02

Comportamentos compostos por outros comportamentos e o Decorator

Começando deste ponto? Você pode fazer o DOWNLOAD (https://s3.amazonaws.com/caelum-online-public/python-dp1/stages/04-design-patterns.zip) completo do projeto do capítulo anterior e continuar seus estudos a partir deste capítulo.

Voltando ao problema dos impostos diferenciados

Nos capítulos anteriores resolvermos questões que dizem respeito à aplicação de impostos com cálculos diferenciados. Um exemplo desses cálculos, é dado pelo código abaixo:

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

    def realiza_calculo(self, orcamento, imposto):
        valor = imposto.calcula(orcamento)
        print valor

if __name__ == '__main__':

    from orcamento import Orcamento, Item
    from impostos import ISS, ICMS, ICPP, IKCV

    orcamento = Orcamento()
    orcamento.adiciona_item(Item('ITEM 1', 50))
    orcamento.adiciona_item(Item('ITEM 2', 200))
    orcamento.adiciona_item(Item('ITEM 3', 250))

    calculador_de_impostos = Calculador_de_impostos()
    print 'ISS e ICMS'
    calculador_de_impostos.realiza_calculo(orcamento, ISS())
    calculador_de_impostos.realiza_calculo(orcamento, ICMS())

    print 'ICPP e IKCV'
    calculador_de_impostos.realiza_calculo(orcamento, ICPP())
    calculador_de_impostos.realiza_calculo(orcamento, IKCV())
```

Impostos compostos e o ônus de novas classes

No entanto, o exemplo demonstra um simples cálculo. Em muitos projetos, pode ser necessário criar comportamentos que sejam compostos por outros comportamentos. Um exemplo seria calcularmos o ICMS em cima do ISS, como no código abaixo:

```
# -*- coding: UTF-8 -*-
# calculador_de_impostos.py
# apenas exemplo, não funciona porque não temos as novas classes de impostos ainda
class Calculador_de_impostos(object):

    def realiza_calculo(self, orcamento, imposto):
        valor = imposto.calcula(orcamento)
        print valor
```

```
if __name__ == '__main__':

    from orcamento import Orcamento, Item
    from impostos import ISS, ICMS, ICPP, IKCV

    orcamento = Orcamento()
    orcamento.adiciona_item(Item('ITEM 1', 50))
    orcamento.adiciona_item(Item('ITEM 2', 200))
    orcamento.adiciona_item(Item('ITEM 3', 250))

    calculador_de_impostos = Calculador_de_impostos()
    print 'ISS e ICMS'
    calculador_de_impostos.realiza_calculo(orcamento, ISS())
    calculador_de_impostos.realiza_calculo(orcamento, ICMS())

    # novidade aqui! Mais um imposto, só que dessa vez combina ISS com ICMS
    print 'ISS_com_ICMS'
    calculador_de_impostos.realiza_calculo(orcamento,  ISS_com_ICMS()) # classe não existe ainda, se

    print 'ICPP e IKCV'
    calculador_de_impostos.realiza_calculo(orcamento, ICPP()) # imprime 25.0
    calculador_de_impostos.realiza_calculo(orcamento, IKCV()) # imprime 30.0

    # novidade aqui! Mais um imposto, só que dessa vez combina ICPP com IKVC
    print 'ICPP_com_IKVC'
    calculador_de_impostos.realiza_calculo(orcamento,  ICPP_com_IKVC()) # classe não existe ainda, s
```

Repare que para isso, precisaremos criar uma nova classe, chamada `ISS_com_ICMS`. A codificação dessa classe seria uma simples sequência de calculos envolvendo os dois impostos. No entanto, o mais importante a se notar nesse ponto, é o fato de que sempre que quisermos cálculos compostos de impostos, temos que criar uma nova classe. Ou seja, se quisermos calcular ISS com o hipotético ICCP, criaríamos a classe `ISS_com_ICCP`. Imagine se tivermos 3 impostos: serão 8 combinações; 4 impostos já seriam 16! Trabalhoso demais de implementar! Com isso, percebemos que nosso código é pouco flexível, pois teríamos que saber de antemão as várias combinações possíveis de impostos e criarmos uma classe para cada.

Criando novos impostos sem novas classes

Pensando no momento do uso das classes de impostos e tendo em vista que queremos ter diferentes combinações de impostos sem precisarmos criar classes novas, poderíamos fazer os impostos, **opcionalmente** depender de outro imposto. Assim, teríamos algo como:

