Conhecimento Condensado da Comunidade

Aprendendo Padrões de Projeto em Python

Tire proveito da eficácia dos padrões de projeto (design patterns) com Python para resolver problemas do mundo real em arquitetura e design de software

Apresentação de Anand B. Pillai, membro do conselho da Python Software Foundation

[PACKT]*
PUBLISHING

novatec

Chetan Giridhar

// DADOS DE ODINRIGHT

Sobre a obra:

A presente obra é disponibilizada pela equipe ***eLivros*** e seus diversos parceiros, com o objetivo de oferecer conteúdo para uso parcial em pesquisas e estudos acadêmicos, bem como o simples teste da qualidade da obra, com o fim exclusivo de compra futura.

É expressamente proibida e totalmente repudíavel a venda, aluguel, ou quaisquer uso comercial do presente conteúdo.

Sobre nós:

O ***eLivros*** e seus parceiros disponibilizam conteúdo de dominio publico e propriedade intelectual de forma totalmente gratuita, por acreditar que o conhecimento e a educação devem ser acessíveis e livres a toda e qualquer pessoa. Você pode encontrar mais obras em nosso site: ***eLivros***.

Como posso contribuir?

Você pode ajudar contribuindo de várias maneiras, enviando livros para gente postar ***Envie um livro*** ;)

Ou ainda podendo ajudar financeiramente a pagar custo de servidores e obras que compramos para postar, ***faça uma doação aqui*** :)

"***Quando o mundo estiver unido na busca do conhecimento, e não mais lutando por dinheiro e poder, então nossa sociedade poderá enfim evoluir a um novo nível.***"

eLivros.love

Converted by convertEPub

Aprendendo Padrões de Projeto em Python

Tire proveito da eficácia dos padrões de projeto (design patterns) em Python para resolver problemas do mundo real em arquitetura e design de software

Chetan Giridhar

Packt>
Novatec

Copyright © Packt Publishing 2016. First published in the English language under the title "Learning Python Design Patterns – Second Edition" (9781785888038)

Copyright © Packt Publishing 2016. Publicação original em inglês intitulada "Learning Python Design Patterns – Second Edition" (9781785888038). Esta tradução é publicada e vendida com a permissão da Packt Publishing.

© Novatec Editora Ltda. [2016].

Todos os direitos reservados e protegidos pela Lei 9.610 de 19/02/1998. É proibida a reprodução desta obra, mesmo parcial, por qualquer processo, sem prévia autorização, por escrito, do autor e da Editora.

Editor: Rubens Prates
Tradução: Lúcia A. Kinoshita
Revisão gramatical: Marta Almeida de Sá
Editoração eletrônica: Carolina Kuwabata

ISBN do ebook: 978-65-86057-15-7

ISBN do impresso: 978-85-7522-523-3

Novatec Editora Ltda.
Rua Luís Antônio dos Santos 110
02460-000 – São Paulo, SP – Brasil
Tel.: +55 11 2959-6529
E-mail: novatec@novatec.com.br
Site: www.novatec.com.br
Twitter: twitter.com/novateceditora
Facebook: facebook.com/novatec
LinkedIn: linkedin.com/in/novatec

Sumário

Apresentação

Sobre o autor

Sobre o revisor

Prefácio

capítulo 1 ■ Introdução aos padrões de projeto

- Compreendendo a programação orientada a objetos
  - Objetos
  - Classes
  - Métodos
- Principais aspectos da programação orientada a objetos
  - Encapsulamento
  - Polimorfismo
  - Herança
  - Abstração
  - Composição
- Princípios do design orientado a objetos
  - Princípio do aberto/fechado
  - Princípio da inversão de controle
  - Princípio da segregação de interfaces
  - Princípio da responsabilidade única
  - Princípio da substituição
- Conceito de padrões de projeto
  - Vantagens dos padrões de projeto
  - Taxonomia dos padrões de projeto
  - Contexto – aplicabilidade dos padrões de projeto
- Padrões para linguagens dinâmicas

Classificando os padrões
Padrões de criação
Padrões estruturais
Padrões comportamentais
Resumo

capítulo 2 ■ Padrão de projeto Singleton

Compreendendo o padrão de projeto Singleton
Implementando um Singleton clássico em Python
Instanciação preguiçosa no padrão Singleton
Singletons no nível de módulo
Padrão Singleton Monostate
Singletons e metaclasses
Um cenário do mundo real – o padrão Singleton, parte 1
Um cenário do mundo real – o padrão Singleton, parte 2
Desvantagens do padrão Singleton
Resumo

capítulo 3 ■ Padrão Factory – construindo fábricas para criar objetos

Compreendendo o padrão Factory
Padrão Simple Factory
Padrão Factory Method
Implementando o Factory Method
Vantagens do padrão Factory Method
Padrão Abstract Factory
Implementando o padrão Abstract Factory
Comparação entre Factory Method e Abstract Factory
Resumo

capítulo 4 ■ Padrão Façade – sendo adaptável com o Façade

Compreendendo os padrões de projeto estruturais
Compreendendo o padrão de projeto Façade
Um diagrama de classes UML
Façade

Sistema
Cliente
Implementando o padrão Façade no mundo real
Princípio do conhecimento mínimo
Perguntas frequentes
Resumo

capítulo 5 ■ Padrão Proxy – controlando o acesso a objetos

Compreendendo o padrão de projeto Proxy
Um diagrama de classes UML para o padrão Proxy
Compreendendo os diferentes tipos de proxies
Proxy virtual
Proxy remoto
Proxy de proteção
Proxy inteligente
Padrão Proxy no mundo real
Vantagens do padrão Proxy
Comparação entre os padrões Façade e Proxy
Perguntas frequentes
Resumo

capítulo 6 ■ Padrão Observer – de olho nos objetos

Introdução aos padrões comportamentais
Compreendendo o padrão de projeto Observer
Um diagrama de classes UML para o padrão Observer
Padrão Observer no mundo real
Modelos do padrão Observer
Modelo pull
Modelo push
Baixo acoplamento e o padrão Observer
Padrão Observer – vantagens e desvantagens
Perguntas frequentes
Resumo

capítulo 7 ■ Padrão Command – encapsulando chamadas
Introdução ao padrão Command
Compreendendo o padrão de projeto Command
  Um diagrama de classes UML para o padrão Command
Implementando o padrão Command no mundo real
  Considerações de design
Vantagens e desvantagens dos padrões Command
Perguntas frequentes
Resumo

capítulo 8 ■ Padrão Template Method – encapsulando algoritmos
Definindo o padrão Template Method
  Compreendendo o padrão de projeto Template Method
  Um diagrama de classes UML para o padrão Template Method
Padrão Template Method no mundo real
Padrão Template Method – hooks
Princípio de Hollywood e o Template Method
As vantagens e as desvantagens do padrão Template Method
Perguntas frequentes
Resumo

capítulo 9 ■ Modelo-Visão-Controlador – padrões compostos
Uma introdução aos padrões compostos
Padrão Modelo-Visão-Controlador
  Modelo – conhecimento da aplicação
  Visão – a aparência
  Controlador – a cola
Um diagrama de classes UML para o padrão de projeto MVC
Padrão MVC no mundo real
  Módulos
Vantagens do padrão MVC
Perguntas frequentes

Resumo

capítulo 10 ■ Padrão de projeto State
Definindo o padrão de projeto State
Compreendendo o padrão de projeto State
Compreendendo o padrão de projeto State com um diagrama UML
Um exemplo simples do padrão de projeto State
O padrão de projeto State com implementação em Python v3.5
Vantagens e desvantagens do padrão State
Resumo

capítulo 11 ■ Antipadrões
Uma introdução aos antipadrões
Antipadrões no desenvolvimento de software
Código espaguete
Martelo de Ouro
Fluxo de Lava
Programação do tipo copiar e colar ou cortar e colar
Antipadrões na arquitetura de software
Reinventando a roda
Vendor lock-in
Design by Committee
Resumo

Apresentação

> *"Controlar a complexidade é a essência da programação de computadores."*
>
> *– Brian Kernighan*

> *"Todos os problemas em ciência da computação podem ser resolvidos com outro nível de 'indireção' (abstração)."*
>
> *– David Wheeler*

As duas citações anteriores de dois cientistas da computação muito conhecidos ilustram o problema encarado pelo designer de softwares modernos – criar uma solução boa, estável, reutilizável e flexível para um problema de design de software.

Os padrões de projeto (design patterns) resolvem os problemas anteriores da forma mais elegante possível. Esses padrões fazem a abstração e sintetizam a experiência adquirida por muitos designers e arquitetos de software ao longo de vários anos resolvendo problemas semelhantes, apresentando-a na forma de componentes e interfaces organizados e bem definidos. São soluções que resistiram ao teste do tempo no que se refere à reusabilidade, flexibilidade, escalabilidade e manutenibilidade.

Muitos livros sobre padrões de projeto já foram escritos, e o famoso livro da GoF (Gang of Four, ou Gangue dos Quatro) constitui a pedra angular de quase todo o domínio.

No entanto, na era da web e da computação móvel, em que os programas tendem a ser escritos em linguagens de alto nível como Python, Ruby e Clojure, com frequência, precisamos de livros que traduzam a linguagem usada, muitas vezes obscura, para termos mais conhecidos, com código reutilizável escrito nessas linguagens de programação mais novas e dinâmicas. Isso é sobretudo válido para programadores

iniciantes que, em geral, tendem a ficar perdidos com as complexidades do design contrapostas à implementação e que às vezes precisam da ajuda de um especialista.

Este livro desempenha muito bem esse papel. Foram utilizados os templates de padrões de projeto conforme apresentados no livro da GoF e acrescentaram-se outros para complementar – no entanto, antes de mergulhar nos padrões propriamente ditos, oferece-se ao leitor jovem e inexperiente o básico sobre os princípios de design de software que foram usados no raciocínio por trás da criação e da evolução desses padrões de projeto. Esta obra não leva o caro leitor cegamente pelo labirinto do mundo dos padrões, mas apresenta os fundamentos muito antes de abrir essa porta e conduzir o leitor por esse caminho de aprendizado.

O livro usa a linguagem Python para implementar os códigos de exemplo dos padrões – e isso faz bastante sentido. Como alguém que passou mais de doze anos na companhia dessa maravilhosa linguagem de programação, eu posso comprovar a sua beleza e simplicidade, além de sua eficácia na resolução de problemas que variam dos mais rotineiros até os mais complexos. Python é muito apropriada para o programador jovem e iniciante; pelo fato de ser fácil de aprender, também é muito divertido programar com essa linguagem. O jovem programador provavelmente achará o tempo desfrutado na companhia de Python ao longo do livro muito gratificante e produtivo.

Chetan Giridhar trabalha e contribui com Python há mais de sete anos. Ele é perfeitamente apto para a tarefa de escrever um livro como este, pois vivenciou alguns dos ciclos de aprendizado das complexidades de implementação e design e aprendeu muito durante esse processo. É muito conhecido por suas palestras sobre uma série de assuntos variados relacionados à Python e se apresenta para um público amplo em conferências de Python; por exemplo, a PyCon Índia. Já esteve entre os palestrantes convidados para conferências nos Estados Unidos, na Ásia/Pacífico e na Nova Zelândia.

Acredito que este livro, *Aprendendo padrões de projeto com Python*, será um acréscimo excelente à série de livros da Packt Publishing e possibilitará a inclusão de um conjunto de habilidades na caixa de ferramentas dos jovens programadores Python, conduzindo-os de forma gentil e inteligente até que sejam capazes de fazer um design de programas modulares e eficientes em Python.

Anand B. Pillai
CTO – Skoov.com
Membro do conselho da Python Software Foundation
Fundador do Bangalore Python User's Group

Sobre o autor

Chetan Giridhar é líder de tecnologia, entusiasta de código aberto e desenvolvedor Python. Escreveu vários artigos sobre tecnologia e práticas de desenvolvimento em revistas como *LinuxForYou* e *Agile Record* e publicou artigos técnicos no periódico *Python Papers*. Já fez palestras nas conferências PyCon, como PyCon Índia, Ásia-Pacífico e Nova Zelândia, e gosta de trabalhar com comunicações em tempo real, sistemas distribuídos e aplicações em nuvem. Chetan é revisor da Packt Publishing e tem contribuído com livros sobre IPython Visualizations e Core Python.

Gostaria de agradecer à equipe da Packt Publishing, especialmente a Merint Thomas Mathew, e ao revisor técnico Maurice HT Ling, por trazerem à tona o melhor conteúdo possível neste livro. Agradeço em especial ao meu mentor Anand B. Pillai, por ter gentilmente aceitado revisar este livro e escrever a apresentação. Este livro não teria sido possível sem a bênção de meus pais, Jyotsana e Jayant Giridhar, e o apoio e o incentivo constantes de minha esposa, Deepti, e de minha filha, Pihu!

Sobre o revisor

Maurice H. T. Ling programa em Python desde 2003. Completou seu doutorado em bioinformática e graduou-se (com honras) em biologia molecular e celular pela University of Melbourne. Atualmente, é pesquisador na Universidade Tecnológica de Nanyang em Singapura e membro honorário da University of Melbourne na Austrália. Maurice é editor-chefe da Computational and Mathematical Biology e coeditor de *The Python Papers*. Recentemente, Maurice participou da fundação da AdvanceSyn Pte. Ltd. – a primeira *startup* de biologia sintética em Singapura – como diretor e CTO (chief technology officer, ou diretor técnico). Também é o principal sócio da Colossus Technologies LLP de Singapura. Suas pesquisas concentram-se no interesse pela vida – vida biológica, vida artificial e inteligência artificial – e se utilizam da ciência da computação e da estatística como ferramentas para entender a vida e seus inúmeros aspectos. Em seu tempo livre, Maurice gosta de ler, apreciar uma xícara de café, escrever em seu diário pessoal ou filosofar sobre vários aspectos da vida. Você pode entrar em contato com ele pelo seu site ou por seu perfil no LinkedIn em *http://maurice.vodien.com* e em *http://www.linkedin.com/in/mauriceling*, respectivamente.

Prefácio

Os padrões de projeto estão entre os métodos mais eficazes para construir sistemas de software de grande porte. Com um foco cada vez maior na otimização dos níveis de arquitetura e design de software, é importante que os arquitetos de software pensem em otimizações na criação de objetos, na estrutura do código e na interação entre objetos nesses níveis. Isso garante que o custo da manutenção de software seja baixo e o código seja facilmente reutilizado e adaptável a mudanças. Além do mais, oferecer frameworks para reusabilidade e separação de responsabilidades é fundamental para o desenvolvimento de software atualmente.

O que este livro inclui

O Capítulo 1, Introdução aos padrões de projeto, apresenta o básico da programação orientada a objetos e discute princípios do design orientado a objetos em detalhes. Este capítulo contém uma breve introdução ao conceito de padrões de projeto para que você possa apreciar o contexto e a aplicação desses padrões no desenvolvimento de software.

O Capítulo 2, Padrão de projeto Singleton, discute um dos padrões de projeto de criação mais simples e conhecidos, usado no desenvolvimento de aplicações – o padrão de projeto Singleton. As maneiras diferentes por meio das quais podemos criar um padrão Singleton em Python também estão incluídas neste capítulo, com exemplos. Esse capítulo também discute o padrão de projeto Monostate (ou Borg), que é uma variante do padrão de projeto Singleton.

O Capítulo 3, Padrão Factory – construindo fábricas para criar objetos, discute outro padrão de criação, o padrão Factory (Fábrica). Você

conhecerá também os padrões Factory Method (Método de Fábrica) e Abstract Factory (Fábrica Abstrata) com um diagrama UML, cenários do mundo real e implementações com Python v3.5.

O *Capítulo 4*, *Padrão Façade – sendo adaptável com o Façade*, descreve outro tipo de padrão de projeto: o padrão estrutural. Seremos apresentados ao conceito de Façade (Fachada) e veremos como ele é aplicável ao design de software com a ajuda do padrão de projeto Façade. Você também conhecerá a sua implementação com uma aplicação Python de exemplo usando um cenário do mundo real.

O Capítulo 5, Padrão Proxy – controlando o acesso a objetos, trata do padrão Proxy, que se enquadra na categoria de padrões de projeto Estruturais. Seremos apresentados ao Proxy como um conceito, discutiremos o padrão de projeto e veremos como ele é usado no desenvolvimento de aplicações de software. Você também conhecerá as diferentes variantes do padrão Proxy: Virtual Proxy (Proxy Virtual), Smart Proxy (Proxy Inteligente), Remote Proxy (Proxy Remoto) e Protective Proxy (Proxy Protetor).

O Capítulo 6, Padrão Observer – de olho nos objetos, discute o terceiro tipo de padrão de projeto: o padrão comportamental. Seremos apresentados ao padrão de projeto Observer (Observador) com exemplos. Nesse capítulo, aprenderemos a implementar os modelos Push e Pull do padrão Observer e conheceremos os princípios do baixo acoplamento. Também veremos como a aplicação desse padrão é crucial para aplicações em nuvem e sistemas distribuídos.

O Capítulo 7, Padrão Command – encapsulando chamadas, descreve o padrão de projeto Command (Comando). Seremos apresentados a esse padrão e discutiremos como ele é usado no desenvolvimento de aplicações de software com um cenário do mundo real e uma implementação Python. Também analisaremos dois aspectos principais do padrão Command: uma implementação de operações refazer/rollback e a execução de tarefas assíncronas.

O Capítulo 8, Padrão Template Method – encapsulando algoritmos,

discute o padrão de projeto Template. Assim como o padrão Command, o padrão Template se enquadra na categoria dos padrões comportamentais. Discutiremos o padrão Template Method (Método Template), e você conhecerá o Hooks com uma implementação. Também discutiremos o princípio de Hollywood, que nos ajuda a apreciar melhor esse padrão.

O Capítulo 9, Model-View-Controller – padrões compostos, discute os padrões compostos. Seremos apresentados ao padrão de projeto Model-View-Controller (Modelo-Visão-Controlador) e discutiremos como ele é usado no desenvolvimento de aplicações de software. Sem dúvida, o MVC é um dos padrões de projeto mais usados; de fato, muitos frameworks Python são baseados nesse princípio. Você conhecerá os detalhes da implementação do MVC com um exemplo de aplicação escrito em Python Tornado (um framework utilizado pelo Facebook).

O Capítulo 10, Padrão de projeto State, apresenta o padrão de projeto State (Estado), que se enquadra na categoria de padrões comportamentais, assim como os padrões Command e Template. Discutiremos como ele é usado no desenvolvimento de aplicações de software.

O Capítulo 11, Antipadrões, discute os antipadrões – o que não devemos fazer como arquitetos ou engenheiros de software.

De que você precisa neste livro

Tudo que você precisa é de Python v3.5, e você pode fazer o seu download em *https://www.python.org/downloads/*.

A quem este livro se destina

Este livro foi escrito para desenvolvedores Python e arquitetos de software que se importam com princípios de design de software e detalhes dos aspectos do desenvolvimento de aplicações em Python. Ele exige uma compreensão básica dos conceitos de programação e

experiência com desenvolvimento em Python no nível de iniciantes. O livro também será útil para alunos e professores em ambientes de aprendizado ao vivo.

Convenções

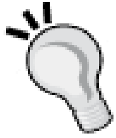
Neste livro, você encontrará vários estilos de texto que fazem distinção entre diversos tipos de informação. A seguir, apresentamos alguns exemplos desses estilos e uma explicação sobre os seus significados.

Palavras de código no texto, nomes de tabelas de banco de dados, nomes de pastas, nomes e extensões de arquivos, nomes de path, URLs dummy, entradas de usuário e handles do Twitter são apresentados da seguinte forma: "O objeto Car terá atributos como fuel level, isSedan, speed, steering wheel e coordinates."

Um bloco de código é apresentado assim:

```
class Person(object):
    def __init__(self, name, age): #construtor
        self.name = name #membros de dados/ atributos
        self.age = age
    def get_person(self,): # função-membro
         return "<Person (%s, %s)>" % (self.name, self.age)


p = Person("John", 32) # p é um objeto do tipo Person
print("Type of Object:", type(p), "Memory Address:", id(p))
```

Termos novos e **palavras importantes** são grafados em negrito. Palavras que você vê na tela, por exemplo, em menus ou em caixas de diálogo aparecem no texto assim: "Em Python, o conceito de encapsulamento (ocultar dados e métodos) não é implícito, pois não há palavras reservadas como **public**, **private** e **protected** (como em outras linguagens, por exemplo, C++ ou Java) necessárias para tratar esse conceito."

 Avisos ou notas importantes aparecem em uma caixa assim.

Dicas e truques aparecem assim.

CAPÍTULO 1
Introdução aos padrões de projeto

Neste capítulo, apresentaremos o básico sobre programação orientada a objetos e discutiremos os princípios do design orientado a objetos em detalhes. Isso nos deixará preparados para os assuntos mais avançados discutidos mais adiante no livro. Este capítulo também fará uma breve introdução ao conceito de padrões de projeto para que você possa apreciar o contexto e a aplicação dos padrões de projeto no desenvolvimento de software. Também classificaremos os padrões de projeto com base em três aspectos principais: padrões de criação, estruturais e comportamentais. Desse modo, essencialmente, a discussão deste capítulo envolverá os seguintes tópicos:

- compreender a programação orientada a objetos;
- discutir os princípios do design orientado a objetos;
- entender o conceito de padrões de projeto, sua taxonomia e o contexto;
- discutir padrões para linguagens dinâmicas;
- classificar os padrões em padrões de criação, padrões estruturais e padrões comportamentais.

Compreendendo a programação orientada a objetos

Antes de começar o aprendizado sobre padrões de projeto, é sempre bom abordar o básico e descrever os paradigmas orientados a objetos em

Python. O mundo da orientação a objetos apresenta o conceito de *objetos* que têm atributos (membros de dados) e procedimentos (funções-membro). Essas funções são responsáveis pela manipulação dos atributos. Por exemplo, considere um objeto Car. O objeto Car terá atributos como fuel level, isSedan, speed, steering wheel e coordinates, e os métodos seriam accelerate() para aumentar a velocidade e takeLeft() para fazer o carro virar à esquerda. Python é uma linguagem orientada a objetos desde o seu lançamento inicial. Como dizem por aí, *tudo em Python é um objeto*. Toda instância de classe ou variável tem seu próprio endereço de memória ou sua identidade. Os objetos, que são instâncias de classes, interagem entre si para servir ao propósito de uma aplicação em desenvolvimento. Entender os conceitos essenciais da programação orientada a objetos envolve compreender os conceitos de objetos, classes e métodos.

Objetos

Os pontos a seguir descrevem os objetos:

- Eles representam entidades em sua aplicação em desenvolvimento.
- As entidades interagem entre si para resolver problemas do mundo real.
- Por exemplo, Person é uma entidade, e Car também é uma entidade. Person dirige Car para se deslocar de um lugar para outro.

Classes

As classes ajudam os desenvolvedores a representar entidades do mundo real:

- As classes definem objetos com atributos e comportamentos. Os atributos são membros de dados e os comportamentos são manifestados pelas funções-membro.
- As classes são constituídas de construtores que proporcionam o estado inicial para esses objetos.

- As classes são como templates, portanto, podem ser facilmente reutilizadas.

Por exemplo, a classe Person terá atributos name e age e uma função-membro gotoOffice() que define o comportamento de se deslocar até o escritório para trabalhar.

Métodos

Os pontos a seguir discutem o que os métodos fazem no mundo orientado a objetos:

- Eles representam o comportamento do objeto.
- Os métodos operam em atributos, além de implementarem a funcionalidade desejada.

A seguir, apresentamos um bom exemplo de uma classe e de um objeto criados com Python v3.5:

```
class Person(object):
    def __init__(self, name, age): #construtor
        self.name = name #membros de dados/ atributos
        self.age = age
    def get_person(self,): #função-membro
        return "<Person (%s, %s)>" % (self.name, self.age)


p = Person("John", 32) # p é um objeto do tipo Person
print("Type of Object:", type(p), "Memory Address:", id(p))
```

A saída do código anterior deve ter o seguinte aspecto:

Principais aspectos da programação orientada a objetos

Agora que já entendemos o básico sobre a programação orientada a objetos, vamos explorar os seus aspectos principais.

Encapsulamento

As principais características do encapsulamento são:

- O comportamento de um objeto permanece oculto para o mundo externo, ou os objetos mantêm suas informações de estado como privadas.
- Os clientes não podem alterar o estado interno dos objetos atuando diretamente neles; em vez disso, eles requisitam o objeto enviando mensagens. Com base no tipo das requisições, os objetos podem responder alterando o seu estado interno utilizando funções-membro especiais como `get` e `set`.
- Em Python, o conceito de encapsulamento (ocultar dados e métodos) não é implícito, pois não há palavras reservadas como `public`, `private` e `protected` (como em outras linguagens, por exemplo, C++ ou Java), necessárias para tratar esse conceito. É claro que podemos tornar a acessibilidade privada usando o prefixo __ no nome da variável ou da função.

Polimorfismo

As principais características do polimorfismo estão descritas a seguir.

- O polimorfismo pode ser de dois tipos:
  - um objeto oferece diferentes implementações do método de acordo com os parâmetros de entrada;
  - a mesma interface pode ser usada por objetos de tipos diferentes.
- Em Python, o polimorfismo é um recurso embutido na linguagem. Por exemplo, o operador + pode atuar em dois inteiros para somá-los ou pode operar em strings para concatená-las.

No exemplo a seguir, strings, tuplas ou listas podem ser acessadas com um índice inteiro. Isso mostra como Python demonstra o polimorfismo em tipos embutidos:

```
a = "John"
b = (1,2,3)
c = [3,4,6,8,9]
print(a[1], b[0], c[2])
```

Herança

Os conceitos a seguir nos ajudam a entender melhor o processo de herança.

- A herança indica que uma classe deriva (a maior parte de) sua funcionalidade da classe-pai.
- A herança é descrita como uma opção para reutilizar funcionalidades definidas na classe-base e permite extensões independentes da implementação do software original.
- A herança cria hierarquias por meio de relacionamentos entre objetos de classes diferentes. Em Python, de modo diferente de Java, há suporte para herança múltipla (herdar de várias classes-bases).

No código de exemplo a seguir, `class A` é a classe-base e `class B` deriva suas características da `class A`. Assim, os métodos da `class A` podem ser acessados pelo objeto da `class B`:

```
class A:
    def a1(self):
        print("a1")

class B(A):
    def b(self):
        print("b")


b = B()
b.a1()
```

Abstração

Estas são as principais características da abstração:

- Oferece uma interface simples aos clientes, por meio da qual eles podem interagir com os objetos da classe e chamar métodos definidos na interface.
- Abstrai a complexidade de classes internas com uma interface, de modo que o cliente não tenha de conhecer as implementações internas.

No exemplo a seguir, os detalhes internos da classe Adder são abstraídos com o método add():

```
class Adder:
    def __init__(self):
        self.sum = 0
    def add(self, value):
        self.sum += value

acc = Adder()
for i in range(99):
    acc.add(i)


print(acc.sum)
```

Composição

A composição se refere aos seguintes pontos:

- É uma maneira de combinar objetos ou classes em estruturas de dados ou implementações de software mais complexas.
- Na composição, um objeto é usado para chamar funções-membro em outros módulos, disponibilizando, assim, as funcionalidades básicas para os módulos sem o uso de herança.

No exemplo a seguir, há uma composição da `class A` dentro da `class B`:

```
class A(object):
    def a1(self):
        print("a1")

class B(object):
    def b(self):
        print("b")
        A().a1()


objectB = B()
objectB.b()
```

Princípios do design orientado a objetos

Vamos agora discutir outro conjunto de conceitos que será fundamental para nós. São os princípios do design orientado a objetos, que atuarão como uma caixa de ferramentas para nós enquanto conhecemos os padrões de projeto em detalhes.

