02

Muitos Descontos e o Chain of Responsibility

Começando deste ponto? Você pode fazer o DOWNLOAD (https://s3.amazonaws.com/caelum-online-public/python-dp1/stages/02-design-patterns.zip) completo do projeto do capítulo anterior e continuar seus estudos a partir deste capítulo.

Orçamento e diferentes descontos

Nosso orçamento pode receber um desconto de acordo com o tipo da venda que será efetuada. Por exemplo, se o cliente comprou mais de 5 itens, ele recebe 10% de desconto; se o total do orçamento for maior que R$ 500,00, ele recebe 7% de desconto, e assim por diante. A ideia é aplicar apenas um desconto de uma corrente (*chain*) de descontos.

O primeiro passo será criarmos uma nova classe `Orcamento` para este capítulo, com a diferença de que ela terá uma lista de itens. Cada item será uma instância da classe `Item`, declarada no mesmo arquivo `orcamento.py`:

```
# -*- coding: UTF-8 -*-
# orcamento.py
class Orcamento(object):

    def __init__(self):
        self.__itens = []

    # quando a propriedade for acessada, ela soma cada item retornando o valor do orçamento
    @property
    def valor(self):
        total = 0.0
        for item in self.__itens:
            total+= item.valor
        return total

    # retornamos uma tupla, já que não faz sentido alterar os itens de um orçamento
    def obter_itens(self):

        return tuple(self.__itens)

    # perguntamos para o orçamento o total de itens
    @property
    def total_itens(self):
        return len(self.__itens)

    def adiciona_item(self, item):
        self.__itens.append(item)

# um item criado não pode ser alterado, suas propriedades são somente leitura
class Item(object):

    def __init__(self, nome, valor):
        self.__nome = nome
        self.__valor = valor

    @property
    def valor(self):
        return self.__valor
```

```
    @property
    def nome(self):
        return self.__nome
```

O problema da solução procedural

Em uma implementação tipicamente procedural, teríamos algo do tipo:

```
# -*- coding: UTF-8 -*-
# calculador_de_descontos.py
class Calculador_de_descontos(object):

    def calcula(self, orcamento):

        if orcamento.total_itens > 5:
            return orcamento.valor * 0.1

        elif orcamento.valor > 500:
            return orcamento.valor * 0.07

        # outras possíveis regras aqui

if __name__ == '__main__':

    from orcamento import Orcamento, Item

    orcamento = Orcamento()
    orcamento.adiciona_item(Item('Item A', 100.0))
    orcamento.adiciona_item(Item('Item B', 50.0))
    orcamento.adiciona_item(Item('Item C', 400.0))

    calculador_de_descontos = Calculador_de_descontos()
    desconto = calculador_de_descontos.calcula(orcamento)
    print 'Desconto calculado : %s' % (desconto)
    # imprime 38.5
```

Separando melhor as responsabilidades

Ver esse monte de `if` s em sequência nos lembra o capítulo anterior, no qual extraímos cada um deles para uma classe específica. Vamos repetir o feito, já que com classes menores, o código se torna mais simples de entender:

```
# -*- coding: UTF-8 -*-
# descontos.py
class Desconto_por_cinto_itens(object):

  def calcular(orcamento):

    if orcamento.total_itens > 5:
        return orcamento.valor * 0.1
    else:
      # se não segue a regra, o desconto é zero!
      return 0
```

```
class Desconto_por_mais_de_quinhentos_reais(object):

  def calcular(orcamento):

    if orcamento.valor > 500:
      return orcamento.valor * 0.07
    else:
      # se não segue a regra, o desconto é zero
      return 0
```

Veja que tivemos sempre que colocar um `return 0`, afinal o método precisa sempre retornar um desconto calculado, mesmo que não exista (zero). Vamos agora substituir o conjunto de `if`s na classe `Calculador_de_desconto`. Repare que essa classe procura pelo desconto que deve ser aplicado; caso o anterior não seja válido, tenta o próximo.

```
# -*- coding: UTF-8 -*-
# calculador_de_descontos.py
from descontos import Desconto_por_cinco_itens, Desconto_por_mais_de_quinhentos_reais

class Calculador_de_descontos(object):

    def calcula(self, orcamento):

        desconto = Desconto_por_cinco_itens().calcular(orcamento)
        if desconto == 0:
            desconto = Desconto_por_mais_de_quinhentos_reais().calcular(orcamento)

        # mais um if que testa se desconto é 0 e aplica aplica o desconto

        return desconto

        # outras possíveis regras aqui

if __name__ == '__main__':

    from orcamento import Orcamento, Item

    orcamento = Orcamento()
    orcamento.adiciona_item(Item('Item A', 100.0))
    orcamento.adiciona_item(Item('Item B', 50.0))
    orcamento.adiciona_item(Item('Item C', 400.0))

    calculador_de_descontos = Calculador_de_descontos()
    desconto = calculador_de_descontos.calcula(orcamento)
    print 'Desconto calculado : %s' % (desconto)
    # imprime 38.5
```

E se novos descontos aparecerem?

O código já está um pouco melhor. Cada regra de negócio está em sua respectiva classe. O problema agora é como fazer essa sequência de descontos ser aplicada na ordem, pois precisamos colocar mais um `if` sempre que um novo desconto aparecer.