```
# -*- coding: UTF-8 -*-
# calculador_de_impostos.py
# apenas exemplo, não funciona porque não temos as novas classes de impostos ainda
class Calculador_de_impostos(object):

    def realiza_calculo(self, orcamento, imposto):
        valor = imposto.calcula(orcamento)
        print valor

if __name__ == '__main__':

    from orcamento import Orcamento, Item
```

```
from impostos import ISS, ICMS, ICPP, IKCV

orcamento = Orcamento()
orcamento.adiciona_item(Item('ITEM 1', 50))
orcamento.adiciona_item(Item('ITEM 2', 200))
orcamento.adiciona_item(Item('ITEM 3', 250))

calculador_de_impostos = Calculador_de_impostos()
print 'ISS e ICMS'
calculador_de_impostos.realiza_calculo(orcamento, ISS())
calculador_de_impostos.realiza_calculo(orcamento, ICMS())

print 'ISS_com_ICMS'
# veja, não é necessária mais uma classe. Imposto recebe outro em seu construtor
ISS_com_ICMS = ISS(ICMS())
calculador_de_impostos.realiza_calculo(orcamento, ISS_com_ICMS)

print 'ICPP e IKCV'
calculador_de_impostos.realiza_calculo(orcamento, ICPP()) # imprime 25.0
calculador_de_impostos.realiza_calculo(orcamento, IKCV()) # imprime 30.0

print 'ICPP_com_IKCV'
# veja, não é necessária mais uma classe. Imposto recebe outro em seu construtor
ICPP_com_IKCV = ICPP(IKVC())
calculador_de_impostos.realiza_calculo(orcamento, ICPP_com_IKVC)
```

No código acima, note que não precisamos criar mais nenhuma classe e apenas compomos os comportamentos com as que já possuíamos. Mas o código acima ainda não funciona! A classe `ISS` precisa receber outros impostos. E se quiséssemos fazer o processo inverso, ou seja, calcular o ISS, em cima do ICMS, precisaríamos que a classe `ICMS` recebesse também outros impostos. E pela flexibilidade, deveria poder ser recebido **qualquer** imposto.

Compondo impostos

Com isso, percebemos que em nosso caso, **todos** os impostos, podem receber outros impostos. Podemos explicitar isso através de construtores na classe **Imposto**. No lugar de alterar cada uma de nossas classes de imposto para suportar um novo imposto em seu construtor, podemos criar uma classe abstrata com o construtor que temos interesse, inclusive podemos também tornar o método `calcular` abstrato, assim, ninguém poderá se esquecer de implementá-lo nas classes filhas, isto é, uma rigidez que nos ajuda:

```
# -*- coding: UTF-8 -*-
# impostos.py
# declarado no início do arquivo!

from abc import ABCMeta, abstractmethod

class Imposto(object):
    __metaclass__ = ABCMeta

    def __init__(self, outro_imposto = None):
        self.__outro_imposto = outro_imposto

    @abstractmethod
    def calcula(self, orcamento):
```

```
        pass
# classes posteriores omitidas
```

O próximo passo é fazermos as classes filhas de **Imposto** . Veja que a classe `Template_de_imposto_condicional` que fizemos no capítulo de *Template method* também herda de `Imposto` :

```
#classe Imposto está em cima

class Template_de_imposto_condicional(Imposto):