Princípio do aberto/fechado

O princípio do aberto/fechado determina que *classes ou objetos e métodos devem estar abertos para extensão, mas fechados para modificações.*

Em uma linguagem simples, isso significa que, ao desenvolver sua aplicação de software, você deverá garantir que escreverá suas classes ou

seus módulos de forma genérica, de modo que, sempre que sentir a necessidade de estender o comportamento da classe ou do objeto, você não precisará alterar a classe propriamente dita. Em vez disso, uma extensão simples da classe deverá ajudar a implementar o novo comportamento.

Por exemplo, o princípio do aberto/fechado é manifestado em um caso em que um usuário tenha de criar uma implementação de classe estendendo a classe-base abstrata a fim de implementar o comportamento necessário, em vez de alterar a classe abstrata.

As vantagens desse princípio de design são:

- As classes existentes não são alteradas e, desse modo, as chances de regressão são menores.
- Ajuda a manter a compatibilidade com versões de código anteriores.

Princípio da inversão de controle

O princípio da inversão de controle determina que *módulos de alto nível não devem ser dependentes de módulos de baixo nível; ambos devem ser dependentes de abstrações. Os detalhes devem depender das abstrações, e não o inverso*.

Esse princípio sugere que quaisquer dois módulos não devem ser altamente dependentes um do outro. De fato, o módulo-base e o módulo dependente devem estar desacoplados com uma camada de abstração entre eles.

Esse princípio também sugere que os detalhes de sua classe devem representar as abstrações. Em alguns casos, a filosofia é invertida e os próprios detalhes da implementação determinam a abstração, o que deve ser evitado.

As vantagens do princípio da inversão do controle são:

- O alto acoplamento entre módulos deixa de ser predominante e, portanto, não há complexidade/rigidez no sistema.
- Como há uma camada evidente de abstração entre os módulos

dependentes (proporcionada por um hook [gancho] ou um parâmetro), é fácil lidar com dependências entre os módulos de uma forma mais conveniente.

Princípio da segregação de interfaces

Conforme determina o princípio da segregação de interfaces, *os clientes não devem ser forçados a depender de interfaces que não utilizam.*

Esse princípio diz respeito a desenvolvedores de software escreverem boas interfaces. Por exemplo, ele lembra os desenvolvedores/arquitetos que eles devem desenvolver métodos relacionados à funcionalidade. Se houver algum método não relacionado à interface, a classe dependente da interface terá de implementá-lo desnecessariamente.

Por exemplo, uma interface `Pizza` não deve ter um método chamado `add_chicken()`. A classe `Veg Pizza` baseada na interface `Pizza` não deve ser forçada a implementar esse método.

As vantagens desse princípio de design são:

- Ele força os desenvolvedores a escrever interfaces enxutas e a ter métodos que sejam específicos da interface.
- Ajuda você a não encher as interfaces com métodos indesejados.

Princípio da responsabilidade única

Conforme determina o princípio da responsabilidade única, *uma classe deve ter apenas um motivo para mudar.*

Segundo esse princípio, quando desenvolvemos classes, elas devem cuidar bem de sua funcionalidade em particular. Se uma classe estiver tratando duas funcionalidades, será melhor dividi-la. O princípio refere-se à funcionalidade como um motivo para mudança. Por exemplo, uma classe pode passar por mudanças por causa da diferença de comportamento esperada dela, mas se uma classe for modificada por dois motivos (basicamente, mudanças em duas funcionalidades), então ela deverá ser definitivamente separada.

As vantagens desse princípio de design são:

- Sempre que houver uma mudança em uma funcionalidade, essa classe em particular deverá ser alterada, e nada mais.
- Além disso, se uma classe tiver várias funcionalidades, as classes dependentes deverão passar por mudanças em razão de diversos motivos, o que deve ser evitado.

Princípio da substituição

O princípio da substituição determina que *classes derivadas devem ser capazes de substituir totalmente as classes-bases.*

Esse princípio é bem simples, pois afirma que quando os desenvolvedores de aplicação escrevem classes derivadas, eles devem estender as classes-bases. Também sugere que a classe derivada deve estar o mais próximo possível da classe-base, de modo que a classe derivada possa substituir a classe-base sem qualquer modificação no código.

Conceito de padrões de projeto

Por fim, é hora de começarmos a falar sobre padrões de projeto! O que são padrões de projeto?

Os padrões de projeto foram inicialmente introduzidos pela GoF (Gang of Four, ou Gangue dos Quatro), que os mencionou como soluções para determinados problemas. Se quiser saber mais, GoF refere-se aos quatro autores do livro *Padrões de projeto: soluções reutilizáveis de software orientado a objetos*. Os autores do livro são *Erich Gamma*, *Richard Helm*, *Ralph Johnson e John Vlissides*, com apresentação de *Grady Booch*. Esse livro discute soluções de engenharia de software para problemas que ocorrem comumente em design de software. Inicialmente foram identificados 23 padrões de projeto, e a primeira implementação foi feita com base na linguagem de programação Java. Os padrões de projeto não são invenções, mas descobertas.

As principais características dos padrões de projeto são estas:

- São independentes de linguagem e podem ser implementados em linguagens diferentes.
- São dinâmicos, e novos padrões são introduzidos ocasionalmente.
- São passíveis de personalização e, portanto, são úteis aos desenvolvedores.

No início, quando você ouvir falar de padrões de projeto, poderá ter a seguinte impressão:

- Eles são uma panaceia para todos os problemas de design que você teve até agora.
- São uma maneira extraordinária e especialmente inteligente de resolver um problema.
- Muitos experts do mundo do desenvolvimento de software concordam com essas soluções.
- Há algo repetível no design – daí a palavra padrão.

Você também já deve ter tentado solucionar os problemas que um padrão de projeto procura resolver, mas talvez sua solução tenha sido incompleta, e a completude que estamos buscando é inerente ou está implícita no padrão de projeto. Quando dizemos completude, ela pode se referir a muitos fatores, como design, escalabilidade, reutilização, utilização de memória e outros. Essencialmente, um padrão de projeto diz respeito a aprender com o sucesso de outros, e não com suas próprias falhas!

Outra discussão interessante que surge em relação aos padrões de projeto é: quando deve usá-los? É na fase de análise ou de design do SDLC (Software Development Life Cycle, ou Ciclo de Vida do Desenvolvimento de Software)?

O interessante é que os padrões de projeto são soluções para problemas conhecidos. Desse modo, eles podem ser muito utilizados na análise ou no design e, conforme esperado, na fase de desenvolvimento, por causa da relação direta com o código da aplicação.

Vantagens dos padrões de projeto

As vantagens dos padrões de projeto são estas:

- Os padrões são reutilizáveis em vários projetos.
- Problemas no nível de arquitetura podem ser solucionados.
- Os padrões resistiram ao teste do tempo e sua eficácia foi comprovada, pois incluem a experiência dos desenvolvedores e arquitetos.
- Apresentam confiabilidade e podemos contar com eles.

Taxonomia dos padrões de projeto

Nem todo código ou design pode ser classificado como um padrão de projeto. Por exemplo, uma construção ou estrutura de dados de programação que resolva um problema não pode ser chamada de padrão. Vamos entender os termos de forma bem simplista.

- **Trecho de código**: é um código em alguma linguagem que serve a determinado propósito, por exemplo, uma conexão com um banco de dados em Python pode ser um trecho de código.
- **Design**: é uma solução melhor para resolver um problema em particular.
- **Convenção**: é uma maneira de resolver algum tipo de problema, e pode ser bem genérica e aplicável a uma situação que se tenha em mãos.
- **Padrão**: é uma solução eficiente e escalável, resistente ao teste do tempo, que resolverá toda uma classe de problemas conhecidos.

Contexto – aplicabilidade dos padrões de projeto

Para usar os padrões de projeto de modo eficiente, os desenvolvedores de aplicações devem ter conhecimento do contexto em que os padrões se aplicam. Podemos classificar o contexto nas seguintes categorias

principais:

- **Participantes** – são as classes usadas nos padrões de projeto. As classes desempenham diferentes papéis para atingir vários objetivos no padrão.
- **Requisitos não funcionais** – requisitos como otimização de memória, usabilidade e desempenho se enquadram nessa categoria. Esses fatores exercem impacto na solução de software completa e, portanto, são essenciais.
- **Negociações** – nem todos os padrões de projeto se enquadram no desenvolvimento de aplicações exatamente como estão, portanto negociações se fazem necessárias. São decisões que você deve tomar quando usar um padrão de projeto em uma aplicação.
- **Resultados** – os padrões de projeto podem ter um impacto negativo em outras partes do código caso o contexto não seja apropriado. Os desenvolvedores devem compreender as consequências do uso dos padrões de projeto.

Padrões para linguagens dinâmicas

Python é uma linguagem dinâmica, como Lisp. A natureza dinâmica de Python pode ser descrita da seguinte maneira:

- Tipos ou classes são objetos em tempo de execução.
- As variáveis podem ter um tipo a partir de um valor e podem ser modificadas em tempo de execução. Por exemplo, em `a = 5` e `a = "John"`, a variável `a` recebe um valor em tempo de execução e o tipo também muda.
- Linguagens dinâmicas têm mais flexibilidade no que diz respeito a restrições de classe.
- Por exemplo, em Python, o polimorfismo está embutido na linguagem; não há palavras reservadas como `private` e `protected`; e tudo é público por padrão.

- Python representa um caso em que padrões de projeto podem ser facilmente implementados em linguagens dinâmicas.

Classificando os padrões

O livro da GoF sobre padrões de projeto discute 23 padrões e os classifica em três categorias principais:

- padrões de criação;
- padrões estruturais;
- padrões comportamentais.

A classificação dos padrões é feita principalmente com base na forma como os objetos são criados, as classes e os objetos são estruturados em uma aplicação de software e inclui também a forma como os objetos interagem entre si. Vamos discutir cada uma das categorias em detalhes nesta seção.

Padrões de criação

A seguir, apresentamos as propriedades dos padrões de criação:

- funcionam com base no modo como os objetos podem ser criados;
- isolam os detalhes da criação dos objetos;
- o código é independente do tipo do objeto a ser criado.

Um exemplo de um padrão de criação é o padrão Singleton.

Padrões estruturais

A seguir, apresentamos as propriedades dos padrões estruturais:

- eles determinam o design da estrutura de objetos e classes para que estes possam ser compostos e resultados mais amplos sejam alcançados;
- o foco está em simplificar a estrutura e identificar o relacionamento entre classes e objetos;
- estão centrados em herança e composição de classes.

Um exemplo de um padrão estrutural é o padrão Adapter (Adaptador).

Padrões comportamentais

A seguir, apresentamos as propriedades dos padrões comportamentais:

- estão preocupados com a interação entre os objetos e suas responsabilidades;
- os objetos devem ser capazes de interagir e, mesmo assim, devem ter baixo acoplamento.

Um exemplo de um padrão comportamental é o padrão Observer (Observador).

Resumo

Neste capítulo, aprendemos os conceitos básicos da programação orientada a objetos, como objetos, classes e variáveis, e vimos características como polimorfismo, herança e abstração, juntamente com exemplos de código.

Também conhecemos agora os princípios do design orientado a objetos que nós como desenvolvedores/arquitetos devemos considerar quando fizermos o design de uma aplicação.

Por fim, prosseguimos explorando mais informações sobre os padrões de projeto, suas aplicações e o contexto em que eles podem ser aplicados, além de discutir sua classificação.

No final deste capítulo, estamos prontos para dar o próximo passo e estudar os padrões de projeto em detalhes.

CAPÍTULO 2

Padrão de projeto Singleton

No capítulo anterior, exploramos os padrões de projeto e sua classificação. Como já sabemos, os padrões de projeto podem ser classificados em três categorias principais: padrões estruturais, comportamentais e de criação.

Neste capítulo, discutiremos o padrão de projeto Singleton – um dos padrões de criação mais simples e conhecidos, usado no desenvolvimento de aplicações. Também teremos uma rápida introdução ao padrão Singleton, descreveremos um exemplo do mundo real em que esse padrão pode ser usado e o explicaremos em detalhes com a ajuda de implementações em Python. Você conhecerá o padrão de projeto Monostate (ou Borg), que é uma variante do padrão de projeto Singleton.

Os seguintes tópicos serão abordados de forma sucinta:

- uma explicação sobre o padrão de projeto Singleton;
- um exemplo do mundo real do padrão Singleton;
- a implementação do padrão Singleton em Python;
- o padrão Monostate (Borg).

No final do capítulo, faremos um pequeno resumo dos Singletons. Isso ajudará você a pensar em alguns dos aspectos do padrão de projeto Singleton por conta própria.

Compreendendo o padrão de

projeto Singleton

O Singleton proporciona uma forma de ter um e somente um objeto de determinado tipo, além de disponibilizar um ponto de acesso global. Por isso, os Singletons são geralmente usados em casos como logging ou operações de banco de dados, spoolers de impressão e muitos outros cenários em que seja necessário que haja apenas uma instância disponível para toda a aplicação a fim de evitar requisições conflitantes para o mesmo recurso. Por exemplo, podemos querer usar um único objeto de banco de dados para executar operações e manter a consistência dos dados, ou um objeto da classe logging para vários serviços, de modo a descarregar as mensagens de log em um arquivo em particular, sequencialmente.

Em suma, com o padrão de projeto Singleton, as intenções são:

- garantir que um e somente um objeto da classe seja criado;
- oferecer um ponto de acesso para um objeto que seja global no programa;
- controlar o acesso concorrente a recursos compartilhados.

A seguir, apresentamos o diagrama UML do Singleton:

Aprendendo Padrões
de Projeto em Python

Tire proveito da eficácia dos padrões de projeto (design patterns) em Python
para resolver problemas do mundo real em arquitetura e design de software

Uma maneira simples de implementar o Singleton é deixar o construtor privado e criar um método estático que faça a inicialização do objeto. Dessa forma, um objeto é criado na primeira chamada e a classe devolverá o mesmo objeto a partir daí.

Em Python, implementaremos esse padrão de um modo diferente, pois não há opção para criar construtores privados. Vamos dar uma olhada em como os Singletons são implementados na linguagem Python.

Implementando um Singleton clássico em

Python

Eis um código de exemplo do padrão Singleton em Python v3.5. Nesse exemplo, teremos duas tarefas principais:

1. Permitiremos a criação de apenas uma instância da classe Singleton.
2. Se uma instância já existir, serviremos o mesmo objeto novamente.

O código a seguir mostra isso:

```
class Singleton(object):
    def __new__(cls):
        if not hasattr(cls, 'instance'):
            cls.instance = super(Singleton, cls).__new__(cls)
        return cls.instance

s = Singleton()
print("Object created", s)

s1 = Singleton()
print("Object created", s1)
```

Eis a saída do trecho de código anterior:

```
Object created <__main__.Singleton object at 0x102078ba8>
Object created <__main__.Singleton object at 0x102078ba8>
```

No trecho de código anterior, sobrescrevemos o método __new__ (método especial de Python para instanciar objetos) para controlar a criação do objeto. O objeto s é criado com o método __new__, mas antes disso é feita uma verificação para saber se o objeto já existe. O método hasattr (método especial de Python para saber se um objeto tem determinada propriedade) é usado para verificar se o objeto cls tem a propriedade instance, que confere se a classe já tem um objeto. No momento em que o objeto s1 é solicitado, hasattr() detecta que um objeto já existe e, então, s1 aloca a instância do objeto existente (localizado em 0x102078ba8).

Instanciação preguiçosa no padrão Singleton

Um dos casos de uso para o padrão Singleton é a instanciação preguiçosa (lazy instantiation). Por exemplo, no caso das importações de módulos, podemos acidentalmente criar um objeto mesmo quando ele não for necessário. A instanciação preguiçosa garante que o objeto seja criado quando realmente precisarmos dele. Considere a instanciação preguiçosa como uma maneira de trabalhar com recursos reduzidos e criá-los somente quando houver necessidade.

No código de exemplo a seguir, quando fazemos s=Singleton(), o método __init__ é chamado, mas nenhum objeto novo é criado. Entretanto a criação propriamente dita do objeto ocorre quando chamamos Singleton.getInstance(). É assim que fazemos uma instanciação preguiçosa.

```
class Singleton:
    __instance = None
    def __init__(self):
        if not Singleton.__instance:
            print(" __init__ method called..")
        else:
            print("Instance already created:", self.getInstance())
    @classmethod
    def getInstance(cls):
        if not cls.__instance:
            cls.__instance = Singleton()
        return cls.__instance


s = Singleton() ## classe é inicializada, mas o objeto não é criado
print("Object created", Singleton.getInstance()) # O objeto é criado aqui
s1 = Singleton() ## instância já criada
```

Singletons no nível de módulo

Todos os módulos são Singletons por padrão por causa do comportamento de importação de Python. Python funciona da seguinte maneira:

1. Ele verifica se um módulo Python foi importado.

2. Em caso afirmativo, devolve o objeto para o módulo. Caso contrário, importa e instancia o módulo.

3. Assim, quando um módulo for importado, ele será inicializado. No entanto, quando o mesmo módulo for importado novamente, ele não será inicializado mais uma vez, o que está relacionado ao comportamento do Singleton, segundo o qual deve haver apenas um objeto e esse mesmo objeto deve ser devolvido.

Padrão Singleton Monostate

Discutimos a Gangue dos Quatro e seu livro no *Capítulo 1, Introdução aos padrões de projeto.* Segundo o padrão de projeto Singleton da GoF, deve haver um e somente um objeto de uma classe. Entretanto, de acordo com Alex Martelli, o que um programador geralmente precisa é de instâncias que compartilhem o mesmo estado. Ele sugere que os desenvolvedores devem se preocupar com o estado e o comportamento, e não com a identidade. Como o conceito é baseado no compartilhamento do mesmo estado por todos os objetos, esse padrão também é conhecido como Monostate.

O padrão Monostate pode ser implementado de forma muito simples em Python. No código a seguir, atribuímos a variável de classe `__shared_state` à variável `__dict__` (uma variável especial em Python). Python usa `__dict__` para armazenar o estado de todos os objetos de uma classe. No código a seguir, atribuímos `__shared_state` propositalmente a todas as instâncias criadas. Então, se criarmos duas instâncias `'b'` e `'b1'`, teremos dois objetos distintos, de modo diferente do Singleton, em que temos apenas um objeto. Entretanto os estados dos objetos, `b.__dict__` e `b1.__dict__`, são iguais. Mesmo que a variável `x` do objeto mude para o objeto `b`, a mudança será copiada para a variável `__dict__` compartilhada entre todos os objetos, e até mesmo `b1` obterá a mudança de `x` de um para quatro:

```
class Borg:
    __shared_state = {"1":"2"}
    def __init__(self):
```

```
        self.x = 1
        self.__dict__ = self.__shared_state
        pass

b = Borg()
b1 = Borg()
b.x = 4

print("Borg Object 'b': ", b) ## b e b1 são objetos distintos
print("Borg Object 'b1': ", b1)
print("Object State 'b':", b.__dict__)## b e b1 compartilham o mesmo estado
print("Object State 'b1':", b1.__dict__)
```

A seguir, apresentamos a saída do trecho de código anterior:

```
Borg Object 'b':  <__main__.Borg object at 0x102078da0>
Borg Object 'b1':  <__main__.Borg object at 0x102078dd8>
Object State 'b': {'x': 4, '1': '2'}
Object State 'b1': {'x': 4, '1': '2'}
```

Outra forma de implementar o padrão Borg é ajustar o próprio método __new__. Como sabemos, esse método é responsável pela criação da instância do objeto:

```
class Borg(object):
    _shared_state = {}
    def __new__(cls, *args, **kwargs):
        obj = super(Borg, cls).__new__(cls, *args, **kwargs)
        obj.__dict__ = cls._shared_state
        return obj
```

Singletons e metaclasses

Vamos começar com uma rápida introdução às metaclasses. Uma metaclasse é uma classe de outra classe, o que significa que a classe é uma instância de sua metaclasse. Com as metaclasses, os programadores têm a oportunidade de criar classes de seus próprios tipos a partir de classes Python predefinidas. Por exemplo, se você tiver um objeto MyClass, poderá criar uma metaclasse MyKls, que redefine o comportamento de MyClass conforme for necessário. Vamos entendê-las em detalhes.

Em Python, tudo é um objeto. Se dissermos a=5, então type(a) devolve <type 'int'>, que significa que a é do tipo int. No entanto type(int) devolve <type 'type'>, que sugere a presença de uma metaclasse, pois int é uma classe do tipo type.

A definição de classe é determinada por sua metaclasse, portanto, quando criamos uma classe com class A, Python a cria com A = type(name, bases, dict).

- name é o nome da classe.
- base é a classe-base.
- dict é a variável de atributos.

Se uma classe tiver uma metaclasse predefinida (com o nome MetaKls), Python criará a classe com A = MetaKls(name, bases, dict).

Vamos dar uma olhada em um exemplo de implementação de metaclasse em Python 3.5:

```
class MyInt(type):
    def __call__(cls, *args, **kwds):
        print("***** Here's My int *****", args)
        print("Now do whatever you want with these objects...")
        return type.__call__(cls, *args, **kwds)


class int(metaclass=MyInt):
    def __init__(self, x, y):
        self.x = x
        self.y = y


i = int(4,5)
```

A seguir, apresentamos a saída do trecho de código anterior:

```
***** Here's My int ***** (4, 5)
Now do whatever you want with these objects...
```

O método especial __call__ de Python é chamado quando um objeto precisa ser criado para uma classe já existente. Nesse código, quando

instanciamos a classe int com int(4,5), o método __call__ da metaclasse MyInt é chamado, o que significa que a metaclasse agora controla a instanciação do objeto. Uau, isso não é ótimo?

A filosofia anterior é usada também no padrão de projeto Singleton. Como a metaclasse tem mais controle sobre a criação da classe e a instanciação de objetos, ela pode ser usada para criar Singletons. (Observação: para controlar a criação e a inicialização de uma classe, as metaclasses sobrescrevem os métodos __new__ e __init__.)

A implementação do Singleton com metaclasses pode ser mais bem explicada com o código de exemplo a seguir:

```
class MetaSingleton(type):
    _instances = {}
    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            cls._instances[cls] = super(MetaSingleton, \
                cls).__call__(*args, **kwargs)
        return cls._instances[cls]

class Logger(metaclass=MetaSingleton):
    pass

logger1 = Logger()
logger2 = Logger()
print(logger1, logger2)
```

Um cenário do mundo real – o padrão Singleton, parte 1

Como um caso de uso prático, veremos uma aplicação de banco de dados para demonstrar como Singletons podem ser usados. Considere um exemplo de um serviço de nuvem que envolva várias operações de leitura e escrita no banco de dados. O serviço de nuvem completo está dividido em vários serviços que executam operações no banco de dados. Uma ação na UI (aplicação web) chamará internamente uma API que, em algum momento, resultará em uma operação de banco de dados.

É evidente que o recurso compartilhado pelos diferentes serviços é o próprio banco de dados. Portanto, se precisarmos fazer um design melhor do serviço de nuvem, devemos prestar atenção nos seguintes pontos:

- consistência entre as operações no banco de dados – uma operação não deve resultar em conflitos com outras operações.
- a utilização de memória e de CPU deve estar otimizada para o tratamento de várias operações no banco de dados.

Apresentamos a seguir um exemplo de implementação com Python:

```
import sqlite3
class MetaSingleton(type):
    _instances = {}
    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            cls._instances[cls] = super(MetaSingleton, \
                cls).__call__(*args, **kwargs)
        return cls._instances[cls]

class Database(metaclass=MetaSingleton):
connection = None
def connect(self):
    if self.connection is None:
        self.connection = sqlite3.connect("db.sqlite3")
        self.cursorobj = self.connection.cursor()
    return self.cursorobj

db1 = Database().connect()
db2 = Database().connect()

print ("Database Objects DB1", db1)
print ("Database Objects DB2", db2)
```

Eis a saída do trecho de código anterior:

```
Database Objects DB1 <sqlite3.Cursor object at 0x102464570>
Database Objects DB2 <sqlite3.Cursor object at 0x102464570>
```

No código anterior, podemos ver que os pontos a seguir foram incluídos:

1. Criamos uma metaclasse chamada MetaSingleton. Conforme explicamos na seção anterior, o método especial __call__ de Python é usado na metaclasse para criar um Singleton.
2. A classe database está decorada com a classe MetaSingleton e começa a agir como um Singleton. Assim, quando a classe database é instanciada, ela cria apenas um objeto.
3. Quando a aplicação web quiser executar determinadas operações no banco de dados, ela instanciará a classe de banco de dados várias vezes, mas somente um objeto será criado. Como existe apenas um objeto, as chamadas ao banco de dados serão sincronizadas. Além do mais, isso não é custoso para os recursos do sistema, e podemos evitar problemas com recursos de memória ou CPU.

Considere que, em vez de ter uma aplicação web, temos uma configuração em cluster, com várias aplicações web, porém apenas um banco de dados. Esta não é uma boa situação para os Singletons porque, a cada adição de uma aplicação web, um novo Singleton será criado e um novo objeto que faz consultas no banco de dados será adicionado. Isso resulta em operações não sincronizadas no banco de dados e exige muito dos recursos. Em casos assim, implementar um *pool* de conexões com o banco de dados será melhor do que implementar Singletons.

Um cenário do mundo real – o padrão Singleton, parte 2

Vamos considerar outro cenário em que implementaremos serviços de verificação de sanidade (como o Nagios) para a nossa infraestrutura. Criamos a classe HealthCheck, que é implementada como um Singleton. Vamos manter também uma lista de servidores nos quais a verificação de sanidade deve ser feita. Se um servidor for removido dessa lista, o software de verificação de sanidade deverá detectar isso e removê-lo dos servidores configurados para ser verificados.

No código a seguir, os objetos hc1 e hc2 são iguais, pois a classe é um

Singleton.

Os servidores são adicionados à infraestrutura para verificação de sanidade com o método addServer(). Inicialmente, a iteração da verificação de sanidade é executada nesses servidores. O método changeServer() remove o último servidor e adiciona outro à infraestrutura programada para a verificação de sanidade. Desse modo, quando a verificação de sanidade é executada na segunda iteração, ela tem a lista de servidores alterada.

Tudo isso é possível com Singletons. Quando os servidores são adicionados ou removidos, a verificação de sanidade deve ser o mesmo objeto que tem o conhecimento das mudanças feitas na infraestrutura:

```
class HealthCheck:
    _instance = None
    def __new__(cls, *args, **kwargs):
        if not HealthCheck._instance:
            HealthCheck._instance = super(HealthCheck, \
                cls).__new__(cls, *args, **kwargs)
        return HealthCheck._instance
    def __init__(self):
        self._servers = []
    def addServer(self):
        self._servers.append("Server 1")
        self._servers.append("Server 2")
        self._servers.append("Server 3")
        self._servers.append("Server 4")
    def changeServer(self):
        self._servers.pop()
        self._servers.append("Server 5")

hc1 = HealthCheck()
hc2 = HealthCheck()

hc1.addServer()
print("Schedule health check for servers (1)..")
for i in range(4):
    print("Checking ", hc1._servers[i])
```

```
hc2.changeServer()
print("Schedule health check for servers (2)..")
for i in range(4):
    print("Checking ", hc2._servers[i])
```

Esta é a saída desse código:

```
Schedule health check for servers (1)..
Checking  Server 1
Checking  Server 2
Checking  Server 3
Checking  Server 4
Schedule health check for servers (2)..
Checking  Server 1
Checking  Server 2
Checking  Server 3
Checking  Server 5
```

Desvantagens do padrão Singleton

Embora os Singletons sejam usados em vários lugares com um ótimo efeito, esse padrão pode apresentar algumas complicações. Como os Singletons têm um ponto de acesso global, os problemas a seguir podem ocorrer:

- Variáveis globais podem ser alteradas por engano em um lugar e, como o desenvolvedor pode achar que elas permaneceram inalteradas, as variáveis poderão acabar sendo usadas em outro lugar na aplicação.
- Várias referências podem ser criadas para o mesmo objeto. Como o Singleton cria apenas um objeto, várias referências podem ser criadas nesse ponto para o mesmo objeto.
- Todas as classes que são dependentes de variáveis globais acabam se tornando altamente acopladas, pois uma mudança feita por uma classe no dado global poderá inadvertidamente exercer impacto em outra classe.

