02

Criação de Objetos e Builder

Começando deste ponto? Você pode fazer o DOWNLOAD (https://s3.amazonaws.com/caelum-online-public/python-dp1/stages/06-design-patterns.zip) completo do projeto do capítulo anterior e continuar seus estudos a partir deste capítulo.

Uma nota fiscal, diferentes maneiras de constuir

O mundo real é complexo. E essa complexidade é geralmente refletida nas classes do nosso sistema. Imagine uma Nota Fiscal: ela é composta de razão social, CNPJ, data de emissão, ítens, detalhes e assim por diante.

Imagine essa classe, tendo atributos que representam cada uma das informações acima. Imagine também que essa classe receba todos esses atributos no construtor:

```
# -*- coding: UTF-8 -*-
# nota_fiscal.py
from datetime import date

class Item(object):

    def __init__(self, descricao, valor):
        self.__descricao = descricao
        self.__valor = valor

    @property
    def descricao(self):
        return self.__descricao

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

class Nota_fiscal(object):

    def __init__(self, razao_social, cnpj, itens, data_de_emissao, detalhes):
        self.__razao_social = razao_social
        self.__cnpj = cnpj
        self.__data_de_emissao = data_de_emissao
        if len(detalhes) > 20:
            raise Exception('Detalhes da nota não pode ter mais do que 20 caracteres')
        self.__detalhes = detalhes
        self.__itens = itens

    @property
    def razao_social(self):
        return self.__razao_social

    @property
    def cnpj(self):
        return self.__cnpj

    @property
    def data_de_emissao(self):
```

```
            return self.__data_de_emissao

        @property
        def detalhes(self):
            return self.__detalhes
```

No construtor da nossa classe até verificamos se o texto de detalhes está no limite de caracteres aceitos. Perfeito!

Agora, vejamos um exemplo que instância nossa classe.

```
# nota_fiscal.py
# código das classes omitidos

if __name__ == '__main__':

    itens=[
        Item(
            'ITEM A',
            100
        ),
        Item(
            'ITEM B',
            200
        )
    ]

    nota_fiscal = Nota_fiscal(
        'FHSA Limitada',
        '012345678901234',
        itens,
        date.today(),
        ''
    )
```

Veja que nesse construtor nada nos impede de invertermos acidentalmente a ordem dos parâmetros, algo do tipo:

```
# trocou razao_social pelo CNPJ ( "foi sem querer querendo...")
nota_fiscal = Nota_fiscal(
    '012345678901234',
    'FHSA Limitada',
    itens,
    date.today(),
    ''
)
```

Pior, as vezes ainda temos o problema de parâmetros opcionais. No exemplo acima, o código é funcional mesmo com a troca de parâmetros. Mas se omitirmos um parâmetro, por exemplo, a `data_de_emissao` :

```
# trocou razao_social pelo CNPJ e ainda deixou de passar um parâmetro
nota_fiscal = Nota_fiscal(
    '012345678901234',
    'FHSA Limitada',
    itens,
```

```
        ''
    )
```

recebemos o erro:

```
TypeError: __init__() takes exactly 6 arguments
```

Não estranhe o `6 arguments`, mesmo nossa classe só recebendo cinco. Isso ocorre por causa do `self`, o primeiro parâmetro do construtor que é passado automaticamente pelo Python. Ok? (:

Parâmetros nomeados ao nosso socorro

O Python pode nos ajudar. O primeiro passo é passarmos os parâmetros do construtor da classe através de **parâmetros nomeados**, dessa maneira, não corremos o risco de trocarmos a ordem dos parâmetros, inclusive faremos isso com as instâncias da classe `Item`:

```
# nota_fiscal.py
# código das classes omitidos

if __name__ == '__main__':

    itens=[
        Item(
            descricao='ITEM A',
            valor=100
        ),
        Item(
            descricao='ITEM B',
            valor=200
        )
    ]

    nota_fiscal = Nota_fiscal(
        cnpj='012345678901234',
        razao_social='FHSA Limitada',
        itens=itens,
        detalhes='Referente à ...',
        data_de_emissao=date.today()
    )
```

Repare que a ordem de passagens dos parâmetros não importa mais, como é o caso da `data_de_emissao` que foi recebida como último parâmetro, posição diferente no construtor da classe.

Lidando com parâmetros opcionais

Muito bem, resolvemos o problema na passagem de parâmetros. Mas e os parâmetros opcionais? Não podemos simplesmente deixarmos de passar a observação, pois o Python lançará uma exceção dizendo que faltou a passagem de parâmetros. Será que tem solução? Claro que sim! Python aceita definir parâmetros opcionais, basta no próprio construtor passarmos um valor padrão para o atributo.

Mas atenção: todos os parâmetros dos construtores que são opcionais devem vir por último, caso contrário o interpretador do Python recusará a executar nosso código.