    __metaclass__ = ABCMeta

    def calcula(self, orcamento):
        if self.deve_usar_maxima_taxacao(orcamento):
            return self.maxima_taxacao(orcamento)
        else:
            return self.minima_taxacao(orcamento)

    @abstractmethod
    def deve_usar_maxima_taxacao(self, orcamento): pass

    @abstractmethod
     def maxima_taxacao(self, orcamento): pass

    @abstractmethod
    def minima_taxacao(self, orcamento): pass

class ICPP(Template_de_imposto_condicional):

    def deve_usar_maxima_taxacao(self, orcamento):
        return orcamento.valor > 500

    def maxima_taxacao(self, orcamento):
        return orcamento.valor * 0.07

    def minima_taxacao(self, orcamento):
        return orcamento.valor * 0.05


class IKCV(Template_de_imposto_condicional):

    def deve_usar_maxima_taxacao(self, orcamento):
        return orcamento.valor > 500 and self.__tem_item_maior_que_100_reais(orcamento)

    def maxima_taxacao(self, orcamento):
        return orcamento.valor * 0.10

    def minima_taxacao(self, orcamento):
        return orcamento.valor * 0.06

    def __tem_item_maior_que_100_reais(self, orcamento):
        for item in orcamento.obter_itens():
            if item.valor > 100:
                return True
        return False
```

```
class ICMS(Imposto):

    def calcula(self, orcamento):
        return orcamento.valor * 0.1

class ISS(Imposto):

    def calcula(self, orcamento):
        return orcamento.valor * 0.06
```

Nesse momento, já conseguimos fazer com que os impostos recebam outros impostos, mas ainda não estamos compondo os cálculos. Nesse caso do ISS, queremos que o `calcula`, devolva os 6% do orçamento **mais** o resultado do cálculo do outro imposto. A mesma coisa com os outros impostos. Com isso, nosso método `calcula` precisa também do outro imposto, então, vamos utilizá-lo:

```
#classe Imposto la em cima

class Template_de_imposto_condicional(Imposto):

    __metaclass__ = ABCMeta

    def calcula(self, orcamento):
        if self.deve_usar_maxima_taxacao(orcamento):
            return self.maxima_taxacao(orcamento) + self.calculo_do_outro_imposto(orcamento)
        else:
            return self.minima_taxacao(orcamento) + self.calculo_do_outro_imposto(orcamento)

    @abstractmethod
    def deve_usar_maxima_taxacao(self, orcamento): pass

    @abstractmethod
     def maxima_taxacao(self, orcamento): pass

    @abstractmethod
    def minima_taxacao(self, orcamento): pass

class ICPP(Template_de_imposto_condicional):

    def deve_usar_maxima_taxacao(self, orcamento):
        return orcamento.valor > 500

    def maxima_taxacao(self, orcamento):
        return orcamento.valor * 0.07

    def minima_taxacao(self, orcamento):
        return orcamento.valor * 0.05


class IKCV(Template_de_imposto_condicional):

    def deve_usar_maxima_taxacao(self, orcamento):
        return orcamento.valor > 500 and self.__tem_item_maior_que_100_reais(orcamento)

    def maxima_taxacao(self, orcamento):
        return orcamento.valor * 0.10
```

```
    def minima_taxacao(self, orcamento):
        return orcamento.valor * 0.06

    def __tem_item_maior_que_100_reais(self, orcamento):
        for item in orcamento.obter_itens():
            if item.valor > 100:
                return True
        return False

class ISS(Imposto):

    def calcula(self, orcamento):
        return orcamento.valor * 0.1 + self.calculo_do_outro_imposto(orcamento)

class ICMS(Imposto):

    def calcula(self, orcamento):
        return orcamento.valor * 0.06 + self.calculo_do_outro_imposto(orcamento)
```

Esse método `calculo_do_outro_imposto` será o mesmo em todos os impostos. Para evitar repetição de código, ele pode estar na classe `Imposto`, garantindo que todos que herdam de `Imposto` possam chamá-lo para delegar para o outro imposto, casa ele tenha sido definido. Nesse caso, a implementação na classe `Imposto` seria:

