02

Diferentes ações com Command

Encapsulando ações

Temos um sistema que controla pedidos de uma empresa. O pedido, após ser feito pelo site, passa por diversas etapas de um workflow interno. Por exemplo, um pedido precisa ser processado,e depois de pago, os itens devem ser separados e seguir para a entrega ao cliente.

Vamos representar essa classe em Python:

```
# -*- coding: utf-8 -*-

from datetime import date

class Pedido(object):

    def __init__(self, cliente, valor):

        self.__cliente = cliente
        self.__valor = valor
        self.__status = 'NOVO'
        self.__data_finalizacao = None

    def paga(self):
        self.__status = 'PAGO'

    def finaliza(self):
        self.__data_finalizacao = date.today()
        self.__status = 'ENTREGUE'

    @property
    def cliente(self):
        return self.__cliente

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

    @property
    def status(self):
        return self.__status

    @property
    def data_finalizacao(self):
        return self.__data_finalizacao
```

Agora imagine que essa loja receba muitas requisições. Para tratá-las, é necessário fazer uso de algum tipo de fila, que executa diferentes comandos para diferentes pedidos.

Vamos implementar esse leitor da fila:

```
class Fila_de_trabalho(object):

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

Exemplos de Command

Mas o problema é que não adianta guardar somente o pedido, mas sim a ação que precisamos executar em cima dele. Algo como:

```
class Fila_de_trabalho(object):

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

Vamos então criar uma classe abstrata chamada `Comando`, que representará um Comando que deve ser executado:

```
from abc import ABCMeta, abstractmethod
class Comando(object):

    __metaclass__ = ABCMeta

    @abstractmethod
    def executa(self):
        pass
```

Agora vamos criar os comandos. Um deles finalizará o pedido, o outro deles marca o pedido como pago:

```
class Conclui_pedido(Comando):

    def __init__(self, pedido):
        self.__pedido = pedido

    def executa(self):
        self.__pedido.finaliza()

class Paga_pedido(Comando):

    def __init__(self, pedido):
        self.__pedido = pedido

    def executa(self):
        self.__pedido.paga()
```

Repare que cada comando recebe um Pedido, e já sabe exatamente qual método invocar.

Dessa forma, temos uma "fila de comandos" a ser executada, e podemos executar da maneira que acharmos melhor. Vamos melhorar um pouco a classe que cuida da fila, e dar métodos para adicionar e processar todos:

```
class Fila_de_trabalho(object):

    def __init__(self):
        self.__comandos = []

    def adiciona(self, comando):
        self.__comandos.append(comando)

    def processa(self):
        for comando in self.__comandos:
            comando.executa()
```

Agora, um simples programa para testar seu uso:

```
# pedido.py
# código anterior omitido
if __name__ == '__main__':

    pedido1 = Pedido('Flávio', 150)
    pedido2 = Pedido('Almeida', 250)

    fila_de_trabalho = Fila_de_trabalho()
    fila_de_trabalho.adiciona(Paga_pedido(pedido1))
    fila_de_trabalho.adiciona(Paga_pedido(pedido2))
    fila_de_trabalho.adiciona(Conclui_pedido(pedido1))

    fila_de_trabalho.processa()
```

Pronto. Agora temos uma fila, que executa comandos em cima dos nossos pedidos. E executá-los ficou fácil. Criar novos comandos também é fácil.

O nome desse padrão de projeto, que facilita a criação de comandos, chama-se **Command**. Usamos ele quando temos que separar os comandos que serão executados do objeto que ele pertence. Um bom exemplo disso é o uso de filas de trabalho.