Alterando a classe `Nota_fiscal`:

```
# restante do código omitido
class Nota_fiscal(object):
    # os parâmetros opcionais devem ser os últimos
    def __init__(self, razao_social, cnpj, itens, data_de_emissao=date.today(), detalhes=''):
        # restante do código omitido
```

No exemplo acima, se não passarmos `data_de_emissao` e nem `detalhes` eles ganharam como valor padrão a data atual ou uma string vazia respectivamente, inclusive poderíamos até indicar que o parâmetro recebe `None`.

Agora, se omitirmos esses parâmetros, ainda assim, nosso código funcionará:

```
if __name__ == '__main__':

    itens=[
        Item(
            descricao='ITEM A',
            valor=100
        ),
        Item(
            descricao='ITEM B',
            valor=200
        )
    ]

    nota_fiscal = Nota_fiscal(
        cnpj='012345678901234',
        razao_social='FHSA Limitada',
        itens=itens
    )
```

Objetos complexos existem e vão continuar existindo. Caso ele seja complexo, o desenvolvedor deve pensar sempre em utilizar parâmetros nomeados, inclusive indicar quais parâmetros são opcionais no construtor da classe. Qualquer regra que valida esses parâmetros estará no domínio a qual pertence: `Nota_fiscal` e não em uma classe externa.

E o Builder?

Mas onde está o design pattern `Builder`, você deve estar se perguntado? Em nenhum lugar, porque o problema de construção do objeto apresentado foi resolvido utilizando recursos da linguagem Python. Apesar do nosso problema está resolvido, veremos sua solução utilizando o design pattern `Builder` comparando-o com a solução que já temos.

Vamos criar a classe `Criador_de_nota_fiscal` em um arquivo em separado. Essa classe terá como propriedades todas as de uma nota fiscal. Para cada uma destas propriedades, criaremos um método que recebe seu valor, atribui ao nosso Builder e por fim retorna o mesmo. Isso é importante, para podermos encadear as chamadas desses métodos. Por fim, teremos o método `constroi` que retorna uma nota fiscal. Aliás, ele se encarregará de verificar se todos os parâmetros obrigatórios foram passados:

```
# -*- coding: UTF-8 -*-
# criador_de_nota_fiscal.py
from nota_fiscal import Nota_fiscal, Item

class Criador_de_nota_fiscal(object):

    def __init__(self):

        self.__razao_social = None
        self.__cnpj = None
        self.__data_de_emissao = None
        self.__detalhes = ''
        self.__itens = None

    def com_razao_social(self, razao_social):
        self.__razao_social = razao_social
        return self

    def com_cnpj(self, cnpj):
        self.__cnpj = cnpj
        return self

    def com_data_de_emissao(self, data_de_emissao):
        self.__data_de_emissao = data_de_emissao
        return self

    def com_itens(self, itens):
        self.__itens = itens
        return self

    def constroi(self):
        if self.__razao_social is None:
            raise Exception('Razão social deve ser preenchida')
        if self.__cnpj is None:
            raise Exception('CNPJ deve ser preenchido')
        if self.__itens is None:
            raise Exception('Itens deve ser preenchido')

        return Nota_fiscal(
            razao_social=self.__razao_social,
            cnpj=self.__cnpj,
            itens=self.__itens,
            data_de_emissao=self.__data_de_emissao,
            detalhes = self.__detalhes
        )
```

Veja que o método `constroi` se preocupa em passar os parâmetro na ordem correta. Mas espere um pouco: veja que também estamos usando parâmetros nomeados e os valores padrões passados são os mesmos que o construtor da nossa classe passaria! É claro que a sintaxe das chamadas encadeadas do Builder são elegantes e indentificam quais são os parâmetros passados, porém o construtor da nossa classe também indica quais são esses parâmetros, apesar de um pouquinho menos elegante.

```
if __name__ == '__main__':
```

```
itens=[
    Item(
        descricao='ITEM A',
        valor=100
    ),
    Item(
        descricao='ITEM B',
        valor=200
    )
]


# usando nosso Builder.
nota_fiscal = (Criador_de_nota_fiscal()
    .com_razao_social('FHSA Limitada')
    .com_cnpj('012345678901234')
    .com_itens(itens)
    .constroi())

print nota_fiscal.razao_social
print nota_fiscal.cnpj
print nota_fiscal.detalhes
```

Mas nem tudo é oito ou oitenta: o padrão Builder permite uma criação de objeto mais refinada se assim desejarmos. Podemos condicionar a chamada do método `com_cnpj` apenas se o método `com_razao_social` for invocado primeiro e coisas do tipo.

Por fim, o que queremos incitar em você é considerar a aplicação de um design pattern ou não de acordo com o problema em questão. Inclusive levando em consideração os recursos que a linguagem oferece.