Como parte deste capítulo, aprendemos bastante sobre Singletons. Eis alguns pontos que devemos lembrar a respeito desse padrão:

- Há muitas aplicações do mundo real em que precisamos criar apenas um objeto, como pools de threads, caches, caixas de diálogo, configurações de registro, e assim por diante. Se criarmos várias instâncias para cada uma dessas aplicações, teremos um uso excessivo de recursos. Os Singletons funcionam muito bem em situações como essas.
- O Singleton é um método comprovado, resistente ao teste do tempo, e oferece um ponto de acesso global sem muitas desvantagens.
- É claro que há algumas desvantagens. Os Singletons podem exercer um impacto inesperado por trabalharem com variáveis globais ou por instanciarem classes que exigem muitos recursos, mas que acabam por não utilizá-los.

Resumo

Neste capítulo, conhecemos o padrão de projeto Singleton e o contexto em que ele é usado. Compreendemos que os Singletons são utilizados quando precisamos ter apenas um objeto para uma classe.

Também vimos diversas maneiras pelas quais os Singletons podem ser implementados em Python. A implementação clássica permite várias tentativas de instanciação, mas devolve o mesmo objeto.

Também discutimos o padrão Borg ou Monostate, que é uma variação do padrão Singleton. O Borg permite a criação de vários objetos que compartilham o mesmo estado, de modo diferente do padrão de objeto único descrito pela GoF.

Prosseguimos explorando a aplicação web em que o Singleton pode ser aplicado para operações consistentes em banco de dados entre vários serviços.

Por fim, também vimos situações em que os Singletons podem dar errado e quais são as situações que os desenvolvedores devem evitar.

No final deste capítulo, estamos suficientemente à vontade para dar o próximo passo e estudar outros padrões de criação dos quais podemos nos beneficiar.

No próximo capítulo, veremos outro padrão de criação e o padrão de

projeto Factory (Fábrica). Discutiremos os padrões Factory Method (Método de Fábrica) e Abstract Factory (Fábrica Abstrata), além de entender como eles são implementados em Python.

CAPÍTULO 3

Padrão Factory – construindo fábricas para criar objetos

No capítulo anterior, conhecemos os padrões de projeto Singleton – vimos o que são e como são usados no mundo real – e vimos também a sua implementação em Python. O padrão de projeto Singleton é um dos padrões de criação. Neste capítulo, vamos prosseguir e conhecer outro padrão de criação: o padrão Factory (Fábrica).

O padrão Factory, sem dúvida, é o padrão de projeto mais utilizado. Neste capítulo, entenderemos o conceito de Factory e descreveremos o padrão Simple Factory (Fábrica Simples). Então você conhecerá os padrões Factory Method (Método de Fábrica) e Abstract Factory (Fábrica Abstrata) com um diagrama UML, verá cenários do mundo real e implementações com Python v3.5. Também faremos uma comparação entre Factory Method e Abstract Factory.

Neste capítulo, os seguintes tópicos serão abordados de forma sucinta:

- explicação sobre o padrão de projeto Simple Factory;
- discussão sobre o Factory Method e o Abstract Factory e suas diferenças;
- implementação de cenários do mundo real com código Python;
- discussão das vantagens e desvantagens dos padrões e suas comparações.

Compreendendo o padrão Factory

Em programação orientada a objetos, o termo fábrica (factory) refere-se a uma classe responsável por criar objetos de outros tipos. Geralmente, a classe que atua como uma fábrica tem um objeto e métodos associados a ela. O cliente chama esse método com determinados parâmetros, e os objetos dos tipos desejados são criados e devolvidos ao cliente pela fábrica.

Então, na verdade, a pergunta neste caso é: por que precisamos de uma fábrica quando o cliente pode criar diretamente um objeto? A resposta é que uma fábrica oferece certas vantagens, relacionadas a seguir:

- A primeira vantagem é o baixo acoplamento – a criação de um objeto pode ser independente da implementação da classe.
- O cliente não precisa conhecer a classe que cria o objeto, que, por sua vez, é utilizado pelo cliente. É necessário conhecer apenas a interface, os métodos e os parâmetros que devem ser passados para criar os objetos do tipo desejado. Isso simplifica as implementações para o cliente.
- Adicionar outra classe à fábrica para criar objetos de outro tipo pode ser facilmente implementado sem que o cliente altere o código. No mínimo, o cliente precisará passar apenas mais um parâmetro.
- A fábrica também pode reutilizar objetos existentes. Por outro lado, se o cliente criar objetos diretamente, um novo objeto sempre será criado.

Vamos considerar a situação de uma empresa de manufatura que fabrique brinquedos – carrinhos ou bonecas. Vamos supor que uma máquina da empresa esteja no momento produzindo carrinhos de brinquedo. Então o CEO da empresa sente uma necessidade urgente de produzir bonecas com base na demanda do mercado. Essa situação pede o uso do padrão Factory. Nesse caso, a máquina passa a ser a interface e o CEO é o cliente. O CEO só está interessado no objeto (ou brinquedo) a ser fabricado e conhece a interface – a máquina – capaz de criar o

objeto.

Há três variantes do padrão Factory:

- **Padrão Simple Factory (Factory Simples)** – permite que as interfaces criem objetos sem expor a lógica de sua criação.
- **Padrão Factory Method (Método de Fábrica)** – permite que as interfaces criem objetos, mas adia a decisão para que as subclasses determinem a classe para a criação do objeto.
- **Padrão Abstract Factory (Fábrica Abstrata)** – uma Abstract Factory é uma interface para criar objetos relacionados sem especificar/expor suas classes; o padrão fornece objetos de outra fábrica que, internamente, cria outros objetos.

Padrão Simple Factory

Para alguns, o Simple Factory não é, por si só, um padrão. É mais um conceito que os desenvolvedores devem conhecer antes de saberem mais sobre o Factory Method e o Abstract Factory. O Factory ajuda a criar objetos de tipos diferentes em vez de os objetos serem diretamente instanciados.

Vamos entender isso com a ajuda do diagrama a seguir. Nesse caso, a classe cliente utiliza a classe Factory, que tem o método create_type(). Quando o cliente chama o método create_type() com o parâmetro de tipo, a Factory devolve Product1 ou Product2 conforme os parâmetros passados:

Aprendendo Padrões
de Projeto em Python

Tire proveito da eficácia dos padrões de projeto (design patterns) em Python
para resolver problemas do mundo real em arquitetura e design de software

Um diagrama UML do Simple Factory.

Vamos agora compreender o padrão Simple Factory com a ajuda de um código de exemplo em Python v3.5. No trecho de código a seguir,

criamos um produto Abstract chamado Animal. Animal é uma classe-base abstrata (ABCMeta é a metaclasse especial de Python para criar uma classe Abstract) e tem o método do_say(). Criamos dois produtos (Cat e Dog) a partir da interface Animal e implementamos do_say() com os sons apropriados produzidos por esses animais. ForestFactory é uma fábrica que tem o método make_sound(). De acordo com o tipo de argumento passado pelo cliente, uma instância apropriada de Animal será criada em tempo de execução e o som correto será exibido:

```
from abc import ABCMeta, abstractmethod

class Animal(metaclass = ABCMeta):
    @abstractmethod
    def do_say(self):
        pass

class Dog(Animal):
    def do_say(self):
        print("Bhow Bhow!!")

class Cat(Animal):
    def do_say(self):
        print("Meow Meow!!")


## fábrica forest definida
class ForestFactory(object):
    def make_sound(self, object_type):
        return eval(object_type)().do_say()

## código do cliente
if __name__ == '__main__':
    ff = ForestFactory()
    animal = input("Which animal should make_sound Dog or Cat?")
    ff.make_sound(animal)
```

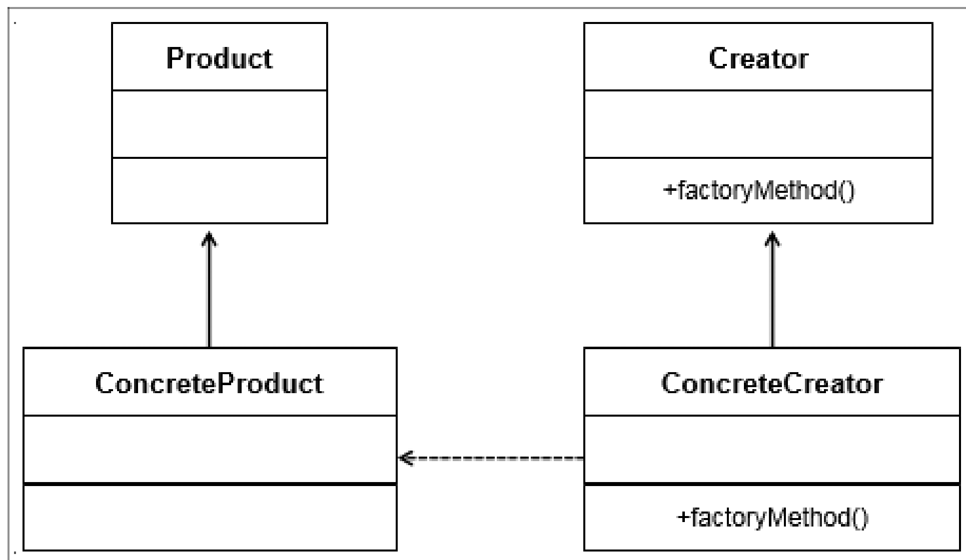
A seguir, apresentamos a saída do trecho de código anterior:

```
Which animal should make_sound Dog or Cat?Cat
Meow Meow!!
```

Padrão Factory Method

Os pontos a seguir nos ajudam a entender o padrão Factory Method:

- Definimos uma interface para criar objetos, mas, em vez de a fábrica ser responsável pela criação dos objetos, a responsabilidade é passada para a subclasse, que decidirá a classe a ser instanciada.
- A criação do Factory Method não é feita por instanciação, mas por herança.
- O Factory Method deixa o design mais personalizável. Ele pode devolver a mesma instância ou subclasse no lugar de um objeto de determinado tipo (como no método de Simple Factory).



Um diagrama UML do Factory Method.

No diagrama UML anterior, temos uma classe abstrata Creator que contém factoryMethod(). O método factoryMethod() é responsável por criar objetos de determinado tipo. A classe ConcreteCreator tem um factoryMethod() que implementa a classe abstrata Creator, e esse método pode mudar o objeto criado em tempo de execução. ConcreteCreator cria ConcreteProduct e garante que o objeto criado implemente a classe Product e forneça uma implementação para todos os métodos da interface Product.

Em suma, factoryMethod() da interface Creator e a classe ConcreteCreator decidem qual subclasse de Product deve ser criada. Assim, o padrão

Factory Method define uma interface para criar um objeto, mas adia a decisão SOBRE qual classe deve ser instanciada, passando-a para as suas subclasses.

Implementando o Factory Method

Vamos considerar um cenário do mundo real para entender a implementação do Factory Method. Imagine que gostaríamos de criar perfis de tipos diferentes em redes sociais como LinkedIn e Facebook para uma pessoa ou uma empresa. Cada um desses perfis deve ter determinadas seções. No LinkedIn, você teria uma seção de patentes que um indivíduo requisitou ou publicações que ele(a) escreveu. No Facebook, você verá seções para um álbum de fotos de uma viagem recente que você fez em um feriado. Além do mais, nesses dois perfis, haveria uma seção comum de informações pessoais. Portanto, resumindo, queremos criar perfis de tipos diferentes com as seções corretas adicionadas ao perfil.

Vamos dar uma olhada na implementação. No exemplo de código a seguir, começaremos definindo a interface `Product`. Criaremos uma classe abstrata `Section` que define como será uma seção. Vamos mantê-la bem simples e fornecer um método abstrato `describe()`.

Agora vamos criar várias classes `ConcreteProduct`: `PersonalSection`, `AlbumSection`, `PatentSection` e `PublicationSection`. Essas classes implementam o método abstrato `describe()` e exibem seus respectivos nomes de seção:

```
from abc import ABCMeta, abstractmethod


class Section(metaclass=ABCMeta):
    @abstractmethod
    def describe(self):
        pass


class PersonalSection(Section):
    def describe(self):
```

```
        print("Personal Section")


class AlbumSection(Section):
    def describe(self):
        print("Album Section")


class PatentSection(Section):
    def describe(self):
        print("Patent Section")


class PublicationSection(Section):
    def describe(self):
        print("Publication Section")
```

Criamos uma classe abstrata Creator chamada Profile. A classe abstrata Profile [Creator] fornece um método de fábrica createProfile(). O método createProfile() deve ser implementado por ConcreteClass para criar realmente os perfis com as seções apropriadas. A classe abstrata Profile não tem conhecimento das seções que cada perfil deve ter. Por exemplo, um perfil do Facebook deve ter informações pessoais e seções para álbum. Portanto vamos deixar a subclasse decidir isso.

Criamos duas classes ConcreteCreator: linkedin e facebook. Cada uma dessas classes implementa o método abstrato createProfile() que realmente cria (instancia) as várias seções (ConcreteProducts) em tempo de execução:

```
class Profile(metaclass=ABCMeta):
    def __init__(self):
        self.sections = []
        self.createProfile()
    @abstractmethod
    def createProfile(self):
        pass
    def getSections(self):
        return self.sections
    def addSections(self, section):
        self.sections.append(section)
```

```
class linkedin(Profile):
    def createProfile(self):
        self.addSections(PersonalSection())
        self.addSections(PatentSection())
        self.addSections(PublicationSection())


class facebook(Profile):
    def createProfile(self):
        self.addSections(PersonalSection())
        self.addSections(AlbumSection())
```

Por fim, implementamos o código do cliente que determina qual classe Creator deve ser instanciada para criar um perfil da opção desejada:

```
if __name__ == '__main__':
    profile_type = input("Which Profile you'd like to create? [LinkedIn or FaceBook]")
    profile = eval(profile_type.lower())()
    print("Creating Profile..", type(profile).__name__)
    print("Profile has sections --", profile.getSections())
```

Se você executar agora o código completo, inicialmente ele pedirá que você forneça o nome do perfil que gostaria de criar. Na imagem de tela a seguir, respondemos Facebook. O código então instancia a classe facebook [ConcreateCreator]. Internamente, isso cria ConcreteProduct(s), isto é, instancia PersonalSection e AlbumSection. Se Linkedin for escolhido, PersonalSection, PatentSection e PublicationSection serão criados.

A seguir, apresentamos a saída do trecho de código anterior:

```
Which Profile you'd like to create? [LinkedIn or FaceBook]FaceBook
Creating Profile.. facebook
Profile has sections -- [<__main__.PersonalSection object at 0x101988b00>, <__main__.AlbumSection object at 0x101988b38>]
```

Vantagens do padrão Factory Method

Como você já conheceu o padrão Factory Method e viu como implementá-lo, vamos analisar as vantagens desse padrão.

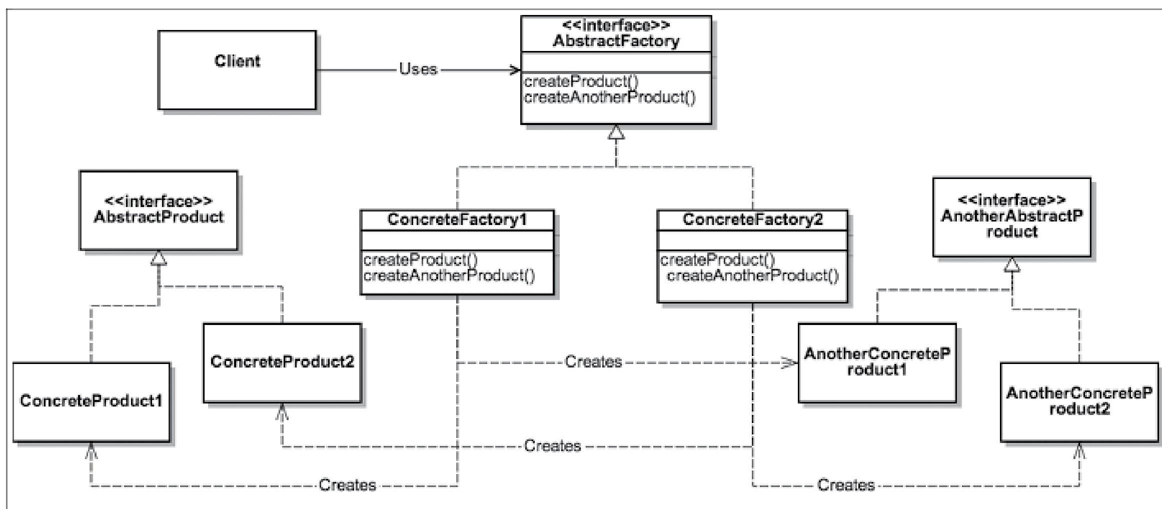
- Ele proporciona muita flexibilidade e deixa o código genérico, não amarrado a determinada classe para instanciação. Dessa maneira, somos dependentes da interface (Product), e não da classe

ConcreteProduct.

- Há baixo acoplamento, pois o código que cria o objeto está separado do código que o utiliza. O cliente não precisa se preocupar com o argumento a ser passado nem com a classe que deve ser instanciada. O acréscimo de novas classes é simples e envolve pouca manutenção.

Padrão Abstract Factory

O objetivo principal do padrão Abstract Factory é fornecer uma interface para criar famílias de objetos relacionados sem especificar a classe concreta. Enquanto o Factory Method adia a criação da instância para as subclasses, o objetivo do método Abstract Factory é criar famílias de objetos relacionados:



Um diagrama UML do padrão Abstract Factory.

Conforme mostra o diagrama, ConcreteFactory1 e ConcreteFactory2 são criados a partir da interface AbstractFactory. Essa interface tem métodos para criar vários produtos.

ConcreteFactory1 e ConcreteFactory2 implementam AbstractFactory e criam instâncias de ConcreteProduct1, ConcreteProduct2, AnotherConcreteProduct1 e AnotherConcreteProduct2.

ConcreteProduct1 e ConcreteProduct2, por sua vez, são criados a partir da interface AbstractProduct, enquanto AnotherConcreteProduct1 e

AnotherConcreteProduct2 são criados a partir da interface AnotherAbstractProduct.

Com efeito, os padrões Abstract Factory garantem que o cliente esteja isolado da criação dos objetos, mas permite que ele utilize os objetos criados. O cliente tem a capacidade de acessar os objetos somente por meio de uma interface. Se os produtos de uma família tiverem de ser usados, o padrão Abstract Factory ajudará o cliente a utilizar os objetos de uma família de cada vez. Por exemplo, se uma aplicação em desenvolvimento tiver de ser independente de plataforma, ela deverá abstrair dependências como sistema operacional e chamadas ao sistema de arquivos, entre outras ações. O padrão Abstract Factory cuida da criação dos serviços necessários para toda a plataforma, para que o cliente não precise criar objetos de plataforma diretamente.

Implementando o padrão Abstract Factory

Considere um exemplo com a sua pizzaria favorita. Ela serve vários tipos de pizza, certo? Espere um minuto, eu sei que você quer pedir uma pizza neste exato momento, mas, por ora, vamos voltar ao exemplo!

Suponha que criamos uma pizzaria em que são servidas pizzas indianas e americanas deliciosas. Para isso, inicialmente criamos uma classe-base abstrata PizzaFactory (AbstractFactory no diagrama UML anterior). A classe PizzaFactory tem dois métodos abstratos, createVegPizza() e createNonVegPizza(), que devem ser implementados por ConcreteFactory. Nesse exemplo, criamos duas fábricas concretas, que são IndianPizzaFactory e USPizzaFactory. Observe a implementação a seguir para as fábricas concretas:

```
from abc import ABCMeta, abstractmethod


class PizzaFactory(metaclass=ABCMeta):

    @abstractmethod
    def createVegPizza(self):
        pass

    @abstractmethod
```

```
    def createNonVegPizza(self):
        pass

class IndianPizzaFactory(PizzaFactory):

    def createVegPizza(self):
        return DeluxVeggiePizza()

    def createNonVegPizza(self):
        return ChickenPizza()


class USPizzaFactory(PizzaFactory):

    def createVegPizza(self):
        return MexicanVegPizza()

    def createNonVegPizza(self):
        return HamPizza()
```

Vamos seguir em frente e definir AbstractProducts. No código a seguir, criamos duas classes abstratas, VegPizza e NonVegPizza (AbstractProduct e AnotherAbstractProduct no diagrama UML anterior). Individualmente, elas têm um método definido: prepare() e serve().

O raciocínio, nesse caso, é que as pizzas vegetarianas são preparadas com uma massa, legumes e temperos apropriados, enquanto as pizzas não vegetarianas são servidas com ingredientes não vegetarianos sobre pizzas vegetarianas.

Então definimos ConcreteProducts para cada um dos AbstractProducts. Nesse caso, criamos DeluxVeggiePizza e MexicanVegPizza e implementamos o método prepare(). ConcreteProducts1 e ConcreteProducts2 representam essas classes no diagrama UML.

A seguir, definimos ChickenPizza e HamPizza e implementamos o método serve() – elas representam AnotherConcreteProducts1 e AnotherConcreteProducts2:

```
class VegPizza(metaclass=ABCMeta):
    @abstractmethod
    def prepare(self, VegPizza):
```

```
        pass

class NonVegPizza(metaclass=ABCMeta):
    @abstractmethod
    def serve(self, VegPizza):
        pass

class DeluxVeggiePizza(VegPizza):
    def prepare(self):
        print("Prepare ", type(self).__name__)

class ChickenPizza(NonVegPizza):
    def serve(self, VegPizza):
        print(type(self).__name__, " is served with Chicken on ",
            type(VegPizza).__name__)

class MexicanVegPizza(VegPizza):
    def prepare(self):
        print("Prepare ", type(self).__name__)

class HamPizza(NonVegPizza):
    def serve(self, VegPizza):
        print(type(self).__name__, " is served with Ham on ",
            type(VegPizza).__name__)
```

Quando um usuário final chegar a PizzaStore e pedir uma pizza americana não vegetariana, USPizzaFactory será responsável por preparar a pizza vegetariana como base e servir a pizza não vegetariana com presunto em cima!

```
class PizzaStore:
    def __init__(self):
        pass
    def makePizzas(self):
        for factory in [IndianPizzaFactory(), USPizzaFactory()]:
            self.factory = factory
            self.NonVegPizza = self.factory.createNonVegPizza()
            self.VegPizza = self.factory.createVegPizza()
            self.VegPizza.prepare()
            self.NonVegPizza.serve(self.VegPizza)
```

```
pizza = PizzaStore()
pizza.makePizzas()
```

A seguir, apresentamos a saída do trecho de código anterior:

```
Prepare  DeluxVeggiePizza
ChickenPizza  is served with Chicken on  DeluxVeggiePizza
Prepare  MexicanVegPizza
HamPizza  is served with Ham on  MexicanVegPizza
```

Comparação entre Factory Method e Abstract Factory

Agora que já conhecemos Factory Method e Abstract Factory, vamos ver uma comparação entre eles:

Factory Method	Abstract Factory
Expõe um método ao cliente para criar os objetos.	O método Abstract Factory contém um ou mais métodos de fábrica para criar uma família de objetos relacionados.
Usa herança e subclasses para definir o objeto a ser criado.	Usa composição para delegar a responsabilidade de criar objetos de outra classe.
O Factory Method é usado para criar um produto.	O método Abstract Factory diz respeito a criar famílias de produtos relacionados.

Resumo

Neste capítulo, conhecemos o padrão de projeto Factory e o contexto em que ele é usado. Compreendemos o básico sobre o Factory e como ele é efetivamente usado em arquitetura de software.

Vimos o Simple Factory, em que uma instância apropriada é criada em tempo de execução com base no tipo do argumento passado pelo cliente.

Também discutimos o padrão Factory Method, que é uma variação do Simple Factory. Nesse padrão, definimos uma interface para criar objetos, mas a criação desses objetos é adiada para a subclasse.

Prosseguimos explorando o método Abstract Factory, que oferece uma interface para criar famílias de objetos relacionados sem especificar a classe concreta.

Também fizemos uma implementação Python do mundo real para todos os três padrões e comparamos o Factory Method com o Abstract Factory.

No final deste capítulo, estamos prontos para dar o próximo passo e estudar outros tipos de padrão, portanto, fique sintonizado.

CAPÍTULO 4

Padrão Façade – sendo adaptável com o Façade

No capítulo anterior, conhecemos o padrão de projeto Factory. Discutimos três variações – os padrões Simple Factory, Factory Method e Abstract Factory. Também vimos como cada um deles é usado no mundo real e observamos implementações em Python. Além disso, comparamos os padrões Factory Method e Abstract Factory, e listamos seus prós e contras. Como já sabemos agora, tanto o padrão de projeto Factory quanto o padrão Singleton (*Capítulo 2, Padrão de projeto Singleton*) são classificados como padrões de projeto de criação.

Neste capítulo, vamos prosseguir e conhecer outro tipo de padrão de projeto: o padrão estrutural. Seremos apresentados ao padrão de projeto Façade (Fachada) e discutiremos como ele é usado no desenvolvimento de aplicações de software. Trabalharemos com um caso de uso de exemplo e faremos a implementação com Python v3.5.

Os seguintes tópicos serão abordados de forma sucinta:

- uma introdução aos padrões de projeto estruturais;
- uma explicação do padrão de projeto Façade com um diagrama UML;
- um caso de uso do mundo real com implementação em Python v3.5;
- o padrão Façade e o princípio do conhecimento mínimo.

Compreendendo os padrões de projeto estruturais

Os conceitos a seguir nos ajudarão a entender melhor os padrões estruturais.

- Os padrões estruturais descrevem como objetos e classes podem ser combinados para compor estruturas maiores.
- Podemos pensar nos padrões estruturais como padrões de projeto que facilitam o design por meio da identificação de maneiras mais simples de perceber ou demonstrar relacionamentos entre entidades. As entidades referem-se a objetos ou classes no mundo da orientação a objetos.
- Enquanto os padrões Classe descrevem abstrações com a ajuda de herança e oferecem uma interface de programação mais conveniente, o padrão Objeto descreve como os objetos podem ser associados e compostos para formar objetos maiores. Os padrões estruturais são uma combinação dos padrões Classe e Objeto.

A seguir, apresentamos alguns exemplos de diferentes padrões de projeto estruturais. Você perceberá como cada um deles envolve interação entre objetos ou classes para alcançar objetivos de nível mais alto de design e de arquitetura.

Alguns exemplos de padrões de projeto estruturais são:

- **Padrão Adapter (Adaptador)** – adapta uma interface a outra para que as expectativas do cliente sejam atendidas; procura combinar interfaces de classes diferentes conforme as necessidades do cliente.
- **Padrão Bridge (Ponte)** – desacopla a interface de um objeto de sua implementação para que ambos possam trabalhar de forma independente.
- **Padrão Decorator (Decorador)** – define responsabilidades adicionais para um objeto em tempo de execução ou dinamicamente; podemos adicionar determinados atributos aos objetos com uma interface.

Há outros padrões estruturais que você conhecerá neste livro. Então,

vamos começar abordando inicialmente o padrão de projeto Façade.

Compreendendo o padrão de projeto Façade

A fachada (façade) geralmente refere-se à face de uma construção, em especial uma face atraente. Também pode estar relacionada a um comportamento ou uma aparência que dê uma falsa ideia dos verdadeiros sentimentos ou da situação de alguém. Quando passam diante de uma fachada, as pessoas podem apreciar a face externa, mas não têm conhecimento das complexidades da estrutura interna. É assim que um padrão de fachada é usado. O Façade oculta as complexidades do sistema interno e oferece uma interface ao cliente para que este possa acessar o sistema de forma bem simplificada.

