02

Estados que variam e o State

Começando deste ponto? Você pode fazer o DOWNLOAD (https://s3.amazonaws.com/caelum-online-public/python-dp1/stages/05-design-patterns.zip) completo do projeto do capítulo anterior e continuar seus estudos a partir deste capítulo.

Os diferentes estados de um orçamento

Nossos orçamentos podem ter diferentes estados durante o seu ciclo de vida. Um orçamento nasce "Em aprovação" e pode virar "Aprovado" ou "Reprovado". Ao final de todo o processo, deverá ser "Finalizado".

Dependendo do estado que o orçamento se encontra, algumas ações podem ser diferentes. Por exemplo, podemos adicionar um desconto extra ao orçamento: quando o orçamento está em aprovação, a empresa oferece 5% a mais de desconto; quando já está aprovado, a empresa oferece 2% de desconto. Orçamentos reprovados e finalizados não recebem nada de desconto extra.

O problema da solução procedural

Em implementações mais procedurais, desenvolvedores geralmente optam por representar o estado do objeto por meio de constantes. Por exemplo:

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

    EM_APROVACAO = 1
    APROVADO = 2
    REPROVADO = 3
    FINALIZADO = 4

    def __init__(self):
        self.__itens = []
        self.estado_atual = 1
        self.__desconto_extra = 0.0

     # código posterior omitido
```

E, no método que aplica o desconto, é comum a utilização de vários `ifs` para tal:

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

    EM_APROVACAO = 1
    APROVADO = 2
    REPROVADO = 3
    FINALIZADO = 4

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

```
        self.__itens = []
        self.estado_atual = 1
        self.__desconto_extra = 0.0

    def aplica_desconto_extra(self):

        if self.estado_atual == Orcamento.EM_APROVACAO:
            self.__desconto_extra+= self.valor * 0.05
        elif self.estado_atual == Orcamento.APROVADO:
            self.__desconto_extra+= self.valor * 0.02
        elif self.estado_atual == Orcamento.REPROVADO:
            raise Exception('Orçamentos reprovados não recebem desconto extra')
        elif self.estado_atual == Orcamento.FINALIZADO:
            raise Exception('Orcamentos finalizados não recebem desconto extra')

    # 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 - self.__desconto_extra # valor agora leva em consideração o desconto aplicado
```

Precisamos testar, vamos adicionar o código de teste diretamente na classe `Orcamento`:

```
# -*- coding: UTF-8 -*-
# Orcamento.py
# restante da classe Orcamento e classe Item omitidos

if __name__ == '__main__':

    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))

    print 'Valor sem desconto extra %s' % (orcamento.valor)
    orcamento.aplica_desconto_extra()
    print 'Valor com desconto extra (em aprovação) %s' % (orcamento.valor)

    orcamento.estado_atual = Orcamento.APROVADO
    orcamento.aplica_desconto_extra()
    print 'Valor com desconto extra (aprovado) %s' % (orcamento.valor)
```

Esse código tende a se tornar difícil de manter por vários motivos:

1) Geralmente outros comportamentos dessa classe também variam de acordo com o estado do objeto;

2) A cada novo estado, um novo `if` deve ser acrescentado em todos os métodos do objeto.

Cada estado, uma classe!

Para eliminarmos esses vários `ifs`, precisamos primeiro separar cada ação de acordo com o estado em uma classe diferente. Por exemplo:

```
# ainda não entra no código, só ilustrativo por enquanto

 class Em_aprovacao(object):
   # ...
 }
 class Aprovado(object):
   # ...
 }
 class Reprovado(object): {
   # ...
 }
 class Finalizado(object): {
   # ...
 }
```

Todas elas são possíveis estados de um orçamento, o que nos leva a crer que poderíamos ter uma método comum para todas elas. Todos esses estados devem também dar o desconto extra para o orçamento, de acordo com sua regra de negócio.

Vamos criar uma classe abstrata que será herdada por todos nossos estados. Pode ser no topo do arquivo `Orcamento.py`:

```
# -*- coding: UTF-8 -*-
# Orcamento.py
# classe abstrata entra no início do arquivo Orcamento.py
from abc import ABCMeta, abstractmethod

class Estado_de_um_orcamento(object):
    __metaclass__ = ABCMeta

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

Veja que pela natureza dinâmica do Python, bastava todos os nossos estados terem um método com mesmo nome para o *Duck Typing* funcionar, mas quando criamos uma classe abstrata, obrigamos que classes filhas implementem o método `aplica_desconto_extra`. Fomos mais verbosos e rígidos aqui, mas ganhamos a garantia que o método será implementado.

Agora, vamos criar as classes que representam estados do Orcamento, também em `Orcamento.py`, abaixo da declaração da classe abstrata:

```
# -*- coding: UTF-8 -*-
# Orcamento.py
# início do arquivo

from abc import ABCMeta, abstractmethod

class Estado_de_um_orcamento(object):
    __metaclass__ = ABCMeta

    @abstractmethod
```

```
    def aplica_desconto_extra(self, orcamento):
        pass

class Em_aprovacao(Estado_de_um_orcamento):
    def aplica_desconto_extra(self, orcamento):
        orcamento.desconto_extra+= orcamento.valor * 0.05

class Aprovado(Estado_de_um_orcamento):
    def aplica_desconto_extra(self, orcamento):
        orcamento.desconto_extra+= orcamento.valor * 0.02

class Reprovado(Estado_de_um_orcamento):
    def aplica_desconto_extra(self, orcamento):
        raise Exception('Orçamentos reprovados não recebem desconto extra')

class Finalizado(Estado_de_um_orcamento):
    def aplica_desconto_extra(self, orcamento):
        raise Exception('Orcamentos finalizados não recebem desconto extra')

# código posterior omitido
```

Veja que ele dá um desconto de 2% já que ele representa o desconto que pode ser dado quando o objeto estiver no estado `APROVADO`.

Repare que todas as implementações recebem um `Orcamento` e precisam alterar `desconto_extra`, mas ele é privado em nosso orçamento. Vamos criar um método público que fará a adição do desconto que queremos aplicar de acordo com o estado, inclusive já vamos alterar nossas classes de estado para usarem esse método. Ah! Vamos aproveitar para remover o método `aplica_desconto_extra` antigo, que ainda está na classe `Orcamento`:

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

from abc import ABCMeta, abstractmethod

class Estado_de_um_orcamento(object):
    __metaclass__ = ABCMeta

    @abstractmethod
    def aplica_desconto_extra(orcamento):
        pass

class Em_aprovacao(Estado_de_um_orcamento):
    def aplica_desconto_extra(self, orcamento):
        orcamento.adiciona_desconto_extra(orcamento.valor * 0.05)

class Aprovado(Estado_de_um_orcamento):
    def aplica_desconto_extra(self, orcamento):
        orcamento.adiciona_desconto_extra(orcamento.valor * 0.02)

class Reprovado(Estado_de_um_orcamento):
    def aplica_desconto_extra(self, orcamento):
        raise Exception('Orçamentos reprovados não recebem desconto extra')

class Finalizado(Estado_de_um_orcamento):
    def aplica_desconto_extra(self, orcamento):
        raise Exception('Orcamentos finalizados não recebem desconto extra')
```

```
class Orcamento(object):

    EM_APROVACAO = 1
    APROVADO = 2
    REPROVADO = 3
    FINALIZADO = 4

    def __init__(self):
        self.__itens = []
        self.estado_atual = 1
        self.__desconto_extra = 0.0

    # método púbico que dá acesso ao desconto_extra

    def adiciona_desconto_extra(self, desconto):
        self.__desconto_extra+= desconto
```

Se tentarmos rodar `Orcamento.py` teremos um erro, porque nosso código de teste não encontra mais `aplica_desconto_extra` na classe Orcamento. Logo logo iremos resolver.

Usando classes para representar o estado interno

Precisamos agora fazer com que o `Orcamento` use essas classes para representar seu estado interno, e não mais as constantes. Além disso, precisamos fazer com que a classe que representa o estado do orçamento, como as classes `Aprovado`, `Reprovado` e `Em_aprovacao`, respondam pelo comportamento de `desconto_extra`:

```
# código anterior omitido
class Orcamento(object):

    # removida as constantes

    def __init__(self):
        self.__itens = []
        # começa com o estado em aprovação
        self.estado_atual = Em_aprovacao() # mudança aqui
        self.__desconto_extra = 0

    # nova implementação de aplica_desconto_extra
    def aplica_desconto_extra(self):
        self.estado_atual.aplica_desconto_extra(self)
```

Vamos alterar nosso teste um pouco:

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

    print 'Valor sem desconto extra %s' % (orcamento.valor)
    orcamento.aplica_desconto_extra()
    print 'Valor com desconto extra (em aprovação) %s' % (orcamento.valor)
```

Restringindo a mudança de estado

Funciona, mas podemos incrementar ainda mais nossa classe `Orcamento`, implementando a troca de estados. Por exemplo, se o orçamento está no estado `EM APROVAÇÃO`, ele pode ir apenas para os estados `APROVADO` e `REPROVADO`. Dos estados `APROVADO` e `REPROVADO`, podemos ir apenas para o estado `FINALIZADO`.

Representar isso em código procedural é difícil. Precisaríamos de várias condições (leia-se `ifs`), para alcançar o resultado esperado. O *State* nos ajuda nesse problema também. Basta representarmos as possíveis trocas em todas as classes que representam o estado. Para representar em todas as classes, precisamos alterar classe abstrata `Estado_de_um_orcamento`:

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

from abc import ABCMeta, abstractmethod

class Estado_de_um_orcamento(object):
    __metaclass__ = ABCMeta

    @abstractmethod
    def aplica_desconto_extra(self, orcamento):
        pass

    @abstractmethod
    def aprova(self, orcamento):
        pass

    @abstractmethod
    def reprova(self, orcamento):
        pass

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

Cada estado, por sua vez toma a decisão correta, e muda o estado do orçamento. Veja o estado `Em_aprovacao` abaixo. Observe o método `aprova`: do estado *EM APROVAÇÃO*, pode-se ir para o estado *APROVADO*. É isso que a classe implementa. Ela muda o estado do orçamento para *APROVADO*. Agora repare no método `finaliza()`. Não podemos ir para o estado *FINALIZADO* daqui, e por isso o método lança a exceção.

```
class Em_aprovacao(Estado_de_um_orcamento):
    def aplica_desconto_extra(self, orcamento):
        orcamento.adiciona_desconto_extra(orcamento.valor * 0.05)

    def aprova(self, orcamento):
        orcamento.estado_atual = Aprovado()

    def reprova(self, orcamento):
        orcamento.estado_atual = Reprovado()

    def finaliza(self, orcamento):
        raise Exception('Orcamento em aprovação não podem ir para finalizado diretamente')
```

Veja o estado `Aprovado`, por exemplo:

```
class Aprovado(Estado_de_um_orcamento):
    def aplica_desconto_extra(self, orcamento):
        orcamento.adiciona_desconto_extra(orcamento.valor * 0.02)

    def aprova(self, orcamento):
        raise Exception('Orçamento já está em estado de aprovação')

    def reprova(self, orcamento):
        raise Exception('Orçamento está em estado de aprovação e não pode ser reprovado')

    def finaliza(self, orcamento):
        orcamento.estado_atual = Finalizado()
```

Os demais estados:

```
class Reprovado(Estado_de_um_orcamento):
    def aplica_desconto_extra(self, orcamento):
        raise Exception('Orçamentos reprovados não recebem desconto extra')

    def aprova(self, orcamento):
        raise Exception('Orçamento reprovado não pode ser aprovado')

    def reprova(self, orcamento):
        raise Exception('Orçamento já está em estado de reprovação')

    def finaliza(self, orcamento):
        raise Exception('Orçamento reprovado não pode ser finalizado')

class Finalizado(Estado_de_um_orcamento):
    def aplica_desconto_extra(self, orcamento):
        raise Exception('Orcamentos finalizados não recebem desconto extra')

    def aprova(self, orcamento):
        raise Exception('Orçamento finalizado já foi aprovado')

    def reprova(self, orcamento):
        raise Exception('Orçamento já finalizado não pode ser reprovado')

    def finaliza(self, orcamento):
        raise Exception('Orçamento já foi finalizado')
```

O `Orcamento` por sua vez, sempre que recebe uma ação que depende do seu estado, repassa a chamada para o seu estado atual:

```
class Orcamento(object):

    # removida as constantes

    def __init__(self):
        self.__itens = []
        # começa com o estado em aprovação
```

```
            self.estado_atual = Em_aprovacao()
            self.__desconto_extra = 0

        def aprova(self):
            self.estado_atual.aprova(self)

        def reprova(self):
            self.estado_atual.reprova(self)

        def finaliza(self):
            self.estado_atual.finaliza(self)

    # código posterior omitido
```

Repare que eliminamos todos os `ifs` dessa classe e separamos as responsabilidades. Uma classe para cada possível estado do objeto. Além disso, repare que a criação de um novo "Estado" é fácil: basta criar uma nova classe que herde de `Estado_de_um_orcamento` e ela funcionará sem muito esforço!

Testando nosso código!

```
# Orcamento.py

if __name__ == '__main__':
    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))

    orcamento.aplica_desconto_extra()
    print orcamento.valor # imprime 522.5 porque descontou 5% de 550.0
    orcamento.aprova()

    orcamento.aplica_desconto_extra()
    print orcamento.valor # imprime 512.05 porque descontou 2% de 522.5
    orcamento.finaliza()

    orcamento.aplica_desconto_extra() # lança exceção, porque não pode aplica desconto em um orçamer
```

Essa é a grande graça da orientação a objetos! Classes pequenas e com responsabilidades bem definidas. E com a ajuda do polimorfismo, podemos juntar esses comportamentos e formar um sistema maior.