02

A grande variedade de impostos e o padrão Strategy

Um fato que é conhecido sobre todo software é que ele, mais cedo ou mais tarde, vai mudar: novas funcionalidades aparecerão, outras deverão ser alteradas, etc. O problema é que, geralmente, essas mudanças não são feitas com muito planejamento. Para atingir o objetivo da melhora do 'Design' e diminuir o custo de manutenção, existem algumas técnicas que simplificam o código escrito.

Algumas das principais técnicas para atingir um bom 'Design' são tão comuns que foram catalogadas em um conjunto de alternativas para solucionar problemas de Design de código, chamados de Design Patterns, mais conhecido em português como Padrões de Projetos, os quais, durante esse curso, aprenderemos a utilizar. Um padrão de projeto nada mais é do que uma solução elegante para um problema que é muito recorrente em nosso dia-a-dia.

Preparação do ambiente

Antes de continuar, veja se você tem o ambiente mínimo para dar continuidade ao treinamento. As instruções estão no primeiro exercício (https://cursos.alura.com.br/course/design-patterns-python/section/1/task/3) deste capítulo.

Muitas regras e código complexo

Vamos iniciar nosso treinamento tomando como exemplo uma aplicação cujo objetivo é a criação de orçamentos, temos uma regra de negócio na qual os valores dos orçamentos podem ser submetidos a alguns impostos, como ISS, ICMS e assim por diante, inclusive um orçamento não pode ser alterado depois de criado. Com isso, temos a simples classe que representa o orçamento, recebendo via construtor o seu valor:

```
# -*- coding: UTF-8 -*-
# orcamento.py
class Orcamento(object):
  def __init__(self, valor):
    self.__valor = valor

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

Criamos o atributo **valor** como *property* , mas com acesso de apenas leitura. Isso garante que o valor recebido pelo construtor não seja alterado depois do orçamento criado. Com certeza uma grande vantagem do paradigma da orientação à objetos, pois estamos encapsulando o valor do orçamento, permitindo acesso apenas à leitura.

Se quiséssemos dar acesso de escrito, precisaríamos explicitar isso usando, além do `@property` o `@valor.setter` em outro método da classe. Como só faz sentido receber o valor através do construtor, não criamos um método com o decorator `@valor.setter` .

Com isso, podemos criar novos orçamentos, instanciando objetos do respectivo tipo e caso queiramos calcular um imposto sobre seu valor, basta utilizarmos a `property valor` para isso. Assim, podemos estipular que o ICMS valha 10% e precisamos calculá-lo, baseado no valor do orçamento. Para isso, podemos ter a seguinte classe com um simples método para realizar o cálculo. No lugar de criar um arquivo para testar nossa classe em separado, vamos adicionar um código especial que será executado sempre que executarmos o arquivo pelo terminal, e nunca quando ele for importado:

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

# calculador_de_impostos.py

class Calculador_de_impostos(object):

  def realiza_calculo(self, orcamento):

    icms_calculado = orcamento.valor * 0.1;
    print icms_calculado

if __name__ == '__main__':

    from orcamento import Orcamento

    orcamento = Orcamento(500.0)
    calculador_de_impostos = Calculador_de_impostos()
    calculador_de_impostos.realiza_calculo(orcamento) ## imprimirá 50.0
```

Quando executamos o código pelo terminal, a variável implícita `__name__` contém o valor `main` . Isso permite que dentro do próprio arquivo da classe seja possível adicionar um código que a teste. Ganharemos tempo com esse recurso, evitando a criação de outros arquivos.

```
python calculador_de_impostos.py
```

Podemos ainda querer calcular outro imposto, como o ISS, que é 6% do valor do orçamento. Com isso, adicionamos a nova regra ao código anterior. Mas devemos escolher qual o imposto que será calculado. Portanto, o método `realizaCalculo` deverá receber uma informação, indicando qual o imposto terá o cálculo realizado:

Testando:

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

# calculador_de_impostos.py
class Calculador_de_impostos(object):

  def realiza_calculo(self, orcamento, imposto):

    if 'ICMS' == imposto:
      icms_calculado = orcamento.valor * 0.1
      print icms_calculado

    elif 'ISS' == imposto:
      iss_calculado = orcamento.valor * 0.06
      print iss_calculado

if __name__ == '__main__':

    from orcamento import Orcamento

    orcamento = Orcamento(500.0)
    calculador_de_impostos = Calculador_de_impostos()
```