```
# -*- coding: UTF-8 -*-

from abc import ABCMeta, abstractmethod

class Imposto(object):
    __metaclass__ = ABCMeta

    def __init__(self, outro_imposto = None):
        self.__outro_imposto = outro_imposto

    def calculo_do_outro_imposto(self, orcamento):
        return self.__outro_imposto.calcula(orcamento)

    @abstractmethod
    def calcula(self, orcamento):
        pass
```

Mas e no final, quando não existirem mais impostos a serem calculados? O outro_imposto será nulo? Devemos permitir que um imposto não tenha mais um próximo imposto a ser calculado. Para isso, vamos adicionar um construtor default na classe `Imposto`, e fazer com que o método `calculo_do_outro_imposto` agora trate o caso de não haver um próximo:

```
# -*- coding: UTF-8 -*-

from abc import ABCMeta, abstractmethod

class Imposto(object):
    __metaclass__ = ABCMeta

    def __init__(self, outro_imposto = None):
        self.__outro_imposto = outro_imposto
```

```
    def calculo_do_outro_imposto(self, orcamento):
        if (self.__outro_imposto is None):
            return 0
        else:
            return self.__outro_imposto.calcula(orcamento)

    @abstractmethod
    def calcula(self, orcamento):
        pass
```

Repare que o método `calculo_do_outro_imposto` invoca o método `calcula` caso outro imposto tenha sido definido. Caso contrário, apenas retorna 0, não influenciando na composição do cálculo.

Pronto, agora conseguimos inclusive compor o comportamento. Ao executarmos o código `ISS(ICMS()` o `ICMS` será guardado como o outro imposto e quando o cálculo for realizado, o método `calculo_do_outro_Imposto` que está sendo chamado no `calcula` se encarregará de invocar o cálculo do outro imposto. Não acredita? Teste:

```
python calculador_de_impostos.py
```

```
# -*- coding: UTF-8 -*-
# calculador_de_impostos.py
# apenas exemplo, não funciona porque não temos as novas classes de impostos ainda
class Calculador_de_impostos(object):

    def realiza_calculo(self, orcamento, imposto):
        valor = imposto.calcula(orcamento)
        print valor

if __name__ == '__main__':

    from orcamento import Orcamento, Item
    from impostos import ISS, ICMS, ICPP, IKCV

    orcamento = Orcamento()
    orcamento.adiciona_item(Item('ITEM 1', 50))
    orcamento.adiciona_item(Item('ITEM 2', 200))
    orcamento.adiciona_item(Item('ITEM 3', 250))

    calculador_de_impostos = Calculador_de_impostos()
    print 'ISS e ICMS'
    calculador_de_impostos.realiza_calculo(orcamento, ISS()) # imprime 50.0
    calculador_de_impostos.realiza_calculo(orcamento, ICMS()) # imprime 30.0

    print 'ISS_com_ICMS'
    # veja, não é necessária mais uma classe. Imposto recebe outro em seu construtor
    ISS_com_ICMS = ISS(ICMS())
    calculador_de_impostos.realiza_calculo(orcamento, ISS_com_ICMS) # imprime 80

    print 'ICPP e IKCV'
    calculador_de_impostos.realiza_calculo(orcamento, ICPP()) # imprime 25.0
    calculador_de_impostos.realiza_calculo(orcamento, IKCV()) # imprime 30.0

    print 'ICPP_com_IKCV'
    # veja, não é necessária mais uma classe. Imposto recebe outro em seu construtor
```

