retorno ajudará na hora de definir a ordem da cadeia
```

Agora que temos a classe funcionário, vamos criar uma classe que represente o vendedor, que herda de funcionário e implementa o método para lidar com o pedido.

```
class Vendedor(Funcionario):
    estoque = ["Item 1", "Item 2", "Item 3"]

    def lida_pedido(self, pedido):
        if pedido in self.estoque:
            print("Vendedor diz: Toma aqui seu pedido")
            # Tenta passar o pedido para um superior, se o mesmo existir
        else:
            if self._superior is not None:
                self._superior.lida_pedido(pedido)
            else:
                print("Não temos o que você precisa")
```

Hora do exercício

Crie uma classe que represente o gerente e uma classe que represente o representante regional, ambas devem herdar de **Funcionario** e implementa o método para lidar com o pedido.

Gabarito

```
class Gerente(Funcionario):
    estoque = ["Item 4", "Item 5", "Item 6"]
```

```
def lida_pedido(self, pedido):
    if pedido in self.estoque:
        print("Gerente diz: Toma aqui seu pedido")
    else:
        if self._superior is not None:
            self._superior.lida_pedido(pedido)
        else:
            print("Não temos o que você precisa")

class RepresentanteRegional(Funcionario):
    estoque = ["Item 7", "Item 8", "Item 9"]

    def lida_pedido(self, pedido):
        if pedido in self.estoque:
            print("Representante Regional diz: Toma aqui seu pedido")
        else:
            if self._superior is not None:
                self._superior.lida_pedido(pedido)
            else:
                print("Não temos o que você precisa")
```

Uso do código e criação da cadeia de responsabilidades

```
repre_regional = RepresentanteRegional()
gerente = Gerente()
vendedor = Vendedor()

# Como o método define_superior retorna o parametro que foi passado, é
# possível definir a cadeia assim:
vendedor.define_superior(gerente).define_superior(repre_regional)

vendedor.lida_pedido("Item 1")
# Vendedor diz: Toma aqui seu pedido

vendedor.lida_pedido("Item 9")
# Representante Regional diz: Toma aqui seu pedido
vendedor.lida_pedido("Item 10")
# Não temos o que você precisa
```

Outra forma de pensar cadeia de responsabilidade

Parabéns! Você já criou uma cadeia de responsabilidade!

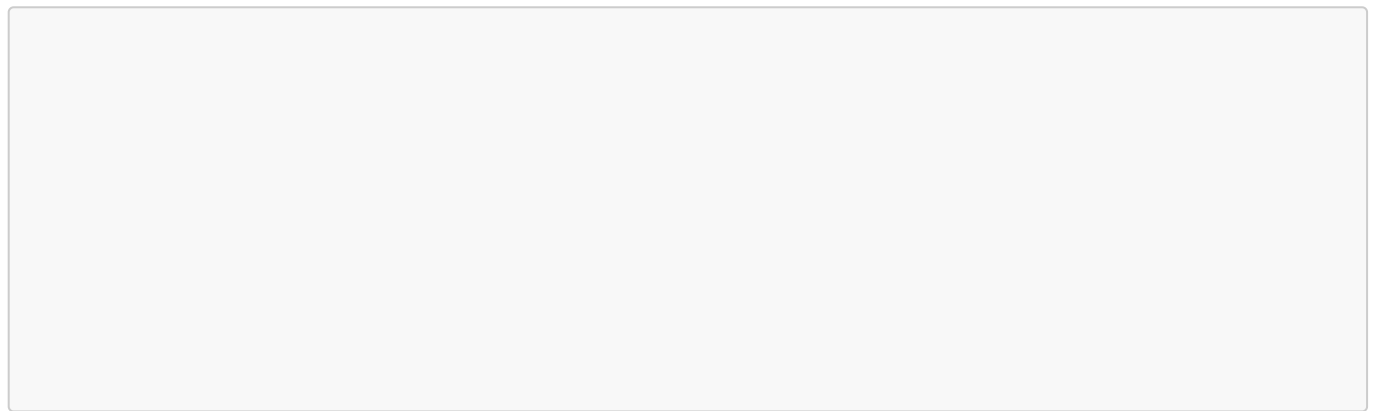
Agora tire um momento para pensar em problemas reais que a solução é por pequenos processamentos encadeados sobre um pedido, lembre-se que não necessariamente o pedido é algo constante, e existem cadeias que modificam o pedido ao longo do tratamento deste.



Jornal do Brasil, 19 de fevereiro de 1997.

Formalização do padrão

Agora que você entendeu o que é uma cadeia de responsabilidades, veja como é modelado o padrão em UML.



Itens do padrão

Handler

- Nó genérico, nele implementamos a lógica de definir o próximo nó de processamento que é genérica à todos handlers
- Não é implementada a lógica do processamento do nó (método handle), pois está é específica à cada nó

Concrete Handlers

- Classe concreta que herda a implementação de definição do próximo handler
- Deve implementar o método handle que trata o pedido que está passando pela cadeia
- O processamento do pedido é normalmente feito de duas formas:
 - Nós que **testam se um pedido está no formato esperado**, normalmente chamam a execução do próximo nó de processamento somente caso o pedido passe em seu teste
 - Nós que **manipulam um pedido durante a cadeia**, normalmente modificam certos atributos de um objeto e passam para o próximo nó modificar outros atributos

```
class IHandler(ABC):  
    nextHandler = None  
    @abstractmethod
```

```

def handle(self, request):
    raise NotImplementedError
def setNext(self, handler):
    self.nextHandler = handler
    return handler

class Handler1(IHandler):
    def handle(self, request):
        print(f"{request} tratado pelo Handler1")
        if self.nextHandler:
            self.nextHandler.handle(request)

class Handler2(IHandler):
    def handle(self, request):
        print(f"{request} tratado pelo Handler2")
        if self.nextHandler:
            self.nextHandler.handle(request)

```

Com a criação destas classes, é possível utilizar a cadeia da seguinte forma

```

# Criação do pedido
pedido = "Pedido de compra"
# Criação dos handlers
handler1 = Handler1()
handler2 = Handler2()
# Definição da ordem da cadeia de handlers
handler1.setNext(handler2)
# Tratamento do pedido pela cadeia
handler1.handle(pedido)
#Output no console:
"""
    Pedido de compra tratado pelo Handler1
    Pedido de compra tratado pelo Handler2
"""

```

Como criar uma cadeia para tratar usuários em Python

Veja a vídeo aula sobre cadeia de responsabilidades em Python

<https://drive.google.com/file/d/1PHMfG8Le3uTXAY8a4ieJ8apJmVWYBfEN/view?usp=sharing>

Gabarito do exercício do vídeo

Processamento de estar logado

```

from Handlers.IUserHandler import IUserHandler
from User import User

```

```
class LoggedHandler(IUserHandler):
    def handle(self, user: User):
        retorno: str = ""
        if user.logged == False:
            retorno += "User isn't logged\n"
        if self._next_handler:
            return retorno + self._next_handler.handle(user)
        else:
            return retorno
```

Processamento do email

```
from Handlers.IUserHandler import IUserHandler
from User import User

class EmailHandler(IUserHandler):
    def handle(self, user: User):
        retorno: str = ""
        if "@" not in user.email:
            retorno += "User's email is invalid'\n"
        if self._next_handler:
            return retorno + self._next_handler.handle(user)
        else:
            return retorno
```

Esse conteúdo parece familiar...

Caso você esteja vendo similaridades com o que já estudou anteriormente, parabéns! Os frameworks que usamos para facilitar o desenvolvimento de softwares usam amplamente padrões de projeto para prover funcionalidades da melhor forma para seus usuários.

Um framework que usa o padrão **Chain of Responsibility** em um de seus componentes é o *Express*, que você aprendeu a utilizar para criação de servidores e apis em Javascript e rodando em cima do Nodejs.

Esse padrão é utilizado pelos **Middleware**s, que são métodos que tem acesso ao request, à response de uma chamada HTTP sendo executados antes de executar o método de alguma rota particular, ou antes, de todas as rotas. Normalmente são usados no express para validação de usuários

Exemplo de middleware

```
const express = require("express")
const server = express()

const middlewareLocal = function(req, res, next){
    // processamento do nó
    console.log("Passou pelo middleware 1")
    // Chamada do proximo middleware
```



```
    next()
  }
  const middlewareGlobal = function(req,res,next){
    console.log("Passou pelo middleware global")
    next()
  }

  // Configuração do middleware que é executado antes de todas as rotas
  server.use(middlewareGlobal)

  //Rotas

  //Rota que usa o middlewareLocal
  server.get("/com",middlewareLocal, function(req,res){
    res.send("Hello World tratado localmente e globalmente")
  })
  // Rota que não usa o middleware1
  server.get("/sem",function(req,res){
    res.send("Hello World tratado globalmente")
  })

  // Execução do servidor na porta 3000
  server.listen(3000,function(){
    console.log("Server is running")
  })
```

Material complementar de middlewares no Express

- [Documentação oficial do express](#)
- [Documentação oficial de como usar no express](#)
- [Guia completo de middlewares no express](#)

Exercício

Crie uma cadeia que recebe como pedido um inteiro que representa um valor monetário em real e mostre a decomposição desse valor considerando notas de 100, 50, 20, 10, 5, 2. Cada nó da cadeia deve ser responsável por remover a quantidade de notas de R\$X,00 e printar a quantidade removida, o pedido para o próximo nó será da quantidade restante monetária

Exemplo de entrada	Exemplo de saída
576	5 nota(s) de R\$ 100.00
	1 nota(s) de R\$ 50.00
	1 nota(s) de R\$ 20.00
	0 nota(s) de R\$ 10.00
	1 nota(s) de R\$ 5.00
	0 nota(s) de R\$ 2.00

Dica: para separar as notas use resto da divisão e divisão inteira

Gabarito

```
from abc import ABC, abstractmethod

class SeparadorDeNotas(ABC):
    proximoNo = None
    @abstractmethod
    def handle(self, nota):
        raise NotImplementedError
    def setNext(self, prox_separador):
        self.proximoNo = prox_separador
        return prox_separador

class Separador100(SeparadorDeNotas):
    def handle(self, nota):
        qnt_notas = nota // 100 # divisão inteira arredondando para baixo
        print(f"{qnt_notas} nota(s) de R$ 100,00")
        restante = nota % 100 # Quantidade após remover as notas de 100 do
valor
        if self.proximoNo and restante>0:
            self.proximoNo.handle(restante)

class Separador50(SeparadorDeNotas):
    def handle(self, nota):
        qnt_notas = nota // 50
        print(f"{qnt_notas} nota(s) de R$ 50,00")
        restante = nota % 50
        if self.proximoNo and restante>0:
            self.proximoNo.handle(restante)

class Separador20(SeparadorDeNotas):
    def handle(self, nota):
        qnt_notas = nota // 20
        print(f"{qnt_notas} nota(s) de R$ 20,00")
        restante = nota % 20
        if self.proximoNo and restante>0:
            self.proximoNo.handle(restante)

class Separador10(SeparadorDeNotas):
    def handle(self, nota):
        qnt_notas = nota // 10
        print(f"{qnt_notas} nota(s) de R$ 10,00")
        restante = nota % 10
        if self.proximoNo and restante>0:
            self.proximoNo.handle(restante)

class Separador5(SeparadorDeNotas):
    def handle(self, nota):
        qnt_notas = nota // 5
        print(f"{qnt_notas} nota(s) de R$ 5,00")
        restante = nota % 5
        if self.proximoNo and restante>0:
```

```
        self.proximoNo.handle(restante)

class Separador2(SeparadorDeNotas):
    def handle(self, nota):
        qnt_notas = nota // 2
        print(f"{qnt_notas} nota(s) de R$ 2,00")
        restante = nota % 2
        if self.proximoNo and restante>0:
            self.proximoNo.handle(restante)

class Separador1(SeparadorDeNotas):
    def handle(self, nota):
        qnt_notas = nota
        print(f"{qnt_notas} nota(s) de R$ 1,00")
        restante = nota % 1
        if self.proximoNo and restante>0:
            self.proximoNo.handle(restante)

# Criação da cadeia de responsabilidades

separador_de_notas = Separador100()

separador_de_notas.setNext(Separador50()).setNext(Separador20()).setNext(Separador10()).setNext(Separador5()).setNext(Separador2()).setNext(Separador1())

# Caso "falte" alguma nota no caixa, basta remover um dos nós da cadeia
separador_de_notas.handle(576)
```