Considere o exemplo de um atendente de loja. Quando você como cliente vai a uma loja para comprar determinados itens, você não tem conhecimento do layout dessa loja. Geralmente, você chama o atendente, que conhece bem o sistema da loja. Com base em suas solicitações, o atendente escolhe alguns itens e os entrega a você. Não é simples? O cliente não precisa saber como é a loja, e ele(a) tem o serviço feito por meio de uma interface simples, o atendente.

O padrão de projeto Façade faz essencialmente o seguinte:

- Oferece uma interface unificada para um conjunto de interfaces em um subsistema e define uma interface de alto nível que ajuda o cliente a usar o subsistema de forma fácil.
- O Façade procura fazer a representação de um subsistema complexo com um único objeto de interface. Ela não **encapsula** o subsistema, mas, na verdade, combina os subsistemas subjacentes.
- Promove o desacoplamento da implementação com vários clientes.

Um diagrama de classes UML

Discutiremos agora o padrão Façade com o auxílio do diagrama UML, mostrado a seguir:

Aprendendo Padrões de Projeto em Python

Tire proveito da eficácia dos padrões de projeto (design patterns) em Python para resolver problemas do mundo real em arquitetura e design de software

Como podemos observar no diagrama UML, você perceberá que há três participantes principais nesse padrão:

- **Façade** – a principal responsabilidade de uma fachada é englobar um grupo complexo de subsistemas de modo que a fachada possa oferecer uma aparência agradável ao mundo externo.
- **Sistema** – representa um conjunto de subsistemas variados que compõem o sistema como um todo, que é difícil de visualizar ou com o qual é complicado trabalhar.
- **Cliente** – o cliente interage com o Façade para que ele possa se comunicar facilmente com o subsistema e ter uma tarefa concluída; ele não precisa se preocupar com a natureza complexa do sistema.

Agora você conhecerá um pouco mais os três participantes principais do ponto de vista da estrutura de dados.

Façade

Os conceitos a seguir darão uma ideia melhor do Façade.

- É uma interface que sabe quais subsistemas são responsáveis por uma requisição.
- Delega as requisições do cliente para os objetos de subsistema apropriados usando composição.

Por exemplo, se o cliente quer que uma tarefa seja executada, ele não

precisa acessar os subsistemas individuais, mas pode simplesmente entrar em contato com a interface (Façade), que fará o trabalho ser executado.

Sistema

No mundo do Façade, o Sistema é uma entidade que faz o seguinte:

- implementa funcionalidades de subsistema e é representado por uma classe; o ideal é que um Sistema seja representado por um grupo de classes responsáveis por diferentes operações.
- cuida da tarefa atribuída ao objeto Façade, mas não tem nenhum conhecimento da fachada e não mantém nenhuma referência a ela.

Por exemplo, quando o cliente pede determinado serviço ao Façade, este escolhe o subsistema correto que executa o serviço de acordo com o seu tipo.

Cliente

Eis o modo como podemos descrever o cliente:

- O cliente é uma classe que instancia o Façade.
- Ele faz requisições ao Façade para que os subsistemas executem uma tarefa.

Implementando o padrão Façade no mundo real

Para demonstrar as aplicações do padrão Façade, vamos considerar uma situação de exemplo pela qual poderíamos passar em nossa vida.

Suponha que haja um casamento em sua família e que você seja responsável por toda a organização. Opa! Você tem um trabalho árduo em mãos. Você deve reservar um hotel ou um local apropriado para casamentos, conversar com alguém responsável por um bufê para organizar o cardápio, precisará de uma florista para todas as decorações

e, por fim, terá que cuidar da programação musical esperada para o evento.

No passado, você faria tudo isso sozinho, por exemplo, conversando com as pessoas relevantes, fazendo a coordenação entre elas e negociando preços. Atualmente, porém, a vida é mais simples. Você conversa com um(a) organizador(a) de eventos que cuida disso para você. Ele(a) se responsabiliza por conversar individualmente com os fornecedores de serviços e faz o melhor negócio possível.

Colocando essa situação sob a perspectiva do padrão Façade, temos:

- **Cliente** – é você, que precisa que toda a organização do casamento esteja concluída a tempo antes do casamento; as preparações precisam ser de primeira e os convidados deverão amar as celebrações.
- **Façade** – é o(a) organizador(a) de eventos, responsável por conversar com todas as pessoas que precisam trabalhar nas organizações específicas, por exemplo, de cardápio e de decorações florais, entre outras.
- **Subsistemas** – representam os sistemas que oferecem os serviços, como de bufê, gerenciamento de hotel e decorações florais.

Vamos desenvolver uma aplicação em Python v3.5 e implementar este caso de uso. Começaremos pelo cliente. É você! Lembre-se de que foi você quem assumiu a responsabilidade de garantir que a organização do casamento fosse feita e que o evento corresse bem!

Vamos prosseguir agora e conversar sobre a classe Façade. Conforme discutimos antes, a classe Façade simplifica a interface para o cliente. Nesse caso, `EventManager` atua como uma fachada e simplifica o trabalho para você (`You`). O Façade conversa com os subsistemas e faz todas as reservas e preparações para o casamento em seu nome. Eis o código Python para a classe `EventManager`:

```
class EventManager(object):
```

```
    def __init__(self):
        print("Event Manager:: Let me talk to the folks\n")

    def arrange(self):
        self.hotelier = Hotelier()
        self.hotelier.bookHotel()

        self.florist = Florist()
        self.florist.setFlowerRequirements()

        self.caterer = Caterer()
        self.caterer.setCuisine()

        self.musician = Musician()
        self.musician.setMusicType()
```

Agora que já descrevemos o Façade e o cliente, vamos explorar os subsistemas. Desenvolvemos as seguintes classes para esse cenário:

- Hotelier – que será usado para reserva de hotel; ele tem um método para verificar se o hotel está livre naquele data (__isAvailable).
- A classe Florist é responsável pelas decorações florais. Florist tem o método setFlowerRequirements(), que deve ser usado para definir as expectativas sobre o tipo de flor necessário para a decoração do casamento.
- A classe Caterer é usada para cuidar do bufê e é responsável pela preparação da comida. Caterer expõe o método setCuisine() para decidir o tipo de cardápio que será servido no casamento.
- A classe Musician foi projetada para a programação musical do casamento. Ela utiliza o método setMusicType() para decifrar os requisitos musicais do evento.

Agora vamos dar uma olhada no objeto Hotelier, seguido do objeto Florist e de seus métodos:

```
class Hotelier(object):
    def __init__(self):
        print("Arranging the Hotel for Marriage? --")
```

```
    def __isAvailable(self):
        print("Is the Hotel free for the event on given day?")
        return True

    def bookHotel(self):
        if self.__isAvailable():
            print("Registered the Booking\n\n")


class Florist(object):
    def __init__(self):
        print("Flower Decorations for the Event? --")

    def setFlowerRequirements(self):
        print("Carnations, Roses and Lilies would be used for Decorations\n\n")


class Caterer(object):
    def __init__(self):
        print("Food Arrangements for the Event --")

    def setCuisine(self):
        print("Chinese & Continental Cuisine to be served\n\n")


class Musician(object):
    def __init__(self):
        print("Musical Arrangements for the Marriage --")

    def setMusicType(self):
        print("Jazz and Classical will be played\n\n")
```

No entanto, você está sendo inteligente aqui e está passando a responsabilidade para o organizador de eventos, certo? Vamos dar uma olhada agora na classe `You`. Nesse exemplo, criamos um objeto da classe `EventManager` para que o organizador possa trabalhar com o pessoal relevante na organização do casamento enquanto você relaxa.

```
class You(object):
    def __init__(self):
        print("You:: Whoa! Marriage Arrangements??!!!")
```

```
    def askEventManager(self):
        print("You:: Let's Contact the Event Manager\n\n")
        em = EventManager()
        em.arrange()
    def __del__(self):
        print("You:: Thanks to Event Manager, all preparations done! Phew!")

you = You()
you.askEventManager()
```

Eis a saída do trecho de código anterior:

```
You:: Whoa! Marriage Arrangements??!!!
You:: Let's Contact the Event Manager


Event Manager:: Let me talk to the folks

Arranging the Hotel for Marriage? --
Is the Hotel free for the event on given day?
Registered the Booking..


Flower Decorations for the Event? --
Carnations, Roses and Lilies would be used for Decorations


Food Arrangements for the Event --
Chinese & Continental Cuisine to be served


Musical Arrangements for the Marriage --
Jazz and Classical will be played


You:: Thanks to Event Manager, all preparations done! Phew!
```

Podemos relacionar o padrão Façade com o cenário do mundo real da seguinte maneira:

- A classe EventManager é o Façade que simplifica a interface para You.
- EventManager usa composição para criar objetos de subsistemas como Hotelier, Caterer e outros.

Princípio do conhecimento mínimo

Como você aprendeu nas partes iniciais do capítulo, o Façade oferece um sistema unificado que facilita o uso de subsistemas. Ele também desacopla o cliente do subsistema de componentes. O princípio de design empregado por trás do padrão Façade é o *princípio do conhecimento mínimo*.

O princípio do conhecimento mínimo nos orienta no sentido de reduzir as interações entre os objetos a apenas alguns *amigos* que sejam próximos a você. Em termos reais, isso significa o seguinte:

- Quando fizermos o design de um sistema, para todo objeto criado, devemos observar o número de classes com que essa classe interage e o modo como a interação ocorre.
- Seguindo o princípio, certifique-se de evitar situações em que haja muitas classes criadas que estejam altamente acopladas umas às outras.
- Se houver muitas dependências entre as classes, o sistema será difícil de manter. Qualquer mudança em uma parte do sistema poderá resultar em alterações não intencionais em outras partes, o que significa que o sistema estará exposto a regressões, e isso deve ser evitado.

Perguntas frequentes

P1. O que é a Lei de Demeter e como ela está relacionada ao padrão Façade?

R: A Lei de Demeter é uma diretriz de design que diz respeito ao seguinte:

1. Cada unidade deve ter um conhecimento apenas limitado de outras unidades do sistema.
2. Uma unidade deve se comunicar apenas com seus amigos.
3. Uma unidade não deve conhecer os detalhes internos do objeto que ela manipula.

O princípio do conhecimento mínimo e a Lei de Demeter são iguais, e ambos apontam para a filosofia do *baixo acoplamento.* O princípio do conhecimento mínimo é adequado ao caso de uso do padrão Façade; o nome é intuitivo, e o princípio atua como uma diretriz, pois não é rigoroso e será útil somente quando for necessário.

P2. Pode haver vários Façades para um subsistema?

R: Sim, pode-se implementar mais de uma fachada para um grupo de componentes do subsistema.

P3. Quais são as desvantagens do princípio do conhecimento mínimo?

R: Um Façade oferece uma interface simplificada para os clientes interagirem com subsistemas. No intuito de fornecer uma interface simplificada, uma aplicação poderá ter várias interfaces desnecessárias que aumentarão a complexidade do sistema e reduzirão o desempenho em tempo de execução.

P4. O cliente pode acessar os subsistemas de forma independente?

R: Sim; na verdade, o padrão Façade oferece interfaces simplificadas para que o cliente não precise se preocupar com a complexidade dos subsistemas.

P5. O Façade acrescenta alguma funcionalidade própria?

R: Um Façade pode adicionar seu "raciocínio" aos subsistemas, por exemplo, garantir que a ordem da invocação dos subsistemas possa ser decidida pelo Façade.

Resumo

Iniciamos o capítulo descrevendo os padrões de projeto estruturais. Em seguida, conhecemos o padrão de projeto Façade e o contexto em que ele é usado. Vimos o básico sobre o Façade e como ele é efetivamente usado em arquiteturas de software. Além disso, vimos como os padrões de projeto Façade criam uma interface simplificada para os clientes usarem. Eles reduzem a complexidade dos subsistemas para que o cliente possa se beneficiar.

O Façade não encapsula o subsistema, e o cliente tem a liberdade de acessá-lo sem intermédio do Façade. Também conhecemos o padrão com um diagrama UML e vimos um exemplo de implementação de código com Python v3.5. Compreendemos o princípio do conhecimento mínimo e como a sua filosofia governa os padrões de projeto Façade.

Também incluímos uma seção de Perguntas Frequentes que ajudará você a ter mais ideias sobre o padrão e suas possíveis desvantagens. Agora estamos preparados para conhecer outros padrões estruturais nos próximos capítulos.

CAPÍTULO 5

Padrão Proxy – controlando o acesso a objetos

No capítulo anterior, começamos com uma rápida introdução aos padrões estruturais e prosseguimos discutindo o padrão de projeto Façade (Fachada). Entendemos o conceito de Façade com um diagrama UML e vimos também como ele é aplicado no mundo real com a ajuda de implementações em Python. Além disso, conhecemos as vantagens e as desvantagens do padrão Façade na seção de Perguntas Frequentes.

Neste capítulo, vamos avançar mais um passo e trabalhar com o padrão Proxy (Procurador), que se enquadra na categoria dos padrões de projeto estruturais. Seremos apresentados ao padrão Proxy como um conceito, prosseguiremos com uma discussão sobre o padrão de projeto e veremos como ele é usado no desenvolvimento de aplicações de software. Trabalharemos com um caso de uso de exemplo e faremos a sua implementação com Python v3.5.

Neste capítulo, os seguintes tópicos serão abordados de forma sucinta:

- uma introdução ao proxy e aos padrões de projeto Proxy;
- um diagrama UML para o padrão Proxy;
- variações dos padrões Proxy;
- um caso de uso do mundo real com implementação em Python v3.5;
- vantagens do padrão Proxy;
- comparação entre os padrões Façade e Proxy;

- perguntas frequentes.

Compreendendo o padrão de projeto Proxy

Em termos gerais, o Proxy é um sistema que serve de intermediário entre o solicitante (seeker) e o provedor (provider). O solicitante é quem faz a requisição, e o provedor entrega os recursos em resposta à requisição. No mundo da web, podemos relacionar isso a um servidor proxy. Quando os clientes (usuários da World Wide Web) fazem uma requisição ao site, inicialmente se conectam com um servidor proxy solicitando recursos; por exemplo, uma página web. O servidor proxy internamente avalia essa requisição, envia para um servidor apropriado e recebe a resposta, que é então entregue ao cliente. Assim, um servidor proxy encapsula requisições, proporciona privacidade e funciona bem em arquiteturas distribuídas.

No contexto dos padrões de projeto, `Proxy` é uma classe que atua como uma interface para objetos reais. Os objetos podem ser de vários tipos, por exemplo, conexões de rede, objetos grandes em memória e arquivos, entre outros. Em suma, `Proxy` é um wrapper ou um objeto agente que encapsula o objeto que está realmente servindo. O Proxy pode oferecer funcionalidades adicionais ao objeto que ele encapsula sem alterar o código do objeto. O principal propósito do padrão Proxy é oferecer um substituto ou um placeholder para outro objeto a fim de controlar o acesso a um objeto real.

O padrão Proxy é usado em vários cenários, por exemplo, nestes:

- Ele representa um sistema complexo de maneira mais simples. Por exemplo, um sistema que envolva vários cálculos ou procedimentos complexos deve ter uma interface mais simples que possa atuar como um proxy em benefício do cliente.
- O padrão oferece mais segurança aos objetos reais. Em muitos casos, o cliente não tem permissão para acessar o objeto real diretamente.

Isso ocorre porque o objeto real pode ser comprometido como resultado de atividades maliciosas. Desse modo, os proxies atuam como um escudo contra intenções maliciosas e protegem o objeto real.

- O Proxy oferece uma interface local para objetos remotos em servidores diferentes. Um exemplo claro desse caso está em sistemas distribuídos nos quais o cliente queira executar determinados comandos no sistema remoto, mas pode não ter permissões diretas para fazer isso acontecer. Assim, ele entra em contato com um objeto local (proxy) com a requisição, que é então executada pelo proxy no computador remoto.
- O padrão fornece um tratamento leve para um objeto com consumo mais elevado de memória. Às vezes, você pode não querer carregar os objetos principais, a menos que eles sejam realmente necessários. Isso porque os objetos reais são realmente pesados e podem precisar de muitos recursos. Um exemplo clássico são as fotos de perfis dos usuários em um site. Será muito melhor mostrar imagens menores nos perfis na visão de lista, mas é claro que você precisará carregar a imagem propriamente dita para exibi-la na visão detalhada do perfil do usuário.

Vamos entender o padrão com um exemplo simples. Considere o exemplo de um ator (Actor) e seu agente (Agent). Quando as produtoras quiserem abordar um Actor para um filme, geralmente conversarão com o Agent, e não diretamente com o Actor. Conforme a agenda do Actor e outros compromissos, o Agent informa à produtora da disponibilidade e do interesse em trabalhar no filme. Nesse cenário, em vez de as produtoras conversarem diretamente com o Actor, o Agent atua como um Proxy que cuida de toda a agenda e dos pagamentos do Actor.

O código Python a seguir implementa esse cenário em que Agent é o Proxy. O objeto Agent é usado para descobrir se Actor está ocupado. Se Actor estiver ocupado, o método Actor().occupied() será chamado; se Actor não estiver ocupado, o método Actor().available() será retornado.

```
class Actor(object):
    def __init__(self):
        self.isBusy = False

    def occupied(self):
        self.isBusy = True
        print(type(self).__name__ , "is occupied with current movie")

    def available(self):
        self.isBusy = False
        print(type(self).__name__ , "is free for the movie")

    def getStatus(self):
        return self.isBusy

class Agent(object):
    def __init__(self):
        self.principal = None

    def work(self):
        self.actor = Actor()
        if self.actor.getStatus():
            self.actor.occupied()
        else:
            self.actor.available()


if __name__ == '__main__':
    r = Agent()
    r.work()
```

O padrão de projeto Proxy faz essencialmente o seguinte:

- fornece um substituto para outro objeto para que você possa controlar o acesso ao objeto original;
- é usado como uma camada ou interface que oferece suporte a acesso distribuído;
- acrescenta delegação e protege o componente real de consequências indesejadas.

Um diagrama de classes UML para o padrão Proxy

Discutiremos agora o padrão Proxy com o auxílio do diagrama UML mostrado a seguir. Conforme discutimos no parágrafo anterior, o padrão Proxy tem três atores principais: a produtora, o Agent e o Actor. Vamos colocá-los em um diagrama UML e ver como é a aparência dessas classes.

Aprendendo Padrões de Projeto em Python

Tire proveito da eficácia dos padrões de projeto (design patterns) em Python para resolver problemas do mundo real em arquitetura e design de software

Como podemos observar no diagrama UML, você perceberá que há três participantes principais nesse padrão:

- Proxy – mantém uma referência que lhe permite acessar o objeto real; oferece uma interface idêntica a Subject para que o Proxy possa substituir o objeto real. Os Proxies também são responsáveis por criar e apagar o RealSubject.
- Subject – oferece uma representação tanto para RealSubject quanto para Proxy. Como Proxy e RealSubject implementam Subject, Proxy pode ser usado em qualquer lugar que RealSubject seja esperado.
- RealSubject – define o objeto real representado pelo Proxy.

Do ponto de vista da estrutura de dados, o diagrama UML pode ser representado assim:

- Proxy – é uma classe que controla o acesso à classe RealSubject. Ela cuida das requisições do cliente e é responsável por criar ou remover o RealSubject.

- Subject/RealSubject – Subject é uma interface que define como devem ser RealSubject e Proxy. RealSubject é uma implementação concreta da interface Subject. Ela oferece a verdadeira funcionalidade que é então usada pelo cliente.
- Client – acessa a classe Proxy para que a tarefa seja executada. Internamente, a classe Proxy controla o acesso a RealSubject e direciona a tarefa solicitada pelo Client.

Compreendendo os diferentes tipos de proxies

Existem várias situações comuns em que os proxies são usados. Descrevemos algumas delas no início deste capítulo. Com base no modo como os Proxies são usados, podemos classificá-los como proxy virtual, proxy remoto, proxy de proteção e proxy inteligente. Vamos conhecê-los um pouco melhor nesta seção.

Proxy virtual

Nesta seção, você conhecerá o proxy virtual em detalhes. Ele é um placeholder para objetos que são muito pesados para instanciar. Por exemplo, se você quiser carregar uma imagem grande em seu site, essa requisição exigirá bastante tempo para a carga. Geralmente, os desenvolvedores criarão um ícone de placeholder na página web sugerindo que há uma imagem. No entanto a imagem só será carregada quando o usuário realmente clicar no ícone, evitando assim o custo de carregar uma imagem pesada em memória. Desse modo, nos proxies virtuais, o objeto real é criado quando o cliente faz sua primeira requisição ou seu primeiro acesso ao objeto.

Proxy remoto

Um proxy remoto pode ser definido nos termos a seguir. Ele oferece uma representação local de um objeto real que está em um servidor remoto ou em um espaço de endereçamento diferente. Por exemplo, você

quer implementar um sistema de monitoração para a sua aplicação que tem vários servidores web, servidores de banco de dados, servidores de tarefas Celery e servidores de caching, entre outros. Se quisermos monitorar a utilização de CPU e de disco nesses servidores, precisaremos de um objeto que esteja disponível no contexto em que a aplicação de monitoração é executada, mas que possa executar comandos remotos para obter os valores dos parâmetros. Em casos assim, ter um objeto de proxy remoto que seja uma representação local do objeto remoto ajudaria.

Proxy de proteção

Você entenderá melhor o proxy de proteção com as considerações a seguir. Esse proxy controla o acesso às partes sensíveis de RealSubject. Por exemplo, no mundo atual dos sistemas distribuídos, as aplicações web têm vários serviços que operam em conjunto para oferecer funcionalidades. Em sistemas como esses, um serviço de autenticação atua como um servidor de proxy de proteção, responsável pela autenticação e pela autorização. Nesse caso, o Proxy internamente ajuda na proteção das funcionalidades essenciais do site verificando agentes não reconhecidos ou não autorizados. Assim, o objeto substituto verifica se quem faz a chamada tem as permissões de acesso necessárias para encaminhar a requisição.

Proxy inteligente

Os proxies inteligentes interpõem ações adicionais quando um objeto é acessado. Por exemplo, considere que haja um componente nuclear no sistema que armazene estados em um local centralizado. Geralmente, um componente como esse é chamado por vários serviços diferentes para que eles possam concluir suas tarefas, e pode resultar em problemas com recursos compartilhados. Em vez de os serviços chamarem diretamente o componente nuclear, um proxy inteligente é embutido e verifica se o objeto real está travado antes de ser acessado a fim de garantir que nenhum outro objeto possa alterá-lo.

Padrão Proxy no mundo real

Vamos considerar um caso de uso de pagamento a fim de demonstrar um cenário do mundo real para o padrão Proxy. Suponha que você vá fazer compras em um shopping e goste de uma bela camisa de brim. Você gostaria de comprar a camisa, mas não tem dinheiro vivo suficiente para isso.

No passado, você iria até um caixa eletrônico, sacaria o seu dinheiro e então voltaria para o shopping para fazer o pagamento. Antes ainda, você tinha um talão de cheques com o qual teria de ir até o banco, sacaria o dinheiro e então voltaria para pagar suas compras.

Graças aos bancos, atualmente temos algo chamado cartão de débito. Portanto, agora, quando quiser comprar algo, você poderá apresentar o seu cartão de débito ao comerciante. Quando fornecer os detalhes de seu cartão, o dinheiro será creditado na conta do comerciante para pagar suas compras.

Vamos desenvolver uma aplicação em Python v3.5 e implementar o caso de uso anterior. Começaremos primeiro pelo cliente. Você foi ao shopping center e gostaria agora de comprar uma bela camisa de brim. Vamos ver como o código de Client será implementado.

- O seu comportamento é representado pela classe You – o cliente.
- Para comprar a camisa, o método make_payment() é fornecido pela classe.
- O método especial __init__() chama o Proxy e o instancia.
- O método make_payment() chama o método do Proxy internamente para fazer o pagamento.
- O método __del__() retorna caso o pagamento seja bem-sucedido.

Desse modo, o código de exemplo é:

```
class You:
    def __init__(self):
        print("You:: Lets buy the Denim shirt")
        self.debitCard = DebitCard()
```

```
        self.isPurchased = None

    def make_payment(self):
        self.isPurchased = self.debitCard.do_pay()

    def __del__(self):
        if self.isPurchased:
            print("You:: Wow! Denim shirt is Mine :-)")
        else:
            print("You:: I should earn more :(")

you = You()
you.make_payment()
```

Vamos discutir agora a classe Subject. Como sabemos, a classe Subject é uma interface implementada pelo Proxy e por RealSubject.

- Nesse exemplo, essa classe é Payment. É uma classe-base abstrata e representa uma interface.
- Payment contém o método do_pay() que deve ser implementado por Proxy e por RealSubject.