```
ICPP_com_IKCV = ICPP(IKVC())
calculador_de_impostos.realiza_calculo(orcamento, ICPP_com_IKVC) # imprime 55.0
```

Quando compomos comportamento, através de classes que recebem objetos do mesmo tipo que elas mesmas (nesse caso, `ISS` que é um `Imposto`, recebe em seu construtor outro `Imposto`) para fazerem parte de seu comportamento, de uma maneira que seu uso é definido a partir do que se passou para a instanciação dos objetos, é o que caracteriza o Design Pattern chamado *Decorator*.

Decorator na linguagem Python

O Python permite a decoração de funções e métodos também utilizando um recurso de linguagem . Será que ele tem vantagem sobre o pattern que acabamos de aplicador? Vamos dar uma olhada.

Queremos compor nosso ISS com o novo imposto IPVX. No lugar de criarmos IPVX como uma classe, vamos criá-lo como uma função, no início do arquivo `impostos.py`:

```
def IPVX():
   pass
   # chama o cálculo do imposto do ISS, pega o resultado e soma com R$ 50,00
```

Nosso decorator não está pronto ainda, mas tenha em mente que aplicados o decorator prefixando- o com `@` e colocando imediatamente acima de métodos ou funções que desejamos decorar. Como queremos decorar o cálculo do ISS, vamos adicioná-lo ao método que calcula o imposto na classe `ISS`:

```
class ISS(Imposto):
    @IPVX
    def calcula(self, orcamento):
        return orcamento.valor * 0.06 + self.calculo_do_outro_imposto(orcamento)
```

Quando nosso decorator estiver pronto, toda vez que o ISS calcular o seu imposto, o valor retornado será acrescido de `R$ 50,00`, tarefa do nosso decorator. Nada mais justo do concluir sua implementação. O primeiro ponto, é que nosso decorator recebe como parâmetro a função o método na qual foi adicionado. Lembre-se que em Python isso é possível, uma vez que funções são cidadãos de primeira classe e podem ser armazenadas em variável e também passadas com parâmetro:

```
def IPVX(metodo_ou_funcao):
   pass
   # aplica imposto fixo, de R$ 50,00
```

Ainda não esta pronto. Nosso decorator precisa empacotar o `metodo_ou_funcao`. Vamos declará-la com o nome `wrapper` (empacotador):

```
def IPVX(metodo_ou_funcao):
    def wrapper(self, orcamento):
        # aplica o imposto
    return wrapper
```

Perceba que a função empacotadora recebe como parâmetro um `self` e também um orçamento. O `self` será o da classe `ISS` e o segundo parâmetro o orçamento que é passado para a função `calcula` da própria classe `ISS`.

Muito bem, agora estamos com a faca e o queijo na mão. Dentro da nossa função empacotadora, vamos chamar o método original passado como `metodo_ou_funcao` e executá-la, mas sem esquecer de passar o `self` recebido como parâmetro, caso contrário a função não executará no contexto da classe `ISS`:

```
def IPVX(metodo_ou_funcao):
    def wrapper(self, orcamento):
        return metodo_ou_funcao(self, orcamento)
    return wrapper
```

Do jeito que está, nosso decorator apenas chama o método de cálculo de imposto da classe `ISS` e não acrescenta nada. Continuamos com o mesmo comportamento. Agora, vamos somar com o resultado `R$ 50,0`:

```
def IPVX(metodo_ou_funcao):
    def wrapper(self, orcamento):
        return metodo_ou_funcao(self, orcamento) + 50.0
    return wrapper
```

Excelente! Agora, toda vez que o imposto ISS for calculado, além da sua lógica de cálculo de imposto, também será aplicado nosso decorator, modificando o resultado final. No final, nosso resultado no console será:

```
ISS e ICMS
80.0 # somou 30 calculado pelo ISS com 50 do nosso decorator que representa o IPVX
```

Podemos aplicar ainda mais de um decorator se assim desejarmos! Interessante não? O problema é que essa solução será sempre aplicada, pois o decorator estará sempre na classe `ISS`. Pode ser que isso seja interessante em alguns casos, porém, a solução que vimos anteriormente permite a composição de nossos impostos em tempo de execução. O que importa é o desenvolver saber quando usar um ou outro de acordo com o problema que deseja resolver.