```
calculador_de_impostos.realiza_calculo(orcamento, 'ICMS') # imprimirá 50
calculador_de_impostos.realiza_calculo(orcamento, 'ISS') # imprimirá 30
```

Note que uma das consequências do código que acabamos de criar é que espalhamos os cálculos e nossas regras de negócio. Dessa maneira, não temos nenhum encapsulamento de nossas regras de negócio e elas se tornam bastante suscetíveis a serem replicadas em outros pontos do código da aplicação. Que tal criarmos essas regras como funções dentro de um módulo do Python e que podem ser importadas sempre que necessário? Poderíamos até usar classes, mas neste momento funções são suficientes.

Encapsulando o comportamento

Ao invés de mantermos as regras espalhadas pela nossa aplicação, podemos encapsulá-las em funções cujas responsabilidades sejam realizar os cálculos. Isso é possível, porque a linguagem Python é multiparadigma, isto é, trabalha com o paradigma orientado a objetos, funcional, inclusive procedural, se assim desejarmos. Para isso, podemos criar as funções ICMS e ISS dentro de um módulo específico, que chamaremos de `impostos` :

```
# -*- coding: UTF-8 -*-
# impostos.py
def calcula_ICMS(orcamento):
    return orcamento.valor * 0.1

def calcula_ISS(orcamento):
    return orcamento.valor * 0.06
```

Agora temos as duas funções que separam a responsabilidade dos cálculos de impostos, com isso, podemos utilizá-las na classe `Calculador_de_impostos` , da seguinte maneira:

```
# -*- coding: UTF-8 -*-
# calculador_de_impostos.py

from impostos import calcula_ISS, calcula_ICMS

class Calculador_de_impostos(object):

  def realiza_calculo(self, orcamento, imposto):

    if 'ICMS' == imposto:
      icms_calculado = calcula_ICMS(orcamento)
      print icms_calculado
    elif 'ISS' == imposto:
      iss_calculado = calcula_ISS(orcamento)
      print iss_calculado

if __name__ == '__main__':

    from orcamento import Orcamento

    orcamento = Orcamento(500.0)
    calculador_de_impostos = Calculador_de_impostos()
    calculador_de_impostos.realiza_calculo(orcamento, 'ICMS') # imprimie 50.0
    calculador_de_impostos.realiza_calculo(orcamento, 'ISS') # imprime 30.0
```

Agora o código está melhor, mas não significa que esteja bom. Um ponto **extremamente crítico** desse código é o fato de que quando quisermos adicionar mais um tipo diferente de cálculo de imposto em nosso calculador, teremos que alterar essa classe adicionando mais um bloco de `if`, além de criarmos uma nova função no módulo `imposto.py`, que encapsulará o cálculo do novo imposto. Parece bastante trabalho.

Eliminando os condicionais com funções de primeira classe e o pattern Strategy

O que queremos em nosso código é não realizar nenhum condicional, ou seja, não termos mais que fazer vários `if`s dentro do `Calculador_de_Impostos`. Dessa forma, não devemos mais receber a `String` com o nome do imposto, no qual realizamos as cláusulas `if`. Mas como escolheremos qual o imposto que deve ser calculado?

Uma primeira possibilidade é criar dois métodos (usaremos o termo `método` quando falamos de classe e o termo `função` quando temos um bloco de código que não pertence a uma classe) separados na classe `Calculador_de_Impostos`. Um para o ICMS e outro para o ISS, dessa forma teremos:

```
from impostos import calcula_ISS, calcula_ICMS

class Calculador_de_impostos(object):

  def realiza_calculo_ICMS(self, orcamento):
    icms_calculado = calcula_ICMS(orcamento)
    print icms_calculado

  def realiza_calculo_ISS(self, orcamento):
    iss_calculado = calcula_ISS(orcamento)
    print iss_calculado

if __name__ == '__main__':

    from orcamento import Orcamento

    orcamento = Orcamento(500.0)
    calculador_de_impostos = Calculador_de_impostos()
    calculador_de_impostos.realiza_calculo_ICMS(orcamento) # imprime 50.0
    calculador_de_impostos.realiza_calculo_ISS(orcamento) # imprime 30.0
```

No entanto, agora só transferimos o problema dos vários `if`s para vários métodos. O que não resolve o problema. O próximo passo para conseguirmos melhorar essa solução é termos um único método, genérico, que consegue realizar o cálculo para **qualquer imposto**, sem fazer nenhum `if` dentro dele:

```
# -*- coding: UTF-8 -*-
# calculador_de_impostos.py
from impostos import calcula_ICMS

class Calculador_de_impostos(object):

  def realiza_calculo(self, orcamento):
    icms_calculado = calcula_ICMS(orcamento)

    # mas e se quisermos outro imposto?
    print icms_calculado
```

E se quisermos outro imposto?

Agora estamos presos ao ICMS. Precisamos que nosso código fique flexível o bastante para utilizarmos diferentes impostos na realização do cálculo. Uma possibilidade para resolvermos esse problema é, ao invés de importarmos a função de cálculo de imposto que desejamos dentro do método, recebermos a função do **Imposto** que queremos utilizar, como no código seguinte:

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

    def realiza_calculo(self, orcamento, calcula_imposto):
        valor = calcula_imposto(orcamento)
        print valor
```

E agora o nosso `Calculador_de_Impostos` está pronto para ser utilizado e flexível o bastante para receber diferentes tipos (ou "estratégias") de impostos. Um código que demonstra essa flexibilidade é o seguinte:

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

    def realiza_calculo(self, orcamento, calcula_imposto):
        valor = calcula_imposto(orcamento)
        print valor

if __name__ == '__main__':

    from orcamento import Orcamento
    from impostos import calcula_ISS, calcula_ICMS

    orcamento = Orcamento(500.0)
    calculador_de_impostos = Calculador_de_impostos()
    calculador_de_impostos.realiza_calculo(orcamento, calcula_ISS)
    calculador_de_impostos.realiza_calculo(orcamento, calcula_ICMS)
```

Agora, com um único método em nosso `Calculador_de_Impostos`, podemos realizar o cálculo de diferentes tipos de impostos, apenas recebendo a estratégia do tipo do imposto que desejamos utilizar no cálculo. Isso é possível, porque funções podem ser passadas como parâmetro para outras funções e métodos. Neste caso, quando passamos `calcula_ISS`, o parâmetro `calcula_imposto` de `realiza_calculo` será esta função. A mesma coisa quando passarmos `calcula_ICMS`.

Repare que a criação de uma nova estratégia de cálculo de imposto não implica em mudanças no código escrito acima! Basta criarmos uma nova função de cálculo de imposto, que nosso `Calculador_de_impostos` conseguirá calculá-lo sem precisar de nenhuma alteração!

Há outra forma?

Tudo bem, terminamos, mas se você segue à risca o paradigma da orientação a objetos pode ser que a declaração dos cálculos de impostos como funções não seja algo atraente e criá-las como classes fosse mais condizente com o que você está acostumado, mesmo que essa classe tenha um método apenas e não guarde estado. Vamos transformá-las em classes, mas ainda tendo em mente a aplicação do padrão `Strategy`. Só não podemos nos esquecer de adicionar o `self` como primeiro parâmetro dos métodos:

```
# -*- coding: UTF-8 -*-
class ICMS(object):
    def calcula_ICMS(self, orcamento):
        return orcamento.valor * 0.1

class ISS(object):
    def calcula_ISS(self, orcamento):
        return orcamento.valor * 0.06
```

Agora temos duas classes: ICMS e ISS, cada uma com um método responsável pelo cálculo do imposto. Será que precisamos alterar a classe `Calcula_Impostos` ? Não, a diferença mora apenas na maneira que passaremos a estratégia como parâmetro: primeiro instanciamos a classe e depois passamos seu método como parâmetro, que nada mais é que uma função que pertence à uma classe:

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

    def realiza_calculo(self, orcamento, calcula_imposto):
        valor = calcula_imposto(orcamento)
        print valor

if __name__ == '__main__':

    from orcamento import Orcamento
    from impostos import ISS, ICMS

    orcamento = Orcamento(500.0)
    calculador_de_impostos = Calculador_de_impostos()
    calculador_de_impostos.realiza_calculo(orcamento, ISS().calcula_ISS)
    calculador_de_impostos.realiza_calculo(orcamento, ICMS().calcula_ICMS)
```

Excelente, nosso código funciona, porém tornamos nosso código um pouco mais complexo, sem qualquer ganho substancial. Como não? Criamos duas classes! Será isso realmente um ganho? Será que esse paradigma da orientação à objetos pode contribuir com algo mais?

Não é incomum no mundo O.O. passarmos instância de classes como parâmetro de métodos, e que tal fazermos a mesma coisa? Se fizermos isso, não será possível passar qualquer função, mas um objeto. No lugar de passarmos uma função como parâmetro para `realiza_calculo` , passaremos instâncias de `ISS` e `ICMS` diretamente:

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

    def realiza_calculo(self, orcamento, imposto):
        valor = imposto.calcula_ISS(orcamento)
        # não funciona se for ICMS
        print valor

if __name__ == '__main__':

    from orcamento import Orcamento
    from impostos import ISS, ICMS

    orcamento = Orcamento(500.0)
```

```
    calculador_de_impostos = Calculador_de_impostos()
    calculador_de_impostos.realiza_calculo(orcamento, ISS())
    calculador_de_impostos.realiza_calculo(orcamento, ICMS())
```

Legal, restringimos o parâmetro da função `realiza_calculo`, que recebe agora um objeto e não uma função diretamente. É claro que uma função ainda pode ser passada devido à natureza dinâmica do Python, mas nosso código não funcionaria. O problema é que `realiza_calculo` não sabe quando chamar `imposto.cacula_ISS` ou `imposto_calcula_ICMS` e caímos no mesmo problema do início do capítulo. Teríamos que adicionar uma série de `if`s. Porém, nem tudo está perdido.

Duck Typing

Podemos convencionar um contrato para todo imposto, isto é, quem for um imposto deve ter o método `calcula`. Isso funciona, mas ainda teremos problema, pois não saberemos qual método chamar dentro de `realiza_calculo`, em `Calcula_Impostos`. Há solução? Sim, podemos convencionar uma interface de uso semelhante para as duas classes, isto é, ambas terão um mesmo nome de método, agora `calcula`:

```
# -*- coding: UTF-8 -*-
class ICMS(object):
    def calcula(self, orcamento):
        return orcamento.valor * 0.1

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

E agora, dentro de `realiza_calculo`, esperamos que o objeto passado como parâmetro contenha esse método:

```
# -*- 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
    from impostos import ISS, ICMS

    orcamento = Orcamento(500.0)
    calculador_de_impostos = Calculador_de_impostos()
    calculador_de_impostos.realiza_calculo(orcamento, ISS())
    calculador_de_impostos.realiza_calculo(orcamento, ICMS())
```

Nosso código funciona, contanto que a instância do objeto passado, seja lá qual for, contenha o método `calcula`. Aqui usamos o `Duck Typing`, isto é, não importa qual seja o objeto recebido como parâmetro, basta que ele tenha os métodos que desejamos chamar. É claro que ainda podemos passar uma instância de um objeto que contenha o método `calcula`, por exemplo, a instância de um objeto `Quadrado`, que sabe calcular sua área e com certeza o resultado do nosso código não seria o esperado, porém conseguimos restringir ainda mais as possibilidades dos parâmetros recebidos: precisa ser uma instância de uma classe e ainda ter o método `calcula`.

Das duas abordagens que vimos qual é a melhor? Depende, a primeira é mais flexível e não necessita a criação de classes, deixando nosso código menos verboso, porém essa flexibilidade pode nos causar problemas se não tivermos cuidado. A segundo forma é mais verbosa e um pouco menos flexível, porém ajuda a restringir um pouco quais são os parâmetros aceitos pela classe.

Um pouco mais sobre multiparadigma e design patterns

Segundo Thomas Kuhn, em seu livro *Estrutura das revoluções Científicas*, é por meio dos paradigmas que os cientistas buscam respostas para os problemas colocados em prática pelas ciências. Programar não deixa de ser uma ciência, não é à toa que temos a Ciência da Computação.

Vimos que o Python suporta os paradigmas *orientado a objetos*, *funcional* e *procedural* e cada um desses paradigmas nos ajudam a criar **perguntas** e a nos fornecer **respostas** sobre dado problema. Porém, muitas vezes quem vive um paradigma, ignora completamente o outro ou tem dificuldade de entendê-lo (aconteceu isso com você?). É aí que mora o maior desafio de uma linguagem multiparadigma: saber transitar entre diferentes paradigmas para solucionar problemas.

Dentro do contexto dos paradigmas, podemos dizer que os design patterns podem existir ou não, dependendo da linguagem utilizada e não é raro a própria linguagem em si já oferecer uma solução para o problema.