Vamos ver esses métodos em ação no código a seguir:

```
from abc import ABCMeta, abstractmethod

class Payment(metaclass=ABCMeta):

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

Desenvolvemos também a classe Bank, que representa RealSubject neste cenário:

- Bank fará realmente o pagamento, passando o valor de sua conta para a conta do comerciante.
- Bank tem vários métodos para processar o pagamento. O método setCard() é usado pelo Proxy para enviar os detalhes do cartão de débito ao banco.
- O método __getAccount() é um método privado de Bank, usado para obter

os detalhes da conta do dono do cartão de débito. Para simplificar, fizemos o número do cartão de débito ser igual ao número da conta.

- Bank também tem o método __hasFunds() para verificar se o dono da conta tem saldo suficiente para pagar pela camisa.
- O método do_pay() implementado pela classe Bank (da interface Payment), na verdade, é responsável por fazer o pagamento ao comerciante se houver saldo disponível:

```
class Bank(Payment):

    def __init__(self):
        self.card = None
        self.account = None

    def __getAccount(self):
        self.account = self.card   # Supõe que o número do cartão
                                   # é o número da conta
        return self.account

    def __hasFunds(self):
        print("Bank:: Checking if Account", self.__getAccount(),
            "has enough funds")
        return True

    def setCard(self, card):
        self.card = card

    def do_pay(self):
        if self.__hasFunds():
            print("Bank:: Paying the merchant")
            return True
        else:
            print("Bank:: Sorry, not enough funds!")
            return False
```

Vamos agora entender a última parte, que é o Proxy.

- A classe DebitCard é o Proxy, nesse caso. Quando você (You) quiser fazer um pagamento, o método do_pay() deve ser chamado. Isso é porque você (You) não quer ir até o banco para sacar o dinheiro e pagar o

comerciante.

- A classe DebitCard atua como um substituto para o RealSubject, que é Bank.
- O método payWithCard() internamente controla a criação do objeto RealSubject, que é a classe Bank, e apresenta os detalhes do cartão para Bank.
- Bank executa as verificações internas na conta e faz o pagamento, conforme descrito no trecho de código anterior:

```
class DebitCard(Payment):

    def __init__(self):
        self.bank = Bank()

    def do_pay(self):
        card = input("Proxy:: Punch in Card Number: ")
        self.bank.setCard(card)
        return self.bank.do_pay()
```

Em um caso positivo, quando houver saldo suficiente, a saída será:

```
You:: Lets buy the Denim shirt
Proxy:: Punch in Card Number: 23-2134-222
Bank:: Checking if Account 23-2134-222 has enough funds
Bank:: Paying the merchant
You:: Wow! Denim shirt is Mine :-)
```

Em um caso negativo – saldo insuficiente – a saída será:

```
You:: Lets buy the Denim shirt
Proxy:: Punch in Card Number: 23-2134-222
Bank:: Checking if Account 23-2134-222 has enough funds
Bank:: Sorry, not enough funds!
You:: I should earn more :(
```

Vantagens do padrão Proxy

Como já vimos o funcionamento do padrão Proxy no mundo real, vamos conferir as vantagens desse padrão.

- Os Proxies podem ajudar a melhorar o desempenho da aplicação ao

fazer caching de objetos pesados ou, geralmente, de objetos acessados com frequência.

- Os Proxies também autorizam o acesso a RealSubject; portanto esse padrão auxilia na delegação somente se as permissões estiverem corretas.
- Os proxies remotos também facilitam a interação com servidores remotos, que podem funcionar como conexões de rede e de bancos de dados, além de poderem ser usados para monitorar sistemas.

Comparação entre os padrões Façade e Proxy

Tanto o padrão Façade quanto o padrão Proxy são padrões de projeto estruturais. Eles são semelhantes no sentido que ambos têm um objeto proxy/fachada na frente de objetos reais. As diferenças, na verdade, estão na intenção ou no propósito dos padrões, como mostra a tabela a seguir.

Padrão Proxy	Padrão Façade
Oferece um substituto ou um placeholder para outro objeto a fim de controlar o acesso a ele.	Oferece uma interface para subsistemas grandes de classes.
Um objeto Proxy tem a mesma interface do objeto-alvo e armazena referências a esses objetos.	Minimiza a comunicação e as dependências entre subsistemas.
Atua como um intermediário entre o cliente e o objeto encapsulado.	Um objeto Façade oferece uma interface única e simplificada.

Perguntas frequentes

P1. Qual é a diferença entre o padrão Decorator (Decorador) e o padrão Proxy?

R: Um Decorator acrescenta comportamentos ao objeto que ele decora em tempo de execução, enquanto um Proxy controla o acesso a um objeto. O relacionamento entre Proxy e RealSubject é estabelecido em

tempo de compilação, e não é dinâmico.

P2. Quais são as desvantagens do padrão Proxy?

R: O padrão Proxy pode aumentar o tempo de resposta. Por exemplo, se o Proxy não tiver uma boa arquitetura ou tiver alguns problemas de desempenho, ele poderá contribuir para aumentar o tempo de resposta de RealSubject. Em geral, tudo depende da qualidade da implementação de um Proxy.

P3. O cliente pode acessar RealSubject de forma independente?

R: Sim, mas há certas vantagens que os Proxies oferecem, atuando como proxies virtuais, proxies remotos e outros, portanto é vantajoso usar o padrão Proxy.

P4. O Proxy acrescenta alguma funcionalidade própria?

R: Um Proxy pode acrescentar funcionalidades a RealSubject sem alterar o código do objeto. Proxy e RealSubject implementariam a mesma interface.

Resumo

Iniciamos o capítulo descrevendo o que são os Proxies. Entendemos o básico sobre o Proxy e como ele é efetivamente usado em arquitetura de software. Em seguida, conhecemos o padrão de projeto Proxy e o contexto em que ele é usado. Vimos como os padrões de projeto Proxy controlam o acesso ao objeto real que oferece a funcionalidade exigida.

Também conhecemos o padrão por meio de um diagrama UML e vimos um exemplo de implementação de código com Python v3.5.

Os padrões Proxy são implementados de quatro maneiras diferentes: proxy virtual, proxy remoto, proxy de proteção e proxy inteligente. Conhecemos cada um deles com um cenário do mundo real.

Comparamos os padrões de projeto Façade e Proxy para que as diferenças entre seus casos de uso e as intenções ficassem claras.

Também incluímos uma seção de Perguntas Frequentes que lhe ajudará a ter mais ideias sobre o padrão e suas possíveis vantagens/desvantagens.

Agora, no final deste capítulo, estamos preparados para conhecer outros padrões estruturais nos próximos capítulos.

CAPÍTULO 6

Padrão Observer – de olho nos objetos

No capítulo anterior, começamos com uma breve introdução ao Proxy e prosseguimos discutindo esse padrão de projeto. Entendemos o conceito do padrão Proxy com um diagrama UML e vimos também como ele é aplicado no mundo real com a ajuda de implementações em Python. Além disso, vimos as vantagens e desvantagens do padrão Proxy na seção de Perguntas Frequentes.

Neste capítulo, discutiremos o terceiro tipo de padrão de projeto – o padrão comportamental. Seremos apresentados ao padrão de projeto Observer (Observador), que se enquadra na categoria dos padrões comportamentais. Discutiremos como esse padrão é usado no desenvolvimento de aplicações de software. Além disso, trabalharemos com um caso de uso de exemplo e faremos a sua implementação com Python v3.5.

Neste capítulo, os seguintes tópicos serão abordados de forma sucinta:

- uma introdução aos padrões de projeto comportamentais;
- o padrão Observer e o seu diagrama UML;
- um caso de uso do mundo real implementado com Python v3.5;
- a eficácia do baixo acoplamento;
- perguntas frequentes.

No final do capítulo, faremos uma síntese de toda a discussão – considere isso como os pontos principais a ser lembrados.

Introdução aos padrões comportamentais

Nos capítulos anteriores do livro, conhecemos os padrões de criação (Singleton) e os padrões estruturais (Façade). Nesta seção, teremos uma ideia sucinta do que são os padrões comportamentais.

Os padrões de criação funcionam com base no modo como os objetos podem ser criados. Eles isolam os detalhes da criação dos objetos. O código é independente do tipo de objeto a ser criado. Os padrões estruturais determinam o design da estrutura de objetos e classes para que estes possam trabalhar em conjunto a fim de alcançar resultados mais amplos. Seu foco principal está em simplificar a estrutura e identificar os relacionamentos entre as classes e os objetos.

Os padrões comportamentais, como o nome sugere, têm como foco as responsabilidades de um objeto. Eles lidam com a interação entre objetos para alcançar funcionalidades mais complexas. Os padrões comportamentais sugerem que, embora os objetos devam ser capazes de interagir uns com os outros, eles devem manter um baixo acoplamento. Conheceremos o princípio do baixo acoplamento mais adiante neste capítulo.

O padrão de projeto Observer é um dos padrões comportamentais mais simples que existem. Vamos nos preparar então e saber mais sobre eles.

Compreendendo o padrão de projeto Observer

No padrão de projeto Observer, um objeto (Subject, ou Sujeito/Assunto/Observável) mantém uma lista de dependentes (Observers, ou Observadores) de modo que o Sujeito possa notificar todos os Observadores acerca das mudanças pelas quais ele passa usando qualquer um dos métodos definidos pelo Observador.

No mundo das aplicações distribuídas, vários serviços interagem uns

com os outros para executar uma operação mais complexa que o usuário queira fazer. Os serviços podem realizar várias operações, mas a operação que eles executam é diretamente ou altamente dependente do estado dos objetos do serviço com o qual a interação ocorre.

Considere um caso de uso de inscrição de usuários em que o serviço de usuário é responsável pelas operações do usuário no site. Vamos supor que tenhamos outro serviço chamado serviço de email que observa o estado do usuário e lhe envia emails. Por exemplo, se o usuário tiver acabado de se inscrever, o serviço de usuário chamará um método do serviço de email que enviará um email para o usuário a fim de fazer uma verificação da conta. Se a conta for verificada, mas tiver poucos créditos, o serviço de email monitorará o serviço de usuário e enviará um email de alerta informando que o usuário tem poucos créditos.

Assim, se houver um serviço básico na aplicação do qual muitos serviços dependam, o serviço básico será o Sujeito que deve ser observado/monitorado pelo Observador para saber se houve mudanças. O Observador, por sua vez, deve fazer alterações no estado de seus próprios objetos ou tomar determinadas atitudes com base nas mudanças que ocorrem no Sujeito. O cenário anterior, em que o serviço dependente monitora mudanças de estado no serviço básico, apresenta um caso clássico do padrão de projeto Observer.

No caso de um sistema de broadcast ou de publicação/assinatura, você verá a utilização do padrão de projeto Observer. Considere um blog como exemplo. Vamos supor que você seja um entusiasta de tecnologia que adora ler os artigos mais recentes sobre Python nesse blog. O que você fará? Você se inscreverá no blog. Assim como você, haverá várias pessoas inscritas que também estarão registradas no blog. Desse modo, sempre que houver um novo blog, você será notificado; se houver uma mudança no blog publicado, você também tomará conhecimento das alterações. O meio pelo qual você será notificado da mudança pode ser um email. Se você aplicar esse cenário ao padrão Observer, o blog será o Sujeito que mantém a lista dos inscritos, isto é, dos Observadores.

Então, quando uma nova entrada for adicionada ao blog, todos os Observadores serão notificados via email ou por meio de qualquer outro sistema de notificação definido pelo Observador.

Os principais objetivos do padrão Observer são estes:

- Ele define uma dependência de um para muitos (one-to-many) entre os objetos, de modo que qualquer mudança em um objeto será notificada aos demais objetos dependentes automaticamente.
- Encapsula o componente nuclear, isto é, o Sujeito.

O padrão Observer é usado nos diversos cenários a seguir:

- na implementação do serviço de Eventos em sistemas distribuídos;
- em um framework para uma agência de notícias;
- no mercado de ações, que também representa um ótimo caso para o padrão Observer.

O código Python a seguir implementa o padrão de projeto Observer:

```
class Subject:
    def __init__(self):
        self.__observers = []

    def register(self, observer):
        self.__observers.append(observer)

    def notifyAll(self, *args, **kwargs):
        for observer in self.__observers:
            observer.notify(self, *args, **kwargs)


class Observer1:
    def __init__(self, subject):
        subject.register(self)

    def notify(self, subject, *args):
        print(type(self).__name__,':: Got', args, 'From', subject)


class Observer2:
```

```
    def __init__(self, subject):
        subject.register(self)

    def notify(self, subject, *args):
        print(type(self).__name__, ':: Got', args, 'From', subject)


subject = Subject()
observer1 = Observer1(subject)
observer2 = Observer2(subject)
subject.notifyAll('notification')
```

Esta é a saída do código anterior:

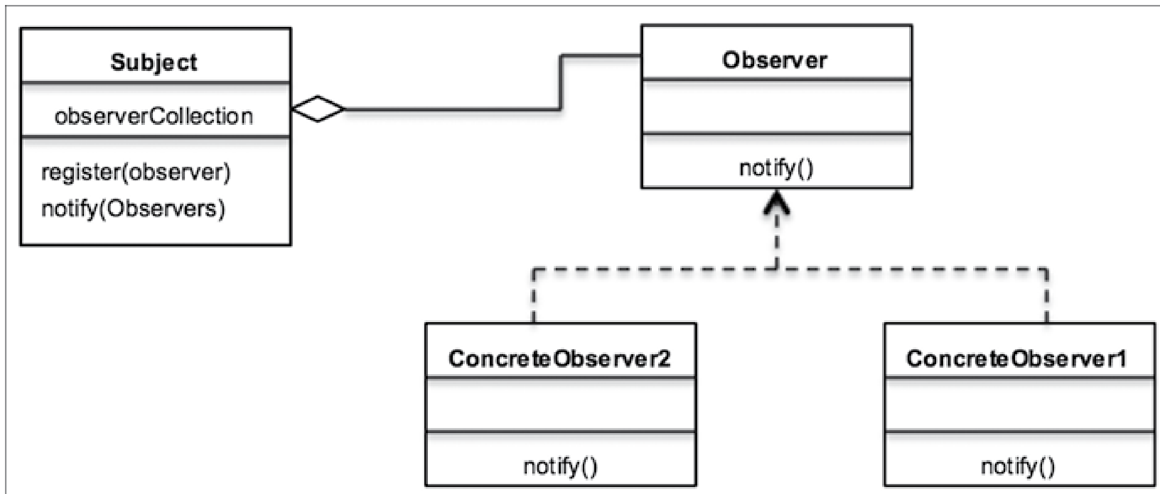
Aprendendo Padrões de Projeto em Python

Tire proveito da eficácia dos padrões de projeto (design patterns) em Python para resolver problemas do mundo real em arquitetura e design de software

Um diagrama de classes UML para o padrão Observer

Vamos entender melhor o padrão Observer com a ajuda do diagrama UML a seguir.

Conforme discutimos no parágrafo anterior, o padrão Observer tem dois atores principais: `Subject` e `Observer`. Vamos colocá-los em um diagrama UML e ver como é a aparência dessas classes:



Como podemos observar no diagrama UML, você perceberá que há três participantes principais nesse padrão:

- Subject – a classe Subject tem conhecimento do Observer. Essa classe tem métodos como register() e deregister(), usados pelos ObserverS para se registrar junto à classe Subject. Um Subject, portanto, pode tratar vários ObserverS.
- Observer – define uma interface para objetos que estão observando o Subject. Define métodos que devem ser implementados pelo ConcreteObserver para que este seja notificado das mudanças no Subject.
- ConcreteObserver – armazena o estado que deve ser consistente com o estado do Subject. Implementa a interface Observer para manter o estado consistente com as mudanças no Subject.

O fluxo é simples. Os ConcreteObserverS se registram junto ao Subject implementando a interface fornecida pelo Observer. Sempre que houver uma mudança de estado, o Subject notificará todos os ConcreteObserverS com o método de notificação fornecido pelos ObserverS.

Padrão Observer no mundo real

Vamos considerar o caso de uma agência de notícias a fim de demonstrar o cenário do mundo real para o padrão Observer. As agências de notícias geralmente reúnem notícias de vários locais e as publicam para que os assinantes tenham acesso a elas. Vamos dar uma

olhada nas considerações de design para esse caso de uso.

Com informações enviadas/recebidas em tempo real, uma agência de notícias deve ser capaz de publicar as notícias aos seus assinantes assim que for possível. Além do mais, em virtude dos avanços no mercado de tecnologia, não só os jornais, mas também os assinantes podem ser de tipos diferentes, por exemplo, assinantes via email, dispositivos móveis, SMS ou chamadas de voz. Também devemos ser capazes de acrescentar qualquer outro tipo de assinante no futuro e planejar o orçamento para qualquer nova tecnologia.

Vamos desenvolver uma aplicação em Python v3.5 e implementar o caso de uso anterior. Começaremos com o Sujeito, que é quem publica as notícias.

- O comportamento do Sujeito é representado pela classe NewsPublisher.
- NewsPublisher oferece uma interface com a qual os assinantes podem trabalhar.
- O método attach() é usado pelo Observer para se registrar junto a NewsPublisher, e o método detach() ajuda na remoção da assinatura de Observer.
- O método subscriber() devolve a lista de todos os assinantes que já se registraram junto a Subject.
- O método notifySubscriber() itera por todos os assinantes que se registraram junto a NewsPublisher.
- O método addNews() é usado pela agência de notícias para criar novas notícias, e getNews() é usado para devolver as notícias mais recentes; então o Observer é notificado.

Vamos observar inicialmente a classe NewsPublisher:

```
class NewsPublisher:
    def __init__(self):
        self.__subscribers = []
        self.__latestNews = None

    def attach(self, subscriber):
```

```
        self.__subscribers.append(subscriber)

    def detach(self):
        return self.__subscribers.pop()

    def subscribers(self):
        return [type(x).__name__ for x in self.__subscribers]

    def notifySubscribers(self):
        for sub in self.__subscribers:
            sub.update()

    def addNews(self, news):
        self.__latestNews = news

    def getNews(self):
        return "Got News:", self.__latestNews
```

Vamos agora discutir a interface Observer.

- Nesse exemplo, Subscriber representa o Observer. É uma classe-base abstrata que representa qualquer outro ConcreteObserver.
- Subscriber tem o método update(), que deve ser implementado pelos ConcreteObserverS.
- O método update() é implementado por ConcreteObserver para que este possa ser notificado pelo Subject (NewsPublishers) de qualquer notícia publicada.

Vamos ver agora o código da classe abstrata Subscriber:

```
from abc import ABCMeta, abstractmethod

class Subscriber(metaclass=ABCMeta):

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

Também desenvolvemos algumas classes que representam ConcreteObserverS.

- Neste caso, temos dois observadores principais: EmailSubscriber e

SMSSubscriber, que implementam a interface de assinante.

- Além desses dois observadores, temos outro, AnyOtherObserver, que demonstra o baixo acoplamento entre os ObserverS e Subject.
- O método __init__() de cada um desses ConcreteObserverS os registra junto a NewsPublisher com o método attach().
- O método update() de ConcreteObserver é usado internamente por NewsPublisher para notificação sobre as adições de notícias.

Eis o modo como a classe SMSSubscriber é implementada:

```
class SMSSubscriber:
    def __init__(self, publisher):
        self.publisher = publisher
        self.publisher.attach(self)

    def update(self):
        print(type(self).__name__, self.publisher.getNews())

class EmailSubscriber:
    def __init__(self, publisher):
        self.publisher = publisher
        self.publisher.attach(self)

    def update(self):
        print(type(self).__name__, self.publisher.getNews())

class AnyOtherSubscriber:
    def __init__(self, publisher):
        self.publisher = publisher
        self.publisher.attach(self)

    def update(self):
        print(type(self).__name__, self.publisher.getNews())
```

Agora que todos os assinantes necessários foram implementados, vamos dar uma olhada nas classes NewsPublisher e SMSSubscriber em ação.

- O cliente cria um objeto para NewsPublisher, que é usado pelos ConcreteObserverS para várias operações.

- As classes SMSSubscriber, EmailSubscriber e AnyOtherSubscriber são inicializadas com objetos publisher.
- Em Python, quando criamos objetos, o método __init__() é chamado. Na classe ConcreteObserver, o método __init__() usa internamente o método attach() de NewsPublisher para se registrar e ser notificado de atualizações em notícias.
- Então exibimos a lista de todos os assinantes (ConcreteObserverS) que foram registrados junto ao Subject.
- O objeto de NewsPublisher (news_publisher) é então usado para criar novas notícias com o método addNews().
- O método notifySubscribers() de NewsPublisher é usado para notificar todos os assinantes sobre a adição de notícias. O método notifySubscribers() internamente chama o método update() implementado pelos ConcreteObserverS para que eles obtenham as notícias mais recentes.
- NewsPublisher também tem o método detach(), que remove o assinante da lista de assinantes registrados.

A implementação a seguir representa as interações entre o Subject e os ObserverS:

```
if __name__ == '__main__':
    news_publisher = NewsPublisher()

    for Subscribers in [SMSSubscriber, EmailSubscriber, AnyOtherSubscriber]:
        Subscribers(news_publisher)
    print("\nSubscribers:", news_publisher.subscribers())

    news_publisher.addNews('Hello World!')
    news_publisher.notifySubscribers()

    print("\nDetached:", type(news_publisher.detach()).__name__)
    print("\nSubscribers:", news_publisher.subscribers())

    news_publisher.addNews('My second news!')
    news_publisher.notifySubscribers()
```

Esta é a saída do código anterior:

```
Subscribers: ['SMSSubscriber', 'EmailSubscriber', 'AnyOtherSubscriber']
SMSSubscriber ('Got News:', 'Hello World!')
EmailSubscriber ('Got News:', 'Hello World!')
AnyOtherSubscriber ('Got News:', 'Hello World!')

Detached: AnyOtherSubscriber

Subscribers: ['SMSSubscriber', 'EmailSubscriber']
SMSSubscriber ('Got News:', 'My second news!')
EmailSubscriber ('Got News:', 'My second news!')
```

Modelos do padrão Observer

Há duas maneiras diferentes de notificar o Observer das mudanças que ocorrem no Subject. Elas podem ser classificadas como modelos push e pull.

Modelo pull

No modelo pull, os Observers desempenham um papel ativo da seguinte maneira:

- O Subject faz um broadcast para todos os Observers registrados quando há uma mudança.
- O Observer é responsável por obter as mudanças, isto é, o assinante deve buscar os dados quando houver uma alteração.
- O modelo pull não é eficiente, pois envolve dois passos – o primeiro passo para o Subject notificar o Observer e o segundo passo para o Observer obter os dados necessários do Subject.

Modelo push

No modelo push, o Subject é quem desempenha um papel dominante, desta maneira:

- De modo diferente do modelo pull, as mudanças são enviadas pelo Subject ao Observer.
- Nesse modelo, o Subject pode enviar informações detalhadas ao Observer (mesmo que não seja necessário). Isso pode resultar em

tempos de resposta mais lentos quando uma grande quantidade de dados é enviada pelo Subject, mas não é realmente usada pelo Observer.

- Somente os dados necessários devem ser enviados pelo Subject para que o desempenho seja melhor.

Baixo acoplamento e o padrão Observer

O baixo acoplamento é um princípio de design importante, que deve ser usado em aplicações de software. O propósito principal é esforçar-se para ter designs com baixo acoplamento entre objetos que interagem uns com os outros. O acoplamento refere-se ao grau de conhecimento que um objeto tem sobre outro com o qual ele interage.

Designs com baixo acoplamento nos permitem construir sistemas orientados a objetos flexíveis, capazes de lidar com mudanças, pois reduzem a dependência entre vários objetos.

A arquitetura com baixo acoplamento garante as seguintes características:

- reduz o risco de que uma mudança em um elemento possa gerar um impacto imprevisto em outros elementos;
- simplifica os testes, a manutenção e a resolução de problemas;
- o sistema pode ser facilmente separado em elementos definidos.

O padrão Observer possibilita um design de objetos em que o Subject e o Observer têm baixo acoplamento. Os pontos a seguir fornecerão uma melhor explicação:

- A única informação que o Subject deve ter sobre o Observer é que ele implementa uma determinada interface. Ele não precisa conhecer a classe ConcreteObserver.
- Qualquer novo Observer pode ser adicionado a qualquer momento (como vimos no exemplo anterior neste capítulo).
- O Subject não precisa ser nem um pouco modificado para adicionar

qualquer novo Observer. No exemplo, vimos que AnyOtherObserver pôde ser adicionado/removido sem nenhuma alteração no Subject.

- Os Subjects ou os Observers não estão amarrados e podem ser usados de modo independente. Assim, o Observer pode ser reutilizado em qualquer outro lugar, caso seja necessário.
- Alterações no Subject ou no Observer não afetarão um ao outro. Como ambos são independentes, isto é, têm baixo acoplamento, eles são livres para fazer suas próprias alterações.

Padrão Observer – vantagens e desvantagens

O padrão Observer apresenta as seguintes vantagens:

- Oferece suporte para o princípio do baixo acoplamento entre objetos que interajam uns com os outros.
- Permite enviar dados a outros objetos de modo eficiente, sem qualquer mudança nas classes Subject ou Observer.
- Os Observers podem ser adicionados/removidos a qualquer momento.

O padrão Observer apresenta as seguintes desvantagens:

- A interface Observer deve ser implementada por ConcreteObserver, o que envolve herança. Não há opção para usar composição, pois a interface Observer pode ser instanciada.
- Se não for implementado corretamente, o Observer pode acrescentar complexidade e levar a problemas imprevistos de desempenho.
- Em uma aplicação de software, às vezes, as notificações podem não ser confiáveis e resultar em condições de concorrência (race conditions) ou inconsistência.

Perguntas frequentes

P1. É possível haver vários Subjects e Observers?

R: É possível que haja uma situação em que uma aplicação de software tenha vários Subjects e Observers. Para isso funcionar, os Observers devem ser notificados das mudanças nos Subjects e de qual Subject sofreu a mudança.

P2. Quem é o responsável por disparar a atualização?

R: Como vimos antes, o padrão Observer pode trabalhar tanto com o modelo push quanto com o modelo pull. Geralmente, o Sujeito dispara o método de atualização quando há mudanças; às vezes, porém, conforme as necessidades da aplicação, o Observador também pode disparar notificações. Entretanto é preciso ter cuidado para que a frequência não seja muito alta; caso contrário, isso poderá resultar na degradação do desempenho, especialmente quando as atualizações para o Sujeito forem menos frequentes.

P3. O Subject ou o Observer podem ser usados para acesso em qualquer outro caso de uso?

R: Sim, é a eficácia do baixo acoplamento manifestada no padrão Observer. Ambos, Subject e Observer, podem ser usados de forma independente.

Resumo

Iniciamos o capítulo descrevendo os padrões de projeto comportamentais. Compreendemos o básico do padrão Observer e como ele é efetivamente usado em arquiteturas de software. Vimos como os padrões de projeto Observer são usados para notificar o Observer das mudanças que ocorrem no Subject. Eles administram a interação e as dependências de um para muitos (one-to-many) entre os objetos.

Também conhecemos o padrão com um diagrama UML e vimos um exemplo de implementação de código com Python v3.5.

Os padrões Observer são implementados de duas formas diferentes: por meio dos modelos push e pull. Conhecemos cada um desses modelos e discutimos sua implementação e o impacto no desempenho.

Compreendemos o princípio do baixo acoplamento no design de

software e vimos de que modo o padrão Observer tira proveito desse princípio no desenvolvimento de aplicações.

Também incluímos uma seção de Perguntas Frequentes que lhe ajudará a ter mais ideias sobre o padrão e suas possíveis vantagens/desvantagens.

No final deste capítulo, estamos preparados para conhecer outros padrões comportamentais nos capítulos seguintes.

CAPÍTULO 7

Padrão Command – encapsulando chamadas

No capítulo anterior, começamos com uma introdução aos padrões de projeto comportamentais. Conhecemos o conceito de ObserverS e discutimos o padrão de projeto Observer. Entendemos o conceito do padrão de projeto Observer com um diagrama UML e vimos também como ele é aplicado no mundo real com a ajuda de implementações em Python. Além disso, discutimos os prós e contras do padrão Observer. Também conhecemos o padrão Observer com uma seção de Perguntas Frequentes e sintetizamos a discussão no final do capítulo.

Neste capítulo, discutiremos o padrão de projeto Command (Comando). Assim como o padrão Observer, o padrão Command se enquadra na categoria dos padrões comportamentais. Seremos apresentados ao padrão de projeto Command e discutiremos como ele é usado no desenvolvimento de aplicações de software. Além disso, trabalharemos com um caso de uso de exemplo e faremos a sua implementação com Python v3.5.

Neste capítulo, os seguintes tópicos serão abordados de forma sucinta:

- uma introdução aos padrões de projeto Command;
- o padrão Command e o seu diagrama UML;
- um caso de uso do mundo real implementado com Python v3.5;
- os prós e contras do padrão Command;
- perguntas frequentes.

Introdução ao padrão Command

Como vimos no capítulo anterior, os padrões comportamentais têm como foco as responsabilidades de um objeto. Eles lidam com a interação entre objetos para alcançar funcionalidades mais complexas. O padrão Command é um padrão de projeto comportamental, em que um objeto é usado para encapsular todas as informações necessárias para executar uma ação ou disparar um evento posteriormente. Essas informações incluem:

- o nome do método;
- um objeto que seja dono do método;
- valores para os parâmetros do método.

Vamos entender o padrão com um exemplo bem simples de software. Considere o caso de um assistente (wizard) de instalação. Um assistente típico pode ter várias etapas ou telas que capturem as preferências de um usuário. Enquanto navega pelo assistente, o usuário faz determinadas escolhas. Os assistentes geralmente são implementados com o padrão Command. Um assistente é iniciado com um objeto chamado `Command`. As preferências ou escolhas feitas pelo usuário nas várias etapas do assistente são então armazenadas no objeto `Command`. Quando o usuário clica no botão **Finish** (Fim) na última tela do assistente, o objeto `Command` executa um método `execute()`, que considera todas as opções armazenadas e faz o procedimento de instalação apropriado. Assim, todas as informações relacionadas às escolhas são encapsuladas em um objeto que pode ser usado mais tarde para executar uma ação.

Outro exemplo simples é o spooler de impressão. Um spooler pode ser implementado como um objeto `Command` que armazena informações como tipo de página (A5-A1), retrato/paisagem, agrupado/não agrupado. Quando o usuário imprimir algo (por exemplo, uma imagem), o spooler executará o método `execute()` do objeto `Command` e a imagem será impressa com as preferências definidas.

Compreendendo o padrão de projeto Command

O padrão Command trabalha com os seguintes termos: Command, Receiver, Invoker e Client.

