02

Códigos parecidos e o Template Method

Começando deste ponto? Você pode fazer o DOWNLOAD (https://s3.amazonaws.com/caelum-online-public/python-dp1/stages/03-design-patterns.zip) completo do projeto do capítulo anterior e continuar seus estudos a partir deste capítulo.

Hum, código parecido!

Em nossa aplicação de orçamentos, diversos impostos possuem cálculos parecidos. Por exemplo, temos dois impostos hipotéticos: ICPP e IKCV. O imposto ICPP é calculado da seguinte forma: caso o valor do orçamento seja menor que R$ 500,00, deve-se cobrar 5%; caso contrário, 7%.

Vamos adicionar o ICPP no já existente `impostos.py`, respeitando a convenção que todo imposto deve ter o método `calcula`, que recebe um orçamento:

```
# # -*- coding: UTF-8 -*-
# impostos.py
# adicionado no início do arquivo
class ICPP(object):

    def calcula(self, orcamento):
        if orcamento.valor > 500:
            return orcamento.valor * 0.07
        else:
            return orcamento.valor * 0.05

# outros impostos omitidos
```

Já o imposto IKCV, caso o valor do orçamento seja maior que R$ 500,00 e algum item tiver valor superior a R$ 100,00, o imposto a ser cobrado é de 10%; caso contrário, 6%.

Também vamos adicioná-lo em `impostos.py`, como primeiro imposto:

```
# # -*- coding: UTF-8 -*-
# impostos.py
# adicionado no início do arquivo
class IKCV(object):

    def calcula(self, orcamento):
        if orcamento.valor > 500 and self.__tem_item_maior_que_100_reais(orcamento):
            return orcamento.valor * 0.10
        else:
            return orcamento.valor * 0.06

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

# outros impostos omitidos
```

Agora, nada mais justo do que rodar um teste para sabermos se tudo está funcionando. Precisamos modificar ligeiramente `calculador_de_impostos.py` :

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

    # adicionado os impostos no import
    from impostos import ISS, ICMS, ICPP, IKCV

    orcamento = Orcamento()
    # adicionando itens ao orçamento
    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())

    # cálculo dos novos impostos
    print 'ICPP e IKCV'
    calculador_de_impostos.realiza_calculo(orcamento, ICPP()) # imprime 25.0
    calculador_de_impostos.realiza_calculo(orcamento, IKCV()) # imprime 30.0
```

Veja que os dois impostos, apesar de diferentes, possuem uma estrutura em comum. Ambos checam o orçamento para ver se devem cobrar a taxação máxima e, a partir daí, cobram a máxima ou a mínima.

Generalizando a estrutura do nosso código

Poderíamos escrever um algoritmo que generaliza os outros dois, algo como um "molde":

```
# # -*- coding: UTF-8 -*-
# impostos.py
# adicionado no início do arquivo
class Template_de_imposto_condicional(object):

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

    # é claro, precisamos implementar os três métodos necessário
```

Bastaria agora fazer com que os impostos ICPP e IKCV possuam suas próprias implementações de `deve_usar_maxima_taxacao`, `maxima_taxacao` e `minima_taxacao`.

Python e classes abstratas

Podemos deixar explícito nesse código que cada um desses métodos são "buracos" e **devem** ser implementados por classes-filhas. **Logo, podemos tornar esses métodos abstratos!**

```
# # -*- coding: UTF-8 -*-
# impostos.py
# adicionado no início do arquivo
from abc import ABCMeta, abstractmethod

class Template_de_imposto_condicional(object):

    __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
```

Agora, vamos fazer com que `ICPP` e `IKCV` herdem do nosso template:

```
# código anterior omitido
class ICPP(Template_de_imposto_condicional):

    def calcula(self, orcamento):
        if orcamento.valor > 500:
            return orcamento.valor * 0.07
        else:
            return orcamento.valor * 0.05

class IKCV(Template_de_imposto_condicional):

    def calcula(self, orcamento):
        if orcamento.valor > 500 and self.__tem_item_maior_que_100_reais(orcamento):
            return orcamento.valor * 0.10
        else:
            return orcamento.valor * 0.06

    def __tem_item_maior_que_100_reais(orcamento):
```

```
            for item in orcamento.obter_itens():
                if item.valor > 100:
                    return True
                return False
    # código posterior omitido
```

Implementando métodos abstratos

Se rodarmos `calculador_de_impostos.py`, receberemos a seguinte mensagem de erro:

```
TypeError: Can't instantiate abstract class ICPP with abstract methods deve_usar_maxima_taxacao, ma
```

Precisamos implementar esses métodos abstratos nas classes filhas:

```
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
```

Veja que ambos as classes de impostos só implementam as partes "que faltam" do algoritmo! A classe `Template_de_imposto_condicional` possui um método que funciona como um template, ou seja, um molde, para as classes filhas. Daí o nome do padrão de projeto: **Template Method**.

Veja que o uso do padrão evitou a repetição de código, e ainda facilitou a implementação das diferentes variações do algoritmo.