Precisávamos fazer com que um desconto qualquer, caso não deva ser executado, automaticamente passe para o próximo, até encontrar um que faça sentido. Algo do tipo:

```
# -*- coding: UTF-8 -*-
# não entra em nenhum lugar ainda, apenas ilustrativo
class Desconto_por_cinco_itens(object):

  def calcular(orcamento):

    if orcamento.total_itens > 5:
        return orcamento.valor * 0.1
    else:
      ## retorna próximo desconto
```

Alterando a classe:

```
# -*- coding: UTF-8 -*-
# descontos.py
class Desconto_por_cinco_itens(object):

  def __init__(self, proximo_desconto):
    self._proximo_desconto = proximo_desconto

  def calcular(orcamento):

    if orcamento.total_itens > 5:
        return orcamento.valor * 0.1
    else:
      return self._proximo_desconto.calcular(orcamento)

class Desconto_por_mais_de_quinhentos_reais(object):

 def __init__(self, proximo_desconto):
    self._proximo_desconto = proximo_desconto

  def calcular(orcamento):

    if orcamento.valor > 500:
      return orcamento.valor * 0.07
    else:
      return self._proximo_desconto.calcular(orcamento)
```

Ou seja, se o orçamento atende a regra de um desconto, o mesmo já calcula o desconto. Caso contrário, ele passa para o "próximo" desconto, qualquer que seja esse próximo desconto.

Criando uma cadeia de descontos

Basta agora plugarmos todas essas classes juntas. Veja que um desconto recebe um "próximo". Para o desconto, pouco importa qual é o próximo desconto. Eles estão totalmente desacoplados um do outro!

```
# -*- coding: UTF-8 -*-
# calculador_de_descontos.py
from descontos import Desconto_por_cinco_itens, Desconto_por_mais_de_quinhentos_reais, Sem_desconto
```

```
class Calculador_de_descontos(object):


    def calcula(self, orcamento):

        desconto = Desconto_por_cinco_itens(
            Desconto_por_mais_de_quinhentos_reais()
        )

        return desconto.calcula(orcamento)

if __name__ == '__main__':

    from orcamento import Orcamento, Item

    orcamento = Orcamento()
    orcamento.adiciona_item(Item('Item A', 100.0))
    orcamento.adiciona_item(Item('Item B', 50.0))
    orcamento.adiciona_item(Item('Item C', 400.0))

    calculador_de_descontos = Calculador_de_descontos()
    desconto = calculador_de_descontos.calcula(orcamento)
    print 'Desconto calculado : %s' % (desconto)
```

O problema é que nosso código não funcionará, receberemos o erro:

```
  File "calculador_de_descontos.py", line 10, in calcula
    Desconto_por_mais_de_quinhentos_reais()
TypeError: __init__() takes exactly 2 arguments (1 given)
```

Nosso `Desconto_por_mais_de_quinhentos_reais` precisa receber em seu construtor um próximo desconto, o problema é que ele é o último da corrente de descontos e não há mais nenhum depois dele. Como resolver? Simples, criaremos uma classe de desconto que não possui próximo desconto. Para deixar clara sua finalidade, seu nome será `Sem_Desconto`. Ainda será necessário implementar o `calcula`, só que dessa vez ele retornará zero (ou utilizar a instrução `pass`, se você preferir):

```
# -*- coding: UTF-8 -*-
class Desconto_por_cinco_itens(object):

  def __init__(self, proximo_desconto):
    self.__proximo_desconto = proximo_desconto

  def calcula(self, orcamento):

    if orcamento.total_itens > 5:
        return orcamento.valor * 0.1
    else:
      return self.__proximo_desconto.calcula(orcamento)

class Desconto_por_mais_de_quinhentos_reais(object):

  def __init__(self, proximo_desconto):
    self.__proximo_desconto = proximo_desconto
```

```
    def calcula(self, orcamento):

      if orcamento.valor > 500:
        return orcamento.valor * 0.07
      else:
        return self.proximo_desconto.calcula(orcamento)

  class Sem_desconto(object):

    def calcula(self, orcamento):

      return 0
```

Agora podemos alterar nosso teste e verificar que tudo sai conforme o esperado:

```
# -*- coding: UTF-8 -*-
from descontos import Desconto_por_cinco_itens, Desconto_por_mais_de_quinhentos_reais, Sem_desconto

class Calculador_de_descontos(object):


    def calcula(self, orcamento):

        desconto = Desconto_por_cinco_itens(
            Desconto_por_mais_de_quinhentos_reais(
                Sem_desconto()
            )
        )

        return desconto.calcula(orcamento)

if __name__ == '__main__':

    from orcamento import Orcamento, Item

    orcamento = Orcamento()
    orcamento.adiciona_item(Item('Item A', 100.0))
    orcamento.adiciona_item(Item('Item B', 50.0))
    orcamento.adiciona_item(Item('Item C', 400.0))

    calculador_de_descontos = Calculador_de_descontos()
    desconto = calculador_de_descontos.calcula(orcamento)
    print 'Desconto calculado : %s' % (desconto)
    # imprime 38.5
```

Esses descontos formam como se fosse uma "corrente", ou seja, um ligado ao outro. Daí o nome do padrão de projeto: **Chain of Responsibility**. A ideia do padrão é resolver problemas como esses: de acordo com o cenário, devemos realizar alguma ação. Ao invés de escrevermos código procedural, e deixarmos um único método descobrir o que deve ser feito, quebramos essas responsabilidades em várias diferentes classes, e as unimos como uma corrente.