- Um objeto Command conhece os objetos Receiver e invoca um método desse objeto.
- Os valores dos parâmetros do método receptor (receiver) são armazenados no objeto Command.
- O chamador ou invocador (invoker) sabe como executar um comando.
- O cliente cria um objeto Command e define o seu receptor.

Os principais objetivos do padrão Command são:

- encapsular uma requisição como um objeto;
- possibilitar a parametrização dos clientes com diferentes requisições;
- permitir salvar as requisições em uma fila (discutiremos isso mais adiante neste capítulo);
- oferecer uma callback orientada a objetos.

O padrão Command pode ser usado nos diversos cenários a seguir:

- parametrizar objetos de acordo com a ação a ser executada;
- adicionar ações em uma fila e executar as requisições em pontos diferentes;
- criar uma estrutura para operações de alto nível baseadas em operações menores.

O código Python a seguir implementa o padrão de projeto Command. Falamos sobre o exemplo do assistente anteriormente no capítulo. Suponha que queiramos desenvolver um assistente para instalação ou, popularmente, um instalador. Em geral, uma instalação implica copiar ou mover arquivos no sistema de arquivos de acordo com as escolhas feitas por um usuário. No exemplo a seguir, no código do cliente,

começamos criando o objeto Wizard e usamos o método preferences(), que armazena as escolhas feitas pelo usuário nas várias telas do assistente. No assistente, quando o botão **Finish** (Fim) for clicado, o método execute() será chamado. Esse método toma as preferências e inicia a instalação:

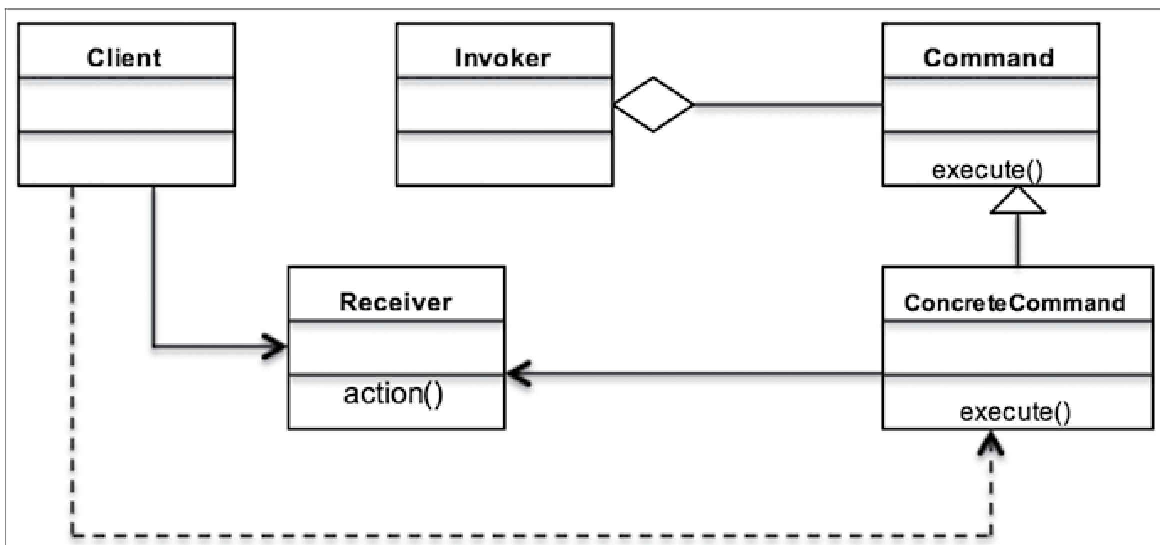
```
class Wizard():

    def __init__(self, src, rootdir):
        self.choices = []
        self.rootdir = rootdir
        self.src = src

    def preferences(self, command):
        self.choices.append(command)

    def execute(self):
        for choice in self.choices:
            if list(choice.values())[0]:
                print("Copying binaries --", self.src, " to ", self.rootdir)
            else:
                print("No Operation")


if __name__ == '__main__':
    ## Código do cliente
    wizard = Wizard('python3.5.gzip', '/usr/bin/')
    ## Os usuários escolhem instalar somente Python
    wizard.preferences({'python':True})
    wizard.preferences({'java':False})
    wizard.execute()
```

Esta é a saída do código anterior:

Um diagrama de classes UML para o padrão Command

Vamos compreender melhor o padrão Command com a ajuda do diagrama UML a seguir.

Conforme discutimos no parágrafo anterior, o padrão Command tem os seguintes participantes principais: Command, ConcreteCommand, Receiver, Invoker e Client. Vamos colocá-los em um diagrama UML e ver como é a aparência dessas classes:



Como podemos observar no diagrama UML, você perceberá que há cinco participantes principais nesse padrão:

- Command – declara uma interface para executar uma operação.
- ConcreteCommand – define um vínculo entre o objeto Receiver e a ação.
- Client – cria um objeto ConcreteCommand e define o seu receptor.

- Invoker – pede a ConcreteCommand que cuide da requisição.
- Receiver – sabe como executar as operações associadas ao tratamento da requisição.

O fluxo é simples. O cliente pede que um comando seja executado. O chamador (invoker) toma o comando, o encapsula e o insere em uma fila. A classe ConcreteCommand é responsável pelo comando solicitado e pede que o receptor execute a ação especificada. O exemplo de código a seguir serve para compreender o padrão e apresenta todos os participantes envolvidos:

```
from abc import ABCMeta, abstractmethod

class Command(metaclass=ABCMeta):
    def __init__(self, recv):
        self.recv = recv

    def execute(self):
        pass


class ConcreteCommand(Command):
    def __init__(self, recv):
        self.recv = recv

    def execute(self):
        self.recv.action()


class Receiver:
    def action(self):
        print("Receiver Action")


class Invoker:
    def command(self, cmd):
        self.cmd = cmd

    def execute(self):
        self.cmd.execute()
```

```
if __name__ == '__main__':
    recv = Receiver()
    cmd = ConcreteCommand(recv)
    invoker = Invoker()
    invoker.command(cmd)
    invoker.execute()
```

Implementando o padrão Command no mundo real

Tomaremos um exemplo com a Bolsa de Valores (muito comentada no mundo da internet) para demonstrar a implementação do padrão Command. O que acontece em uma Bolsa de Valores? Você, como usuário da Bolsa, cria pedidos para comprar ou vender ações. Geralmente, você não as compra nem vende; é o agente ou o corretor que desempenha o papel de intermediário entre você e a Bolsa de Valores. O agente é responsável por levar sua requisição à Bolsa de Valores e fazer o trabalho. Suponha que você queira vender uma ação na segunda-feira de manhã, quando a Bolsa abre. Você ainda pode fazer a solicitação de vender as ações no domingo à noite ao seu agente, mesmo que a Bolsa não esteja aberta ainda. O agente então coloca essa requisição na fila para que seja executada na segunda-feira de manhã, quando a Bolsa estiver aberta para negociações. Este é um caso clássico para o padrão Command.

Considerações de design

Com base no diagrama UML, aprendemos que o padrão Command tem quatro participantes principais: `Command`, `ConcreteCommand`, `Invoker` e `Receiver`. No cenário anterior, devemos criar uma interface `Order` que define o pedido feito por um cliente. Devemos definir classes `ConcreteCommand` para comprar ou vender uma ação. Uma classe também deve ser definida para a Bolsa de Valores. Devemos definir a classe `Receiver`, que realmente executará a negociação, e o agente (conhecido como chamador), que

invoca o pedido e faz com que ele seja executado pelo receptor.

Vamos desenvolver uma aplicação em Python v3.5 e implementar o caso de uso anterior. Começaremos pelo objeto Command, que é Order.

- O objeto Command é representado pela classe Order.
- Order oferece uma interface (classe-base abstrata de Python) para que ConcreteCommand possa implementar o comportamento.
- O método execute() é o método abstrato que deve ser definido pelas classes ConcreteCommand para executar a classe Order.

O código a seguir representa a classe abstrata Order e o método abstrato execute():

```
from abc import ABCMeta, abstractmethod

class Order(metaclass=ABCMeta):

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

Também desenvolvemos algumas classes que representam ConcreteCommand.

- Nesse caso, temos duas classes concretas principais: BuyStockOrder e SellStockOrder, que implementam a interface Order.
- As duas classes ConcreteCommand utilizam o objeto do sistema de negociação de ações para que possam definir as ações apropriadas para o sistema.
- O método execute() de cada uma dessas classes ConcreteCommand utiliza o objeto da Bolsa de Valores para executar as ações de comprar e vender.

Vamos agora dar uma olhada nas classes concretas que implementam a interface:

```
class BuyStockOrder(Order):
    def __init__(self, stock):
        self.stock = stock
```

```
    def execute(self):
        self.stock.buy()


class SellStockOrder(Order):
    def __init__(self, stock):
        self.stock = stock

    def execute(self):
        self.stock.sell()
```

Agora vamos discutir o sistema de negociações de ações e ver como ele é implementado.

- A classe StockTrade representa o objeto Receiver nesse exemplo.
- Ela define vários métodos (ações) para executar as requisições apresentadas pelos objetos ConcreteCommand.
- Os métodos buy() e sell() são definidos pelo receptor e são chamados por BuyStockOrder e SellStockOrder respectivamente para comprar ou vender a ação na Bolsa.

Dê uma olhada na classe StockTrade:

```
class StockTrade:
    def buy(self):
        print("You will buy stocks")

    def sell(self):
        print("You will sell stocks")
```

Outra parte da implementação é o chamador:

- A classe Agent representa o chamador.
- O agente é o intermediário entre o cliente e StockExchange e executa os pedidos apresentados pelo cliente.
- O agente define um membro de dado, __orderQueue (uma lista), que atua como uma fila. Qualquer nova requisição feita pelo cliente é adicionada na fila.
- O método placeOrder() de Agent é responsável por colocar as requisições

na fila e executá-las.

O código a seguir representa a classe Agent, que desempenha o papel do Invoker:

```
class Agent:
    def __init__(self):
        self.__orderQueue = []

    def placeOrder(self, order):
        self.__orderQueue.append(order)
        order.execute()
```

Vamos colocar agora todas as classes anteriores em perspectiva e observar como o cliente é implementado.

- O cliente inicialmente define o seu receptor: a classe StockTrade.
- Ele cria requisições para comprar e vender ações com BuyStockOrder e SellStockOrder (ConcreteCommand), que executam a ação em StockTrade.
- O objeto chamador é criado por meio da instanciação da classe Agent.
- O método placeOrder() de Agent é usado para obter as requisições feitas pelo cliente.

O código a seguir implementa o cliente:

```
if __name__ == '__main__':
    #Cliente
    stock = StockTrade()
    buyStock = BuyStockOrder(stock)
    sellStock = SellStockOrder(stock)

    #Chamador
    agent = Agent()
    agent.placeOrder(buyStock)
    agent.placeOrder(sellStock)
```

A seguir, apresentamos a saída do trecho de código anterior.

```
You will buy stocks
You will sell stocks
```

Há várias maneiras pelas quais o padrão Command é usado em

aplicações de software. Discutiremos duas implementações específicas que são muito relevantes para aplicações em nuvem.

- Operações refazer (redo) ou rollback:
  - Enquanto estão implementando operações de rollback ou de refazer, os desenvolvedores têm duas opções.
  - Eles podem criar um snapshot (imagem instantânea) no sistema de arquivos ou na memória e, quando um rollback for solicitado, reverter para esse snapshot.
  - Com o padrão Command, você pode armazenar a sequência de comandos e, quando uma operação para refazer for solicitada, executar novamente o mesmo conjunto de ações.
- Execução de tarefas assíncronas:
  - Em sistemas distribuídos, com frequência, precisamos do recurso de executar tarefas assíncronas para que o serviço nuclear jamais seja bloqueado caso haja mais requisições.
  - No padrão Command, o objeto chamador pode manter uma fila de requisições e enviar essas tarefas para o objeto Receiver para que seja possível atuar sobre elas de forma independente da thread principal da aplicação.

Vantagens e desvantagens dos padrões Command

O padrão Command apresenta as seguintes vantagens:

- desacopla as classes que invocam a operação do objeto que sabe como executá-la;
- permite criar uma sequência de comandos oferecendo um sistema de fila;
- extensões para adicionar um novo comando são fáceis de implementar, e isso pode ser feito sem alterar o código existente;
- você também pode definir um sistema de rollback com o padrão

Command – por exemplo, no caso do Wizard, poderíamos escrever um método para rollback.

A seguir, apresentamos as desvantagens do padrão Command.

- Há um número elevado de classes e objetos trabalhando em conjunto para alcançar um objetivo. Os desenvolvedores de aplicações devem tomar cuidado para desenvolver essas classes corretamente.
- Todo comando individual é uma classe ConcreteCommand que aumenta o volume de classes para implementação e manutenção.

Perguntas frequentes

P1. É possível que não haja nenhum Receiver e ConcreteCommand que implementem o método de execução?

R: Sim, definitivamente, é possível fazer isso. Muitas aplicações de software usam o padrão Command dessa maneira também. A única questão a ser observada nesse caso é a interação entre o chamador e o receptor. Se o receptor não estiver definido, o nível de desacoplamento diminui; além do mais, a facilidade de parametrizar comandos será perdida.

P2. Qual estrutura de dados devo usar para implementar o sistema de fila no objeto chamador?

R: No exemplo da Bolsa de Valores que estudamos anteriormente neste capítulo, usamos uma lista para implementar a fila. No entanto o padrão Command menciona uma implementação de pilha que é realmente conveniente no caso do desenvolvimento das operações de refazer (redo) ou de rollback.

Resumo

Iniciamos o capítulo descrevendo o padrão de projeto Command e como ele é efetivamente usado na arquitetura de software.

Vimos como os padrões de projeto Command são utilizados para

encapsular todas as informações necessárias e disparar um evento ou uma ação mais tarde.

Também conhecemos o padrão com um diagrama UML e vimos um exemplo de implementação de código com Python v3.5, juntamente com a explicação.

Além disso, incluímos uma seção de Perguntas Frequentes que lhe ajudará a ter mais ideias sobre o padrão e suas possíveis vantagens/desvantagens.

Vamos estudar outros padrões de projeto comportamentais nos próximos capítulos.

CAPÍTULO 8

Padrão Template Method – encapsulando algoritmos

No capítulo anterior, começamos com uma introdução ao padrão de projeto Command (Comando), em que um objeto é usado para encapsular todas as informações necessárias a fim de executar uma ação ou disparar um evento posteriormente. Entendemos o conceito do padrão de projeto Command com um diagrama UML e vimos também como ele é aplicado no mundo real com a ajuda de uma implementação Python. Discutimos os prós e contras dos padrões Command, exploramos mais na seção Perguntas Frequentes e resumimos a discussão no final do capítulo.

Neste capítulo, discutiremos o padrão de projeto Template; assim como o padrão Command, esse padrão se enquadra na categoria dos padrões comportamentais. Seremos apresentados ao padrão de projeto Template e discutiremos como ele é usado no desenvolvimento de aplicações de software. Trabalharemos também com um caso de uso de exemplo e faremos a sua implementação com Python v3.5.

Neste capítulo, os seguintes tópicos serão abordados de forma sucinta:

- uma introdução ao padrão de projeto Template Method (Método Template);
- o padrão Template e o seu diagrama UML;
- um caso de uso do mundo real com código implementado em Python v3.5;

- o padrão Template – seus prós e contras;
- o Princípio de Hollywood, o Template Method e o hook no Template;
- perguntas frequentes.

No final deste capítulo, você será capaz de analisar situações em que o padrão de projeto Template é aplicável e usá-lo de modo eficiente para resolver problemas relacionados a design. Também sintetizaremos toda a discussão sobre o padrão Template Method para apresentar os pontos a ser lembrados.

Definindo o padrão Template Method

Como vimos no capítulo anterior, os padrões comportamentais têm como foco as responsabilidades de um objeto. Eles lidam com a interação entre objetos para alcançar funcionalidades mais complexas. O padrão Template Method é um padrão de projeto comportamental que define o esqueleto do programa ou um algoritmo em um método chamado Método Template. Por exemplo, você pode definir os passos para preparar uma bebida como um algoritmo em um Método Template. O padrão Template Method também ajuda a redefinir ou personalizar determinados passos do algoritmo adiando a implementação de alguns desses passos para as subclasses. Isso significa que as subclasses podem redefinir o seu próprio comportamento. Por exemplo, nesse caso, as subclasses podem implementar os passos para preparar um chá usando o Método Template para preparar uma bebida. É importante observar que a alteração nos passos (conforme feita pelas subclasses) não exerce impacto na estrutura do algoritmo original. Assim, o recurso das subclasses de poder sobrescrever no padrão Template Method permite a criação de diferentes comportamentos ou algoritmos.

Para discutir o padrão Template Method na terminologia do desenvolvimento de software, uma classe abstrata é usada para definir os passos do algoritmo. Esses passos também são conhecidos como

operações primitivas no contexto do padrão Template Method. Eles são definidos com métodos abstratos, e o Método Template define o algoritmo. A ConcreteClass (que é uma subclasse da classe abstrata) implementa os passos do algoritmo específicos para a subclasse.

O padrão Template Method é usado nos seguintes casos:

- quando vários algoritmos ou classes implementam uma lógica semelhante ou idêntica;
- quando a implementação dos algoritmos em subclasses ajuda a reduzir a duplicação de código;
- quando vários algoritmos podem ser definidos ao deixar que as subclasses implementem o comportamento usando o recurso de sobrescrita.

Vamos entender o padrão com um exemplo simples do cotidiano. Pense no que todos fazemos quando preparamos chá ou café. No caso do café, execute os seguintes passos para preparar a bebida:

1. Ferva a água.
2. Passe a água fervente pelo pó de café.
3. Coloque o café em uma xícara.
4. Adicione açúcar ou leite na xícara.
5. Misture, e o café está pronto.

Por outro lado, se quiser preparar uma xícara de chá, você executará os passos a seguir:

1. Ferva a água.
2. Coloque o saquinho de chá.
3. Coloque o chá em uma xícara.
4. Adicione limão ao chá.
5. Misture, e o chá está pronto.

Se analisar os dois métodos de preparação, você perceberá que os dois

procedimentos são mais ou menos iguais. Nesse caso, podemos usar o padrão Template Method de modo eficiente. Como podemos implementá-lo? Definimos uma classe Beverage que tem métodos abstratos comuns para preparar chá e café, por exemplo, boilWater(). Também definimos o Método Template preparation(), que acionará a sequência de passos na preparação da bebida (o algoritmo). Deixaremos as classes concretas PrepareCoffee e PrepareTea definirem os passos personalizados para atingir os objetivos, que são preparar café e chá. É assim que o padrão Template Method evita duplicação de código.

Outro exemplo simples é o compilador usado pelas linguagens de computador. Um compilador executa essencialmente duas tarefas: reúne o código-fonte e compila gerando um objeto-alvo. Se precisarmos definir um compilador cruzado (cross-compiler) para dispositivos iOS, podemos implementar isso com a ajuda do padrão Template Method. Discutiremos melhor esse exemplo de forma detalhada mais adiante neste capítulo.

Compreendendo o padrão de projeto Template Method

Para resumir, os principais objetivos do padrão Template Method são:

- definir o esqueleto de um algoritmo com operações primitivas;
- redefinir determinadas operações na subclasse sem alterar a estrutura do algoritmo;
- reutilizar código e evitar esforços duplicados;
- tirar proveito de interfaces ou implementações comuns.

O padrão Template Method trabalha com os seguintes termos: AbstractClass, ConcreteClass, Método Template e Client.

- AbstractClass: declara uma interface para definir os passos do algoritmo.
- ConcreteClass: define passos específicos da subclasse.
- template_method(): define o algoritmo chamando os métodos dos passos.

Falamos sobre o exemplo do compilador anteriormente no capítulo.

Suponha que queiramos desenvolver o nosso próprio compilador cruzado para um dispositivo iOS e executar o programa.

Inicialmente, desenvolvemos uma classe abstrata (compilador) que define o algoritmo de um compilador. As operações feitas pelo compilador são reunir o código-fonte escrito em uma linguagem de programação e então compilá-lo para obter o código-objeto (formato binário). Definimos esses passos como os métodos abstratos collectSource() e compileToObject(), e definimos também o método run(), responsável pela execução do programa. O algoritmo é especificado pelo método compileAndRun(), que, internamente, chama os métodos collectSource(), compileToObject() e run() para definir o algoritmo do compilador. A classe concreta iOSCompiler agora implementa os métodos abstratos e compila/executa o código Swift no dispositivo iOS.

A linguagem de programação Swift é usada para desenvolver aplicações na plataforma iOS.

O código Python a seguir implementa o padrão de projeto Template Method:

```
from abc import ABCMeta, abstractmethod

class Compiler(metaclass=ABCMeta):
    @abstractmethod
    def collectSource(self):
        pass

    @abstractmethod
    def compileToObject(self):
        pass

    @abstractmethod
    def run(self):
        pass

    def compileAndRun(self):
```

```
        self.collectSource()
        self.compileToObject()
        self.run()


class iOSCompiler(Compiler):
    def collectSource(self):
        print("Collecting Swift Source Code")

    def compileToObject(self):
        print("Compiling Swift code to LLVM bitcode")

    def run(self):
        print("Program runing on runtime environment")


iOS = iOSCompiler()
iOS.compileAndRun()
```

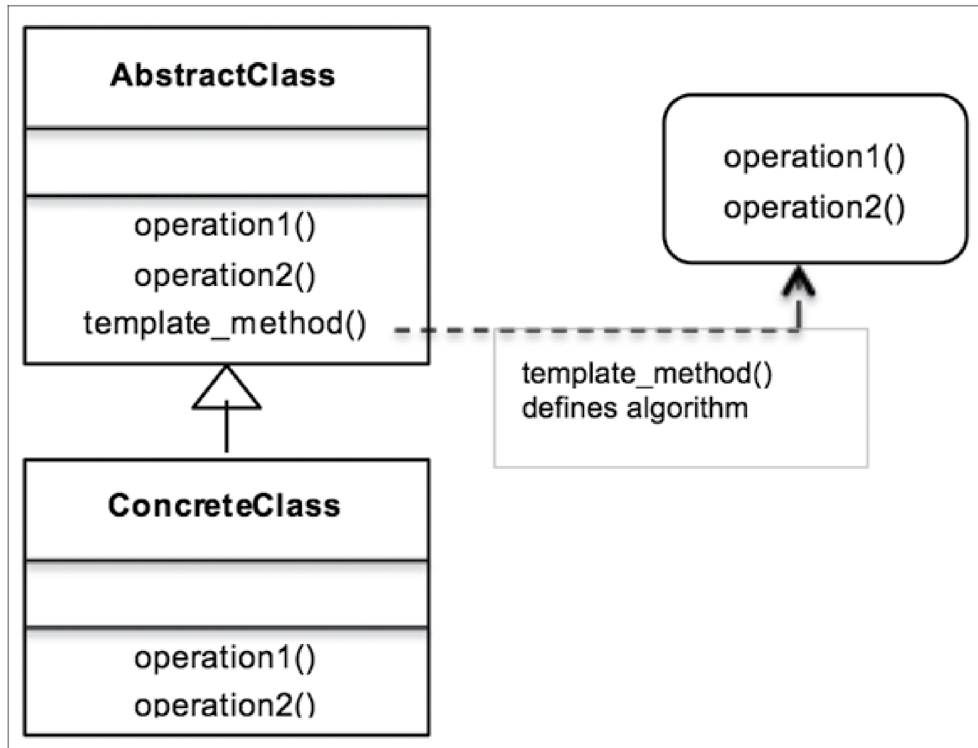
A saída do código anterior deve ter o seguinte aspecto:

Aprendendo Padrões
de Projeto em Python

Tire proveito da eficácia dos padrões de projeto (design patterns) em Python
para resolver problemas do mundo real em arquitetura e design de software

Um diagrama de classes UML para o padrão Template Method

Vamos compreender melhor o padrão Template Method com a ajuda de um diagrama UML:



Conforme discutimos na seção anterior, o padrão Template Method tem os seguintes participantes principais: a classe abstrata, a classe concreta, o método Template e o cliente. Vamos colocá-los em um diagrama UML e ver como é a aparência dessas classes.

Como podemos observar no diagrama UML, você perceberá que há quatro participantes principais nesse padrão:

- AbstractClass – define as operações ou os passos de um algoritmo com a ajuda de métodos abstratos. Esses passos são sobrescritos pelas subclasses concretas.
- template_method() – define o esqueleto do algoritmo. Vários passos, conforme definidos pelos métodos abstratos, são chamados no método Template para definir a sequência ou o algoritmo propriamente dito.
- ConcreteClass – implementa os passos (conforme definidos pelos métodos abstratos) específicos do algoritmo para a subclasse.

O exemplo de código a seguir serve para compreender o padrão com todos os participantes envolvidos:

```
from abc import ABCMeta, abstractmethod

class AbstractClass(metaclass=ABCMeta):
    def __init__(self):
        pass

    @abstractmethod
    def operation1(self):
        pass

    @abstractmethod
    def operation2(self):
        pass

    def template_method(self):
        print("Defining the Algorithm. Operation1 follows Operation2")
        self.operation2()
        self.operation1()


class ConcreteClass(AbstractClass):

    def operation1(self):
        print("My Concrete Operation1")

    def operation2(self):
        print("Operation 2 remains same")


class Client:
    def main(self):
        self.concreate = ConcreteClass()
        self.concreate.template_method()

client = Client()
client.main()
```

A saída do código anterior deve ter o seguinte aspecto:

```
Defining the Algorithm. Operation1 follows Operation2
Operation 2 remains same
My Concrete Operation1
```

Padrão Template Method no mundo real

Vamos considerar um cenário bem fácil de entender para implementar o padrão Template Method. Suponha o caso de uma agência de viagens, por exemplo, Dev Travels. Como essas agências geralmente funcionam? Elas definem várias viagens para diversos locais e apresentam um pacote de feriado para você. Um pacote é essencialmente uma viagem que você, como cliente, faz. Uma viagem tem detalhes como os lugares a ser visitados, o transporte a ser usado e outros fatores que definem o itinerário. Essa mesma viagem pode ser personalizada de modo diferente de acordo com as necessidades dos clientes. Isso pede o padrão Template Method, não é mesmo?

Considerações de design:

- No cenário anterior, baseado no diagrama UML, devemos criar uma interface AbstractClass que define uma viagem.
- A viagem deve conter vários métodos abstratos que definem o transporte usado, os lugares visitados em day1, day2 e day3, supondo que seja uma viagem de três dias em um fim de semana prolongado, além de definir a viagem de volta.
- O Método Template itinerary() define realmente o itinerário da viagem.
- Devemos definir ConcreteClasseS, que nos ajudarão a personalizar as viagens de acordo com as necessidades do cliente.

Vamos desenvolver uma aplicação em Python v3.5 e implementar o caso de uso anterior. Começaremos pela classe abstrata Trip:

- O objeto abstrato é representado pela classe Trip. É uma interface (classe-base abstrata de Python) que define detalhes como o transporte usado e os lugares a ser visitados em dias diferentes.

- setTransport é um método abstrato que deve ser implementado por ConcreteClass para definir o meio de transporte.
- Os métodos abstratos day1(), day2() e day3() definem os lugares a ser visitados no dia especificado.
- O Método Template itinerary() cria o itinerário completo (o algoritmo – nesse caso, a viagem). A sequência da viagem consiste em definir primeiro o meio de transporte e depois os lugares a ser visitados a cada dia e o returnHome.

O código a seguir implementa o cenário da Dev Travels:

```
from abc import abstractmethod, ABCMeta

class Trip(metaclass=ABCMeta):

    @abstractmethod
    def setTransport(self):
        pass

    @abstractmethod
    def day1(self):
        pass

    @abstractmethod
    def day2(self):
        pass

    @abstractmethod
    def day3(self):
        pass

    @abstractmethod
    def returnHome(self):
        pass

    def itinerary(self):
        self.setTransport()
        self.day1()
        self.day2()
        self.day3()
```

```
        self.returnHome()
```

Também desenvolvemos algumas classes que representam a classe concreta:

- Nesse caso, temos duas classes concretas principais – VeniceTrip e MaldivesTrip – que implementam a interface Trip.
- As classes concretas representam duas viagens diferentes feitas pelos turistas de acordo com sua escolha e seus interesses.
- Tanto VeniceTrip quanto MaldivesTrip implementam setTransport(), day1(), day2(), day3() e returnHome().

Vamos definir as classes concretas em código Python:

```
class VeniceTrip(Trip):
    def setTransport(self):
        print("Take a boat and find your way in the Grand Canal")

    def day1(self):
        print("Visit St Mark's Basilica in St Mark's Square")

    def day2(self):
        print("Appreciate Doge's Palace")

    def day3(self):
        print("Enjoy the food near the Rialto Bridge")

    def returnHome(self):
        print("Get souvenirs for friends and get back")


class MaldivesTrip(Trip):
    def setTransport(self):
        print("On foot, on any island, Wow!")

    def day1(self):
        print("Enjoy the marine life of Banana Reef")

    def day2(self):
        print("Go for the water sports and snorkelling")
```

```
    def day3(self):
        print("Relax on the beach and enjoy the sun")

    def returnHome(self):
        print("Dont feel like leaving the beach..")
```

Vamos agora falar da agência de viagem e dos turistas que querem ter férias incríveis:

- A classe TravelAgency representa o objeto Client nesse exemplo.
- Ela define o método arrange_trip(), que oferece aos clientes a opção de fazer uma viagem histórica ou uma viagem para a praia.
- Conforme a opção escolhida pelo turista, uma classe apropriada será instanciada.
- O objeto então chama o Método Template itinerary() e a viagem é organizada para os turistas de acordo com a opção dos clientes.

A seguir, apresentamos a implementação da agência Dev Travels e demonstramos como ela organiza a viagem conforme a opção do cliente:

```
class TravelAgency:
    def arrange_trip(self):
        choice = input("What kind of place you'd like to go historical or to a beach?")
        if choice == 'historical':
            self.trip = VeniceTrip()
            self.trip.itinerary()
        if choice == 'beach':
            self.trip = MaldivesTrip()
            self.trip.itinerary()

TravelAgency().arrange_trip()
```

A saída do código anterior deve ter o seguinte aspecto:

```
What kind of place you'd like to go historical or to a beach?beach
On foot, on any island, Wow!
Enjoy the marine life of Banana Reef
Go for the water sports and snorkelling
Relax on the beach and enjoy the sun
Dont feel like leaving the beach..
```

Se você optar por uma viagem histórica, esta será a saída gerada pelo

código:

```
What kind of place you'd like to go historical or to a beach?historical
Take a boat and find your way in the Grand Canal
Visit St Mark's Basilica in St Mark's Square
Appreciate Doge's Palace
Enjoy the food near the Rialto Bridge
Get souvenirs for friends and get back
```

Padrão Template Method – hooks

Um hook (gancho) é um método declarado na classe abstrata. Em geral, ele recebe uma implementação default. A ideia por trás dos hooks é dar a uma subclasse a capacidade de *se pendurar* no algoritmo sempre que for necessário. Não é obrigatório que a subclasse utilize hooks, e ela pode ignorá-los tranquilamente.

Por exemplo, no caso das bebidas, podemos adicionar um hook simples para ver se especiarias são necessárias para servir com o chá ou com o café conforme o desejo do cliente.

Outro exemplo de hook pode ser visto no caso da agência de viagem. Se tivermos alguns turistas mais idosos, talvez eles não queiram sair em todos os três dias da viagem, pois poderão se cansar facilmente. Nesse caso, podemos desenvolver um hook que garantirá que o segundo dia (day2) seja leve, o que significa que os turistas poderão ir a alguns locais próximos e retornar ao plano no terceiro dia (day3).

Basicamente, usamos métodos abstratos quando a subclasse deve fornecer a implementação, e o hook é usado quando sua implementação na subclasse é opcional.

Princípio de Hollywood e o Template Method

O princípio de Hollywood é o princípio de design sintetizado como: *não ligue para nós; nós ligaremos para você*. É proveniente da filosofia de Hollywood, segundo a qual as produtoras ligam para os atores caso haja

algum papel para eles.

No mundo da orientação a objetos, permitimos que componentes de baixo nível se associem ao sistema usando o princípio de Hollywood. No entanto, os componentes de alto nível determinam como e quando os sistemas de baixo nível são necessários. Em outras palavras, os componentes de alto nível tratam os componentes de baixo nível assim: *não ligue para nós; nós ligaremos para você*.

Isso se relaciona ao padrão Template Method no sentido de que é a classe abstrata de alto nível que organiza os passos para definir o algoritmo. Com base no algoritmo, as classes de baixo nível são solicitadas a definir a implementação concreta dos passos.

As vantagens e as desvantagens do padrão Template Method

O padrão Template Method oferece as seguintes vantagens:

- Como vimos antes neste capítulo, não há duplicação de código.
- A reutilização de código ocorre com o padrão Template Method, pois ele usa herança, e não composição. Somente alguns métodos precisam ser sobrescritos.
- A flexibilidade permite que as subclasses decidam como implementar os passos em um algoritmo.

As desvantagens dos padrões Template Method são:

- Depurar e compreender a sequência do fluxo no padrão Template Method às vezes pode ser confuso. Você pode acabar implementando um método que não deveria ser implementado ou deixando de implementar um método abstrato. A documentação e um tratamento de erros rigoroso devem ser feitos pelo programador.
- A manutenção do framework de templates pode ser um problema, pois alterações em qualquer nível (baixo ou alto) podem causar distúrbios na implementação. Por isso, a manutenção pode ser difícil

com o padrão Template Method.

Perguntas frequentes

P1. Um componente de baixo nível deveria ser proibido de chamar um método em um componente de mais alto nível?

R: Não, um componente de baixo nível definitivamente pode chamar o componente de mais alto nível por meio de herança. No entanto o programador deve garantir que não haja dependência circular, na qual os componentes de baixo nível e de alto nível sejam dependentes uns dos outros.

P2. O padrão Strategy (Estratégia) não é semelhante ao padrão Template?

R: Tanto o padrão Strategy quanto o padrão Template encapsulam algoritmos. O Template depende de herança, enquanto o Strategy usa composição. O padrão Template Method faz uma seleção de algoritmo em tempo de compilação usando subclasses, enquanto o padrão Strategy faz uma seleção em tempo de execução.

Resumo

Iniciamos o capítulo descrevendo o padrão de projeto Template Method e mostrando como ele é efetivamente usado na arquitetura de software.

Também vimos como o padrão de projeto Template Method é usado para encapsular o algoritmo e oferecer flexibilidade na implementação de comportamentos diferentes ao sobrescrever os métodos nas subclasses.

Além disso, conhecemos o padrão com um diagrama UML e vimos um exemplo de implementação de código com Python v3.5, juntamente com a explicação.

Também incluímos uma seção de Perguntas Frequentes que lhe ajudará a ter uma ideia melhor do padrão e de suas possíveis vantagens e

desvantagens.

Discutiremos agora um padrão composto no próximo capítulo – o padrão de projeto MVC.

CAPÍTULO 9

Modelo-Visão-Controlador – padrões compostos

No capítulo anterior, começamos com uma introdução ao padrão de projeto Template Method (Método Template), em que subclasses redefinem os passos concretos do algoritmo, conseguindo, assim, ter flexibilidade e permitindo a reutilização de código. Conhecemos o Template Method e vimos como ele é usado para construir o algoritmo com uma sequência de passos. Discutimos o diagrama UML, os prós e contras do padrão, aprendemos mais sobre ele na seção de Perguntas Frequentes e sintetizamos a discussão no final do capítulo.

Neste capítulo, discutiremos os padrões compostos. Seremos apresentados ao padrão de projeto MVC ou Modelo-Visão-Controlador (Model-View-Controller) e discutiremos como ele é usado no desenvolvimento de aplicações de software. Além disso, trabalharemos com um caso de uso de exemplo e faremos a sua implementação com Python v3.5.

Neste capítulo, os seguintes tópicos serão abordados de forma sucinta:

- uma introdução aos padrões compostos e ao Modelo-Visão-Controlador;
- o padrão MVC e o seu diagrama UML;
- um caso de uso do mundo real implementado com Python v3.5;
- prós e contras do padrão MVC;

- perguntas frequentes.

No final do capítulo, faremos uma síntese de toda a discussão – considere isso como os pontos principais a ser lembrados.

Uma introdução aos padrões compostos

Ao longo deste livro, exploramos vários padrões de projeto. Como vimos, os padrões de projeto são classificados em três categorias principais: padrões estruturais, de criação e comportamentais. Também conhecemos cada um deles com exemplos.

No entanto, em implementação de software, os padrões não funcionam isoladamente. Todo design ou solução de software jamais é implementado com apenas um padrão de projeto. Na verdade, os padrões são usados frequentemente em conjunto e são combinados para conseguir uma dada solução de design. Conforme definido pela GoF, "*um padrão composto combina dois ou mais padrões em uma solução que resolve um problema recorrente ou genérico*". Um padrão composto não é um conjunto de padrões que trabalha em conjunto – é uma solução geral para um problema.

Veremos agora o padrão composto Modelo-Visão-Controlador (Model-View-Controller). É o melhor exemplo de padrão composto e tem sido usado em muitas soluções de design ao longo dos anos.

Padrão Modelo-Visão-Controlador

O MVC é um padrão de software para implementar interfaces de usuário e uma arquitetura que pode ser facilmente modificada e mantida. Essencialmente, o padrão MVC diz respeito à separação da aplicação em três partes básicas: modelo, visão e controlador. Essas três partes estão interconectadas e ajudam a separar os modos como a informação é representada da forma como ela é apresentada.

É assim que o padrão MVC funciona: o modelo representa os dados e a

lógica de negócios (como a informação é armazenada e consultada), a visão nada mais é que a representação (como ela é apresentada) dos dados, e o controlador é a cola que une ambos, ou seja, é a parte que direciona o modelo e a visão para que se comportem de determinada maneira de acordo com as necessidades de um usuário. É interessante observar que a visão e o controlador são dependentes do modelo, mas não o contrário. Isso ocorre principalmente porque um usuário está preocupado com os dados. Podemos trabalhar com os modelos de forma independente, e esse é o aspecto essencial do padrão MVC.

Considere o caso de um site. Esse é um dos exemplos clássicos para descrever o padrão MVC. O que acontece em um site? Você clica em um botão, algumas operações ocorrem e você passa a ver o que desejava. Como isso acontece?

- Você é o usuário e interage com a visão. A visão é a página web apresentada. Você clica nos botões da visão e ela informa ao controlador o que deve ser feito.
- Os controladores obtêm a entrada da visão e a envia ao modelo. O modelo é manipulado com base nas ações do usuário.
- Os controladores também podem pedir à visão para mudar conforme a ação recebida do usuário, por exemplo, alterar os botões, apresentar elementos adicionais na UI, e assim por diante.
- O modelo notifica a alteração de estado à visão. Isso pode ser feito com base em algumas alterações internas ou por acionamentos externos, como cliques em um botão.
- A visão então exibe o estado que ela obtém diretamente do modelo. Por exemplo, se um usuário fizer login no site, uma visão de painel poderá ser apresentada a ele (após o login). Todos os detalhes que devem ser preenchidos no painel são fornecidos à visão pelo modelo.

O padrão de projeto MVC trabalha com os seguintes termos: Modelo, Visão, Controlador e Cliente.

- **Modelo**: declara uma classe para armazenar e manipular os dados.

- **Visão**: declara uma classe para construir interfaces de usuário e fazer exibições de dados.
- **Controlador**: declara uma classe que conecta o modelo e a visão.
- **Usuário**: declara uma classe que solicita determinados resultados com base em certas ações.

A imagem a seguir explica o fluxo do padrão MVC:

Aprendendo Padrões de Projeto em Python

Tire proveito da eficácia dos padrões de projeto (design patterns) em Python para resolver problemas do mundo real em arquitetura e design de software

Para discutir o padrão MVC na terminologia do desenvolvimento de software, vamos dar uma olhada nas classes principais envolvidas nesse padrão:

- A classe `model` é usada para definir todas as operações que ocorrem nos dados (por exemplo, criar, modificar e apagar), além de oferecer métodos para usar os dados.
- A classe `view` é uma representação da interface de usuário. Ela terá métodos que nos ajudam a construir interfaces web ou GUI conforme o contexto e as necessidades da aplicação. Ela não deve conter nenhuma lógica própria, mas deve apenas exibir os dados que receber.
- A classe `controller` é usada para receber dados da requisição e enviá-los a outras partes do sistema. Tem métodos usados para encaminhar as requisições.

O padrão MVC é usado nos seguintes casos:

- quando há necessidade de mudar a apresentação sem alterações na lógica de negócios;

- quando vários controladores podem ser usados para trabalhar com várias visões para mudar a representação na interface do usuário;
- novamente, o modelo pode ser modificado sem alterações na visão, pois eles podem trabalhar de forma independente.

Para resumir, os principais objetivos do padrão MVC são:

- manter os dados e a sua apresentação separados;
- facilitar a manutenção da classe e de sua implementação;
- ter flexibilidade para mudar o modo como os dados são armazenados e exibidos; ambos são independentes e, portanto, têm flexibilidade para mudar.

Vamos dar uma olhada no modelo, na visão e no controlador em detalhes, conforme discutido no livro *Learning Python Design Patterns* de *Gennadiy Zlobin*, publicado pela *Packt Publishing*.

Modelo – conhecimento da aplicação

O modelo é a pedra angular de uma aplicação, pois independe da visão e do controlador. A visão e o controlador, por sua vez, são dependentes do modelo.

O modelo também fornece os dados solicitados pelo cliente. Geralmente nas aplicações, o modelo é representado pelas tabelas do banco de dados que armazenam e devolvem informações. O modelo tem estados e métodos para alterar esses estados, mas não tem ciência de como os dados serão vistos pelo cliente.

É crucial que o modelo permaneça consistente quando houver várias operações; caso contrário, o cliente poderá obter dados corrompidos ou exibir dados desatualizados, o que é extremamente indesejável.

Como o modelo é totalmente independente, os desenvolvedores que trabalham nessa parte podem manter o foco na manutenção sem ter as alterações mais recentes da visão.

Visão – a aparência

A visão é uma representação dos dados na interface vista pelo cliente. A visão pode ser desenvolvida de forma independente, mas não deve conter nenhuma lógica complexa. A lógica deve continuar no controlador ou no modelo.

No mundo atual, as visões devem ser suficientemente flexíveis e apropriadas para diversas plataformas, por exemplo, desktop, dispositivos móveis, tablets e vários tamanhos de tela.

As visões devem evitar interagir diretamente com os bancos de dados e devem contar com os modelos para obter os dados necessários.

Controlador – a cola

O controlador, conforme o nome sugere, controla a interação do usuário na interface. Quando o usuário clicar em determinados elementos da interface, conforme a interação (clique no botão ou um toque), o controlador fará uma chamada ao modelo que, por sua vez, criará, atualizará ou apagará os dados.

Os controladores também passam os dados para a visão, que renderiza a informação de modo que o usuário possa visualizá-la na interface.

O controlador não deve fazer chamadas ao banco de dados nem envolver-se na apresentação dos dados. Ele deve atuar como uma cola entre o modelo e a visão, e deve ser o mais enxuto possível.

Vamos agora entrar em ação e desenvolver uma aplicação de exemplo. O código Python mostrado a seguir implementa o padrão de projeto MVC. Considere que queremos desenvolver uma aplicação que informe os serviços de marketing oferecidos por uma empresa que atue na nuvem, incluindo recursos de email, SMS e voz.

Inicialmente, desenvolvemos a classe `model` (Modelo), que define os serviços oferecidos pelo produto, isto é, email, SMS e voz. Cada um desses serviços tem preços especificados, por exemplo, 1.000 emails custam 2 dólares ao cliente; para 1.000 mensagens, serão cobrados 10 dólares, e 15 dólares a cada 1.000 mensagens de voz. Assim, o modelo representa os dados relacionados aos serviços e preços do produto.

Em seguida, definimos a classe `view` (Visão), que fornece um método para apresentar as informações ao cliente. Os métodos são `list_services()` e `list_pricing()`; conforme os nomes sugerem, um método é usado para exibir os serviços oferecidos pelo produto e o outro lista os preços dos serviços.

Então definimos a classe `Controller`, que especifica dois métodos: `get_services()` e `get_pricing()`. Cada um desses métodos consulta o modelo e obtém os dados. Os dados são então passados para a visão e, assim, são apresentados ao cliente.

A classe `Client` instancia o controlador. O objeto `controller` é usado para chamar os métodos apropriados de acordo com a requisição do cliente:

```
class Model(object):
    services = {
        'email': {'number': 1000, 'price': 2,},
        'sms': {'number': 1000, 'price': 10,},
        'voice': {'number': 1000, 'price': 15,},
    }


class View(object):
    def list_services(self, services):
        for svc in services:
            print(svc, ' ')

    def list_pricing(self, services):
        for svc in services:
            print("For" , Model.services[svc]['number'],
                  svc, "message you pay $",
                  Model.services[svc]['price'])


class Controller(object):
    def __init__(self):
        self.model = Model()
        self.view = View()

    def get_services(self):
```

```
        services = self.model.services.keys()
        return(self.view.list_services(services))

    def get_pricing(self):
        services = self.model.services.keys()
        return(self.view.list_pricing(services))


class Client(object):
    controller = Controller()
    print("Services Provided:")
    controller.get_services()
    print("Pricing for Services:")
    controller.get_pricing()
```

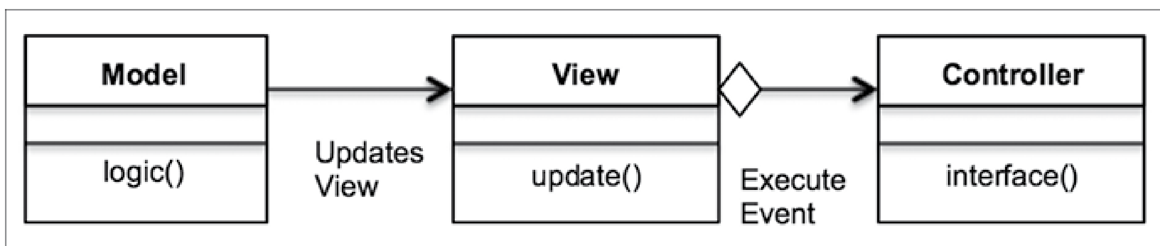
A seguir, apresentamos a saída do código anterior:

```
Services Provided:
sms
email
voice
Pricing for Services:
For 1000 sms message you pay $ 10
For 1000 email message you pay $ 2
For 1000 voice message you pay $ 15
```

Um diagrama de classes UML para o padrão de projeto MVC

Vamos agora entender melhor o padrão MVC com a ajuda do diagrama UML a seguir.

Conforme discutimos nas seções anteriores, o padrão MVC tem os seguintes participantes principais: as classes Model, View e Controller.



No diagrama UML, podemos ver três classes principais nesse padrão:

- A classe Model – define a lógica de negócios ou as operações associadas a determinadas tarefas do cliente.
- A classe View – define a visão ou a representação vista pelo cliente. O modelo apresenta os dados à visão de acordo com a lógica de negócios.
- A classe Controller – é essencialmente uma interface que está entre a visão e o modelo. Quando o cliente executa determinadas ações, o controlador direciona a consulta da visão para o modelo.

O exemplo de código a seguir serve para compreender o padrão com todos os participantes envolvidos:

```
class Model(object):
    def logic(self):
        data = 'Got it!'
        print("Model: Crunching data as per business logic")
        return data


class View(object):
    def update(self, data):
        print("View: Updating the view with results: ", data)


class Controller(object):
    def __init__(self):
        self.model = Model()
        self.view = View()

    def interface(self):
        print("Controller: Relayed the Client asks")
        data = self.model.logic()
        self.view.update(data)


class Client(object):
    print("Client: asks for certain information")
    controller = Controller()
```

```
controller.interface()
```

A seguir, apresentamos a saída do código anterior:

```
Client: asks for certain information
Controller: Relayed the Cient asks
Model: Crunching data as per business logic
View: Updating the view with results:  Got it!
```

Padrão MVC no mundo real

Nossos bons e velhos frameworks de aplicações web são baseados nas filosofias do MVC. Tome como exemplo o Django ou o Rails (Ruby): eles estruturam seus projetos no formato Modelo-Visão-Controlador, embora sejam representados como MTV (Modelo, Template, Visão), em que o modelo é o banco de dados, os templates são as visões e os controladores são as visões/rotas.

Como exemplo, vamos considerar o framework de aplicações web Tornado (*http://www.tornadoweb.org/en/stable/*) para desenvolver uma aplicação single-page (página única). Essa aplicação é usada para administrar as tarefas de um usuário, e esse usuário tem permissões para adicionar, atualizar e apagar tarefas.

Vamos ver as considerações de design:

- Começaremos pelos controladores. No Tornado, os controladores foram definidos como visões/rotas da aplicação. Precisamos definir várias visões, como listagem das tarefas, criação de novas tarefas, encerramento de tarefas e tratamento de uma operação caso uma requisição não possa ser servida.
- Devemos também definir os modelos, isto é, as operações de banco de dados para listar, criar ou remover tarefas.
- Por fim, as visões são representadas pelos templates no Tornado. Com base em nossa aplicação, precisaremos de um template para mostrar, criar ou apagar tarefas, além de um template caso um URL não seja encontrado.

Módulos

Utilizaremos os módulos a seguir nessa aplicação:

- Tornado versão 4.3
- SQLite3 versão 2.6.0

Vamos começar importando os módulos Python em nossa aplicação:

```
importtornado
import tornado.web
import tornado.ioloop
import tornado.httpserver
import sqlite3
```

O código a seguir representa as operações de banco de dados, essencialmente, os modelos no MVC. No Tornado, as operações de banco de dados são executadas com handlers diferentes. Os handlers executam operações no banco de dados conforme a rota requisitada pelo usuário na aplicação web. Nesse caso, discutiremos os quatro handlers que criamos nesse exemplo:

- IndexHandler – devolve todas as tarefas armazenadas no banco de dados. Devolve um dicionário com as tarefas como chaves. Executa a operação SELECT no banco de dados para obter essas tarefas.
- NewHandler – conforme sugere o nome, é útil para adicionar novas tarefas. Verifica se há uma chamada a POST para criar uma nova tarefa e executa uma operação INSERT no banco de dados.
- UpdateHandler – é útil para marcar uma tarefa como concluída ou reabrir uma determinada tarefa. Nesse caso, a operação de banco de dados UPDATE é executada para definir uma tarefa com o status aberto/fechado.
- DeleteHandler – remove uma dada tarefa do banco de dados. Após ter sido apagada, a tarefa não será mais visível na lista de tarefas.

Desenvolvemos também um método _execute() que aceita uma consulta SQLite como entrada e executa a operação de banco de dados solicitada. Esse método executa as seguintes operações no banco de dados SQLite:

- cria uma conexão de banco de dados com o SQLite;
- obtém o objeto cursor;
- usa o objeto cursor para fazer uma transação;
- faz commit da consulta;
- encerra a conexão.

Vamos dar uma olhada nos handlers na implementação Python:

```
class IndexHandler(tornado.web.RequestHandler):
    def get(self):
        query = "select * from task"
        todos = _execute(query)
        self.render('index.html', todos=todos)

class NewHandler(tornado.web.RequestHandler):
    def post(self):
        name = self.get_argument('name', None)
        query = "create table if not exists task (id INTEGER \
            PRIMARY KEY, name TEXT, status NUMERIC) "
        _execute(query)
        query = "insert into task (name, status) \
            values ('%s', %d) " %(name, 1)
        _execute(query)
        self.redirect('/')

    def get(self):
        self.render('new.html')


class UpdateHandler(tornado.web.RequestHandler):
    def get(self, id, status):
        query = "update task set status=%d where \
            id=%s" %(int(status), id)
        _execute(query)
        self.redirect('/')


class DeleteHandler(tornado.web.RequestHandler):
    def get(self, id):
        query = "delete from task where id=%s" % id
```

```
_execute(query)
self.redirect('/')
```

Se observar esses métodos, você notará que há algo chamado `self.render()`. Essencialmente, isso representa as visões no MVC (templates no framework Tornado). Temos três templates principais:

- `index.html` – é um template para listar todas as tarefas;
- `new.html` – é a visão para criar uma nova tarefa;
- `base.html` – é o template-base do qual outros templates herdam.

Considere o código a seguir:

```
base.html
<html>
<!DOCTYPE>
<html>
<head>
    {% block header %}{% end %}
</head>
<body>
    {% block body %}{% end %}
</body>
</html>

index.html

{% extends 'base.html' %}
<title>ToDo</title>
{% block body %}
<h3>Your Tasks</h3>
<table border="1" >
<tralign="center">
<td>Id</td>
<td>Name</td>
<td>Status</td>
<td>Update</td>
<td>Delete</td>
</tr>
  {% for todo in todos %}
<tralign="center">
```

```
<td>{{todo[0]}}</td>
<td>{{todo[1]}}</td>
        {% if todo[2] %}
<td>Open</td>
        {% else %}
<td>Closed</td>
        {% end %}
        {% if todo[2] %}
<td><a href="/todo/update/{{todo[0]}}/0">Close Task</a></td>
        {% else %}
<td><a href="/todo/update/{{todo[0]}}/1">Open Task</a></td>
        {% end %}
<td><a href="/todo/delete/{{todo[0]}}">X</a></td>
</tr>
  {% end %}
</table>

<div>
<h3><a href="/todo/new">Add Task</a></h3>
</div>
{% end %}
```

new.html

```
{% extends 'base.html' %}
<title>ToDo</title>
{% block body %}
<div>
<h3>Add Task to your List</h3>
<form action="/todo/new" method="post" id="new">
<p><input type="text" name="name" placeholder="Enter task"/>
<input type="submit" class="submit" value="add" /></p>
</form>
</div>
{% end %}
```

No Tornado, também temos as rotas da aplicação, que são controladores no MVC. Temos quatro rotas de aplicação nesse exemplo:

- / é a rota para listar todas as tarefas;
- /todo/new é a rota para criar novas tarefas;

- /todo/update é a rota para atualizar o status da tarefa como aberto/fechado;
- /todo/delete é a rota para apagar uma tarefa concluída.

O código de exemplo é:

```
class RunApp(tornado.web.Application):
    def __init__(self):
        Handlers = [
            (r'/', IndexHandler),
            (r'/todo/new', NewHandler),
            (r'/todo/update/(\d+)/status/(\d+)', UpdateHandler),
            (r'/todo/delete/(\d+)', DeleteHandler),
        ]
        settings = dict(
            debug=True,
            template_path='templates',
            static_path="static",
        )
        tornado.web.Application.__init__(self, Handlers, **settings)
```

Também temos configurações da aplicação e podemos iniciar o servidor web HTTP para executar a aplicação:

```
if__name__ == '__main__':
    http_server = tornado.httpserver.HTTPServer(RunApp())
    http_server.listen(5000)
    tornado.ioloop.IOLoop.instance().start()
```

Quando executamos o programa Python:

1. O servidor é iniciado e executa na porta 5000. As visões, os templates e os controladores apropriados são configurados.
2. Ao acessar *http://localhost:5000/*, podemos ver a lista de tarefas; a imagem de tela a seguir mostra a saída no navegador:

Your Tasks

Id	Name	Status	Update	Delete
1	New Task	Open	Close Task	X
2	Wash clothes	Closed	Open Task	X
3	Cook food	Open	Close Task	X
4	Thats enough	Open	Close Task	X
5	Wow! A new Task	Open	Close Task	X

Add Task

3. Também podemos adicionar uma nova tarefa. Após clicar no botão **ADD**, uma nova tarefa será adicionada. Na imagem de tela a seguir, uma nova tarefa **Write the New Chapter** é adicionada e incluída na lista de tarefas:

Quando fornecemos a nova tarefa e clicamos no botão ADD, a tarefa é adicionada à lista de tarefas existente:

Your Tasks

Id	Name	Status	Update	Delete
1	New Task	Open	Close Task	X
2	Wash clothes	Closed	Open Task	X
3	Cook food	Open	Close Task	X
4	Thats enough	Open	Close Task	X
5	Wow! A new Task	Open	Close Task	X
6	Write the New Chapter	Open	Close Task	X

Add Task

4. Também podemos encerrar as tarefas a partir da UI. Por exemplo, atualizamos a tarefa **Cook food** e a lista é atualizada. Podemos reabrir a tarefa se quisermos:

Your Tasks

Id	Name	Status	Update	Delete
1	New Task	Open	Close Task	X
2	Wash clothes	Closed	Open Task	X
3	Cook food	Closed	Open Task	X
4	Thats enough	Open	Close Task	X
5	Wow! A new Task	Open	Close Task	X
6	Write the New Chapter	Open	Close Task	X

5. Podemos também apagar uma tarefa. Nesse caso, apagamos a primeira tarefa, **New Task**, e a lista de tarefas é atualizada com a tarefa removida:

Your Tasks

Id	Name	Status	Update	Delete
2	Wash clothes	Closed	Open Task	X
3	Cook food	Closed	Open Task	X
4	Thats enough	Open	Close Task	X
5	Wow! A new Task	Open	Close Task	X
6	Write the New Chapter	Open	Close Task	X

Vantagens do padrão MVC

A seguir, apresentamos as vantagens do padrão MVC:

- Com o MVC, os desenvolvedores podem separar a aplicação de software em três partes principais – modelo, visão e controlador. Isso ajuda a simplificar a manutenção, garante um baixo acoplamento e reduz a complexidade.
- O MVC permite alterações independentes no frontend com poucas mudanças, ou nenhuma, na lógica do backend; desse modo, os esforços de desenvolvimento ainda podem ocorrer de forma independente.
- De modo semelhante, os modelos ou a lógica de negócios podem ser alterados sem qualquer mudança na visão.
- Além disso, o controlador pode ser alterado sem qualquer impacto nas visões ou nos modelos.
- O MVC também ajuda na contratação de pessoas com capacidades específicas, por exemplo, engenheiros de plataforma e de UI, que podem trabalhar de forma independente em suas áreas de especialização.

Perguntas frequentes

P1. O MVC não é um padrão? Por que ele é chamado de padrão

composto?

R: Os padrões compostos são essencialmente grupos de padrões reunidos para resolver problemas maiores de design no desenvolvimento de aplicações de software. O MVC é o padrão composto mais popular e mais amplamente utilizado. Por ser tão amplamente usado e ser confiável, ele é tratado por si só como um padrão.

P2. O MVC é usado somente em sites?

R: Não, um site é o melhor exemplo para descrever o MVC. No entanto o MVC pode ser usado em várias áreas, como aplicações de GUI ou em qualquer outro local em que seja necessário ter baixo acoplamento e separar os componentes de forma independente. Exemplos típicos de MVC incluem blogs, aplicações de banco de dados de filmes e aplicações web para streaming de vídeo. Embora o MVC seja conveniente em muitos lugares, será um exagero usá-lo em páginas de destino (landing pages), conteúdo de marketing ou em aplicações single-page rápidas.

P3. Várias visões podem trabalhar com vários modelos?

R: Sim, com frequência, você acabará em uma situação em que os dados precisam ser agrupados a partir de vários modelos e apresentados em uma única visão. Um mapeamento de um para um (one-to-one) é raro no mundo das aplicação web atuais.

Resumo

Iniciamos o capítulo descrevendo os padrões compostos e vimos o padrão Modelo-Visão-Controlador e como ele é efetivamente usado na arquitetura de software. Em seguida, vimos como o padrão MVC é utilizado para garantir um baixo acoplamento e manter um framework de várias camadas para o desenvolvimento de tarefas independentes.

Também conhecemos o padrão com um diagrama UML e vimos um exemplo de implementação de código com Python v3.5, juntamente com a explicação. Também incluímos uma seção de Perguntas Frequentes

que lhe ajudará a ter mais ideias sobre o padrão e suas possíveis vantagens e desvantagens.

No próximo capítulo, discutiremos o padrão de projeto State. Até lá!

CAPÍTULO 10

Padrão de projeto State

Neste capítulo, discutiremos o padrão de projeto State (Estado). Assim como os padrões de projeto Command (Comando) ou Template, o padrão State se enquadra na categoria dos padrões comportamentais. Seremos apresentados ao padrão de projeto State e discutiremos como ele é usado no desenvolvimento de aplicações de software. Além disso, trabalharemos com um caso de uso de exemplo, que é um cenário do mundo real, e faremos a sua implementação com Python v3.5.

Neste capítulo, os seguintes tópicos serão abordados de forma sucinta:

- introdução ao padrão de projeto State;
- o padrão de projeto State e o seu diagrama UML;
- um caso de uso do mundo real implementado com Python v3.5;
- as vantagens e desvantagens do padrão State.

No final deste capítulo, você apreciará a aplicação e o contexto do padrão de projeto State.

Definindo o padrão de projeto State

Os padrões comportamentais têm como foco as responsabilidades de um objeto. Eles lidam com a interação entre objetos para alcançar funcionalidades mais complexas. O padrão de projeto State é um padrão Comportamental que, às vezes, também é chamado de padrão de objetos para estados. Nesse padrão, um objeto pode encapsular vários comportamentos de acordo com o seu estado interno. Um padrão State também é considerado como uma maneira de um objeto alterar o seu comportamento em tempo de execução.

Alterar o comportamento em tempo de execução é algo no qual Python é excelente!

Por exemplo, considere o caso de um rádio simples. Um rádio tem canais AM/FM (uma chave para alternar) e um botão de scan para procurar vários canais FM/AM. Quando um usuário liga o rádio, seu estado-base já está definido (vamos supor que esteja definido para FM). Ao clicar no botão Scan, o rádio é sintonizado em várias frequências ou canais válidos de FM. Quando o estado-base é alterado para AM, o botão de scan ajuda o usuário a sintonizar vários canais de AM. Então, de acordo com o estado-base (AM/FM) do rádio, o comportamento do botão de scan muda dinamicamente, sintonizando canais AM ou FM.

Desse modo, o padrão State permite que um objeto altere o seu comportamento quando o seu estado interno muda. Parecerá que o próprio objeto alterou a sua classe. O padrão de projeto State é usado para desenvolver Máquinas de Estado Finitas (Finite State Machines) e ajuda a acomodar Ações de Transição de Estados (State Transition Actions).

Compreendendo o padrão de projeto State

Os padrões de projeto State funcionam com a ajuda de três participantes principais:

- State – é considerada uma interface que encapsula o comportamento do objeto. Esse comportamento está associado ao estado do objeto.
- ConcreteState – é uma subclasse que implementa a interface State. ConcreteState implementa o comportamento propriamente dito associado ao estado particular do objeto.
- Context – define a interface de interesse dos clientes. Context também mantém uma instância da subclasse ConcreteState que, internamente, define a implementação do estado particular do objeto.

Vamos dar uma olhada na implementação do código estrutural do padrão de projeto State com esses três participantes. Nessa

implementação de código, definimos uma interface State que tem um método abstrato Handle(). As classes ConcreteState – ConcreteStateA e ConcreteStateB – implementam a interface State e, desse modo, definem os métodos Handle() específicos das classes ConcreteState. Então, quando a classe Context é definida para um estado, o método Handle() da ConcreteClass desse estado é chamado. No exemplo a seguir, como Context é definido com stateA, o método ConcreteStateA.Handle() é chamado e exibe ConcreteStateA:

```
from abc import abstractmethod, ABCMeta

class State(metaclass=ABCMeta):

    @abstractmethod
    def Handle(self):
        pass


class ConcreteStateB(State):
    def Handle(self):
        print("ConcreteStateB")


class ConcreteStateA(State):
    def Handle(self):
        print("ConcreteStateA")


class Context(State):

    def __init__(self):
        self.state = None

    def getState(self):
        return self.state

    def setState(self, state):
        self.state = state

    def Handle(self):
        self.state.Handle()
```

```
context = Context()
stateA = ConcreteStateA()
stateB = ConcreteStateB()

context.setState(stateA)
context.Handle()
```

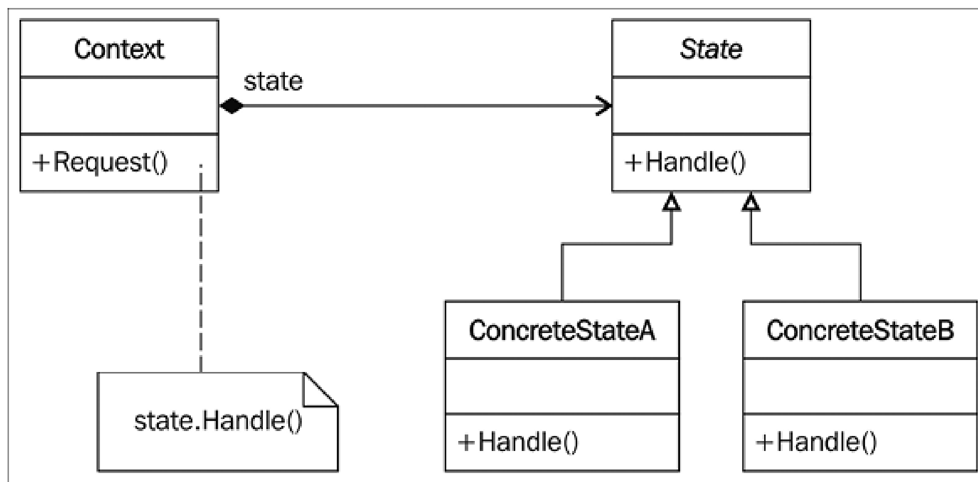
Veremos a saída a seguir:

Aprendendo Padrões
de Projeto em Python

Tire proveito da eficácia dos padrões de projeto (design patterns) em Python
para resolver problemas do mundo real em arquitetura e design de software

Compreendendo o padrão de projeto State com um diagrama UML

Como vimos na seção anterior, há três participantes principais no diagrama UML: State, ConcreteState e Context. Nesta seção, tentaremos apresentá-los em um diagrama de classes UML.



Vamos entender os elementos do diagrama UML em detalhes.

- State: é uma interface que define o método abstrato Handle(). O método Handle() deve ser implementado por ConcreteState.
- ConcreteState: nesse diagrama UML, definimos duas ConcreteClasses – ConcreteStateA e ConcreteStateB. Elas implementam o método Handle() e

definem a ação propriamente dita a ser executada de acordo com a mudança em State.

- Context: é uma classe que aceita a requisição do cliente. Também mantém uma referência ao estado atual do objeto. Com base na requisição, o comportamento concreto será chamado.

Um exemplo simples do padrão de projeto State

Vamos entender todos os três participantes com um exemplo simples. Suponha que queremos implementar um controle remoto de TV com um botão simples para realizar ações de ligar/desligar. Se a TV estiver ligada, o botão do controle remoto desligará a TV e vice-versa. Nesse caso, a interface State definirá o método (por exemplo, doThis()) para executar ações como ligar/desligar a TV. Também precisamos definir ConcreteClass para estados diferentes. Nesse exemplo, temos dois estados principais, StartState e StopState, que indicam quando a TV está ligada e o estado em que ela está desligada, respectivamente.

Para esse cenário, a classe TVContext implementará a interface State e manterá uma referência ao estado atual. Com base na requisição, TVContext a encaminha para ConcreteState, que implementa o comportamento propriamente dito (para um dado estado) e executa a ação necessária. Então, nesse caso, o estado-base é StartState (conforme definido anteriormente) e a requisição recebida pela classe TVContext é desligar a TV. A classe TVContext compreende a necessidade e, de acordo com ela, encaminha a requisição para a classe concreta StopState, que, por sua vez, chama o método doThis() para desligar a TV:

```
from abc import abstractmethod, ABCMeta

class State(metaclass=ABCMeta):

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

```
class StartState (State):
    def doThis(self):
        print("TV Switching ON..")


class StopState (State):
    def doThis(self):
        print("TV Switching OFF..")


class TVContext(State):

    def __init__(self):
        self.state = None

    def getState(self):
        return self.state

    def setState(self, state):
        self.state = state

    def doThis(self):
        self.state.doThis()


context = TVContext()
context.getState()
start = StartState()
stop = StopState()

context.setState(stop)
context.doThis()
```

Eis a saída do código anterior:

```
TV Switching OFF..
```

O padrão de projeto State com

implementação em Python v3.5

Vamos agora dar uma olhada em um caso de uso do mundo real para o padrão de projeto State. Pense em um sistema de computador (desktop/notebook). Ele pode ter vários estados como On, Off, Suspend ou Hibernate. Se quiséssemos manifestar esses estados com a ajuda do padrão de projeto State, como faríamos isso?

Vamos imaginar que começaríamos pela interface ComputerState:

- O estado deve definir dois atributos, que são name e allowed. O atributo name representa o estado do objeto, e allowed é uma lista que define os estados em que o objeto pode estar.
- O estado deve definir um método switch() que mudará o estado do objeto (nesse caso, do computador).

Vamos ver o código de implementação da interface ComputerState:

```
class ComputerState(object):
    name = "state"
    allowed = []

    def switch(self, state):
        if state.name in self.allowed:
            print('Current:',self,' => switched to new state',state.name)
            self.__class__ = state
        else:
            print('Current:',self,' => switching to',state.name,'not possible.')

    def __str__(self):
        return self.name
```

Vamos agora dar uma olhada em ConcreteState, que implementa a interface State. Definiremos quatro estados:

- On – liga o computador. Os estados permitidos aqui são Off, Suspend e Hibernate.
- Off – desliga o computador. O estado permitido, nesse caso, é apenas On.
- Hibernate – esse estado coloca o computador em modo de hibernação.

Quando estiver nesse estado, o computador poderá ser somente ligado.

- Suspend – esse estado suspende o computador; depois que é suspenso, o computador pode ser ligado.

Vamos agora dar uma olhada no código:

```
class Off(ComputerState):
    name = "off"
    allowed = ['on']

class On(ComputerState):
    name = "on"
    allowed = ['off','suspend','hibernate']

class Suspend(ComputerState):
    name = "suspend"
    allowed = ['on']

class Hibernate(ComputerState):
    name = "hibernate"
    allowed = ['on']
```

Vamos explorar agora a classe de contexto (Computer). O contexto executa duas tarefas principais:

- __init__() – esse método define o estado-base do computador.
- change() – esse método alterará o estado do objeto, e a mudança propriamente dita no comportamento é implementada pelas classes ConcreteState (on, off, suspend e hibernate).

Eis a implementação dos métodos anteriores:

```
class Computer(object):
    def __init__(self, model='HP'):
        self.model = model
        self.state = Off()

    def change(self, state):
        self.state.switch(state)
```

O código a seguir implementa o cliente. Criamos o objeto da classe

Computer (Context) e lhe passamos um estado. O estado pode ser On, Off, Suspend e Hibernate. De acordo com o novo estado, o contexto chama o seu método change(state), que alterará o estado do computador:

```
if __name__ == "__main__":
    comp = Computer()
    # Liga
    comp.change(On)
    # Desliga
    comp.change(Off)

    # Liga novamente
    comp.change(On)
    # Suspende
    comp.change(Suspend)
    # Tenta hibernar – não pode!
    comp.change(Hibernate)
    # Liga novamente
    comp.change(On)
    # Finalmente desliga
    comp.change(Off)
```

Agora podemos observar a saída a seguir:

```
Current: off  => switched to new state on
Current: on  => switched to new state off
Current: off  => switched to new state on
Current: on  => switched to new state suspend
Current: suspend  => switching to hibernate not possible
Current: suspend  => switched to new state on
Current: on  => switched to new state off
```

__class__ é um atributo embutido de todas as classes. É uma referência à classe. Por exemplo, self.__class__.__name__ representa o nome da classe. Nesse exemplo, usamos o atributo __class__ de Python para alterar State. Assim, quando passamos o estado para o método change(), a classe dos objetos muda dinamicamente em tempo de execução. O código de comp.change(On) muda o estado do objeto para On e, posteriormente, há mudanças para diferentes estados como Suspend, Hibernate e Off.

Vantagens e desvantagens do

padrão State

Eis as vantagens do padrão de projeto State:

- No padrão de projeto State, o comportamento de um objeto é resultado da função de seu estado, e o comportamento muda em tempo de execução de acordo com o estado. Isso elimina a dependência da lógica condicional de if/else ou de switch/case. Por exemplo, no cenário do controle remoto de TV, poderíamos ter implementado também o comportamento simplesmente escrevendo uma classe e o método, que pediria um parâmetro e executaria uma ação (ligar/desligar a TV) com um bloco if/else.
- Com o padrão State, as vantagens de implementar um comportamento polimórfico são evidentes, além de ser mais fácil adicionar estados para dar suporte a novos comportamentos.
- O padrão de projeto State também melhora a **Coesão**, pois comportamentos específicos de estados são agregados nas classes ConcreteState, que são colocadas em um só lugar no código.
- Com o padrão de projeto State, é muito fácil acrescentar um comportamento simplesmente adicionando mais uma classe ConcreteState. O padrão State, desse modo, aumenta a flexibilidade, permitindo estender o comportamento da aplicação, e facilita a manutenção do código em geral.

Vimos as vantagens dos padrões State. No entanto eles também têm algumas armadilhas.

- Explosão de classes: como todo estado deve ser definido com a ajuda de ConcreteState, há uma chance de acabarmos escrevendo muito mais classes com uma pequena funcionalidade. Considere o caso de máquinas de estado finitas – se houver muitos estados, mas cada estado não for muito diferente de outro estado, ainda precisaremos escrevê-las com classes ConcreteState separadas. Isso aumenta a quantidade de código que devemos escrever, e torna-se difícil revisar a estrutura de uma máquina de estados.

- Com a introdução de cada novo comportamento (mesmo que o acréscimo de um comportamento seja apenas adicionar mais uma ConcreteState), a classe Context deve ser atualizada para lidar com cada comportamento. Isso deixa o comportamento de Context mais frágil a cada novo comportamento adicionado.

Resumo

Para resumir o que aprendemos até agora, nos padrões de projeto State, o comportamento do objeto é decidido com base em seu estado. O estado do objeto pode ser alterado em tempo de execução. A capacidade de Python de mudar o comportamento em tempo de execução facilita bastante a aplicação e a implementação do padrão de projeto State. Esse padrão também nos dá controle sobre a decisão dos estados que os objetos podem assumir, por exemplo, aqueles no exemplo do computador, que vimos anteriormente no capítulo. A classe Context oferece uma interface mais simples para os clientes, e ConcreteState garante que seja fácil adicionar comportamentos aos objetos. Assim, o padrão State melhora a coesão, aumenta a flexibilidade para estender o código e elimina blocos de código redundantes. Estudamos o padrão do ponto de vista acadêmico na forma de um diagrama UML e conhecemos os aspectos da implementação do padrão State com a ajuda de código implementado em Python v3.5. Também demos uma olhada em algumas das armadilhas que podemos encontrar quando se trata do padrão State e vimos que o código pode aumentar significativamente se adicionarmos mais estados ou comportamentos. Espero que você tenha se divertido neste capítulo!

CAPÍTULO 11
Antipadrões

No capítulo anterior, começamos com uma introdução aos padrões de projeto compostos. Vimos como os padrões de projeto funcionam em conjunto para resolver um problema de design do mundo real. Prosseguimos explorando o padrão de projeto Modelo-Visão-Controlador (Model-View-Controller) – o rei dos padrões compostos. Entendemos que o padrão MVC é usado quando precisamos de baixo acoplamento entre os componentes e de uma separação entre os modos como os dados são armazenados e como são apresentados. Também analisamos o diagrama UML do padrão MVC e estudamos como os componentes individuais (modelo, visão e controlador) trabalham entre si. Vimos como o padrão é aplicado no mundo real com a ajuda de uma implementação Python. Discutimos as vantagens do padrão MVC, aprendemos mais sobre ele na seção de Perguntas Frequentes e sintetizamos a discussão no final do capítulo.

Neste capítulo, discutiremos os antipadrões. Ele é diferente de todos os demais capítulos do livro; veremos aqui o que não devemos fazer como arquitetos ou engenheiros de software. Compreenderemos o que são os antipadrões e como eles são visíveis no design de software ou em aspectos do desenvolvimento com a ajuda de exemplos teóricos e práticos.

Os seguintes tópicos serão abordados de forma sucinta neste capítulo:

- uma introdução aos antipadrões;
- antipadrões com exemplos;
- armadilhas comuns durante o desenvolvimento.

No final do capítulo, faremos uma síntese de toda a discussão – considere isso como os pontos principais a ser lembrados.

Uma introdução aos antipadrões

Os princípios de design de software representam um conjunto de regras ou diretrizes que ajudam os desenvolvedores a tomar decisões no nível de design. De acordo com Robert Martin, há quatro características em um design ruim:

- **Imóvel** – uma aplicação é desenvolvida de modo que se torne muito difícil de ser reutilizada.
- **Rígido** – uma aplicação é desenvolvida de modo que qualquer alteração pequena possa resultar na mudança de muitas partes do software.
- **Frágil** – qualquer mudança na aplicação atual resulta em falhas no sistema existente com muita facilidade.
- **Viscoso** – mudanças são feitas pelo desenvolvedor no código ou no próprio ambiente para evitar mudanças difíceis no nível da arquitetura.

As características anteriores de um design ruim, se forem aplicadas, resultam em soluções que não deveriam ser implementadas na arquitetura ou no desenvolvimento de software.

Um antipadrão é o resultado de uma solução ineficiente e contraproducente para problemas recorrentes. O que isso quer dizer? Vamos supor que você se depare com um problema de design de software. Você decide solucionar esse problema. Porém o que acontece se a solução tiver um impacto negativo no design ou causar qualquer problema de desempenho na aplicação? Os antipadrões são processos e implementações defeituosos e comuns em aplicações de software.

Os antipadrões podem ser o resultado de:

- um desenvolvedor desconhecer as práticas de desenvolvimento de

software;
- um desenvolvedor não aplicar padrões de projeto no contexto correto.

Os antipadrões podem se mostrar vantajosos, pois oferecem uma oportunidade para:

- reconhecer problemas recorrentes no mercado de software e oferecer uma solução detalhada para a maioria desses problemas;
- desenvolver ferramentas para reconhecer esses problemas e determinar as causas subjacentes;
- descrever as medidas que podem ser tomadas em vários níveis para melhorar a aplicação e a arquitetura.

Os antipadrões podem ser classificados em duas categorias principais:

1. antipadrões no desenvolvimento de software;
2. antipadrões na arquitetura de software.

Antipadrões no desenvolvimento de software

Quando damos início ao desenvolvimento de software em uma aplicação ou um projeto, pensamos na estrutura do código. Essa estrutura é consistente com a arquitetura do produto, o design, os casos de uso do cliente e muitas outras considerações de desenvolvimento.

Com frequência, quando o software é desenvolvido, ele se desvia da estrutura de código original pelos seguintes motivos:

- o processo de raciocínio do desenvolvedor evolui com o desenvolvimento;
- os casos de uso tendem a mudar com base no feedback dos clientes;
- as estruturas de dados projetadas inicialmente podem passar por mudanças de acordo com considerações de funcionalidade ou de escalabilidade.

Em virtude dos motivos anteriores, o software com frequência passa por

refatorações. A refatoração tem uma conotação negativa para muitas pessoas; na verdade, porém, ela é uma das partes críticas na jornada do desenvolvimento de software, oferecendo uma oportunidade aos desenvolvedores de revisar as estruturas de dados e pensar na escalabilidade e nas necessidades sempre em evolução dos clientes.

Os exemplos a seguir apresentam uma visão geral dos diferentes antipadrões observados no desenvolvimento e na arquitetura de software. Abordaremos apenas alguns deles, juntamente com as causas, os sintomas e as consequências.

Código espaguete

É o antipadrão mais comum e mais conhecido no desenvolvimento de software. Você sabe como é a aparência de um código espaguete? É bem complicada, não é mesmo? Os fluxos de controle do software também se tornam confusos caso as estruturas sejam desenvolvidas de forma *ad hoc*. O código espaguete é difícil de manter e otimizar.

As causas típicas do código espaguete incluem:

- desconhecimento de programação e análise orientadas a objetos;
- a arquitetura ou o design do produto não são considerados;
- mentalidade voltada a correções rápidas.

Você saberá que tem um código espaguete em mãos quando os seguintes pontos forem verdadeiros:

- somente uma reutilização mínima das estruturas é possível;
- os esforços de manutenção são muito altos;
- a capacidade de extensão e a flexibilidade para mudar são reduzidas.

Martelo de Ouro

No mercado de software, você já deve ter visto vários exemplos em que uma dada solução (tecnologia, design ou módulo) é usada em muitos lugares porque essa solução traria vantagens a diversos projetos. Conforme vimos por meio de exemplos neste livro, uma solução é mais

adequada em um determinado contexto e é aplicada a determinados tipos de problema. No entanto as equipes ou os desenvolvedores de software tendem a optar por uma solução comprovada sem considerar se ela é apropriada às necessidades. Esse é o motivo pelo qual esse antipadrão é conhecido como Martelo de Ouro (Golden Hammer): um martelo para todos os pregos possíveis (uma solução para todos os problemas).

As causas típicas do Martelo de Ouro incluem os seguintes cenários:

- a solução chega como uma recomendação vinda de cima (de arquitetos ou líderes de tecnologia), de pessoas que não estão próximas ao problema específico em mãos;
- uma solução gerou muitos benefícios no passado, mas em projetos com um contexto e requisitos diferentes;
- uma empresa está presa a essa tecnologia, pois investiu dinheiro em treinamento dos funcionários ou estes se sentem confortáveis com ela.

As consequências do Martelo de Ouro são:

- uma solução é obsessivamente aplicada a todos os projetos de software;
- o produto é descrito não pelos seus recursos, mas pela tecnologia usada no desenvolvimento;
- nos corredores da empresa, você ouve os desenvolvedores dizendo que "aquilo poderia ter sido melhor do que isso".
- os requisitos não são concluídos e não estão em sincronia com as expectativas dos usuários.

Fluxo de Lava

Esse antipadrão está relacionado ao Código Morto (Dead Code), isto é, um trecho de código inutilizável, que permanece na aplicação de software por medo de que ele cause falhas em outros lugares caso seja modificado. À medida que o tempo passa, esse trecho de código

continua no software e solidifica a sua posição, como se fosse lava se transformando em rocha sólida. Pode ocorrer nos casos em que começarmos a desenvolver o software para dar suporte a determinado caso de uso, mas este muda com o tempo.

As causas de um Fluxo de Lava (Lava Flow) incluem:

- muito código de tentativa e erro na produção;
- código escrito por uma só pessoa, que não é revisado e é passado para outras equipes de desenvolvimento sem qualquer treinamento;
- o raciocínio inicial usado na arquitetura ou no design de software é implementado na base de código, mas ninguém mais o entende.

Os sintomas de um Fluxo de Lava são:

- baixa cobertura de código para testes desenvolvidos (se é que existem);
- muitas ocorrências de código comentado sem motivo;
- interfaces obsoletas ou desenvolvedores tentando contornar o código existente.

Programação do tipo copiar e colar ou cortar e colar

Como você já sabe, este é um dos antipadrões mais comuns. Desenvolvedores experientes disponibilizam trechos de código online (no GitHub ou no Stack Overflow) que são soluções para alguns problemas que ocorrem comumente. