05

Colocando em prática

Hora de praticar ainda mais! Você já passou pelos padrões `Strategy`, `Chain of Responsibility` e `Decorator`. Cada um deles existem para resolver problemas já conhecidos. Inclusive você teve a oportunidade de combinar o paradigma orientado a objetos com o funcional para aplicação desses patterns, até viu algumas soluções procedurais e também como recursos nativos da linguagem Python podem ajudá-lo! Vamos revisitar o problema deste capitulo e deixá-lo no ponto para resolver o problema em questão aplicando o padrão de projeto `State`.

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. Vejamos uma solução procedural para o problema:

```
# -*- 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

    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
```

Você pode até alterar sua classe `Orcamento` para que aplique essa solução procedural, inclusive testando-o:

```
# -*- 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)
`
```

É uma solução totalmente funcional e rápida de ser feita, porém com o passar do tempo ela pode se tornar um pesadelo. Por quê? Se novos estados forem necessários, teremos que adicionar mais uma cláusula `if` no método `aplica_desconto` . E se a regra de um estado mudar? Teremos que abrir para alteração a classe `Orcamento` . Além disso, outros métodos do código podem depender do estado e teríamos que adicionar outras cláusulas `if` nesses métodos! Para complicar, normalmente um estado não pode ir direto para outro sem que haja uma regra. Por exemplo, um estado `Aprovado` só pode ir para `Finalizado` e um `Finalizado` jamais troca de estado!

Hora de lançar mão do padrão de projeto `State` para deixar nosso código mais fácil de manter. Fique à vontade para consultar o texto explicativo ou o vídeo do treinamento.

Responda

INSERIR CÓDIGO		FORMATAÇÃO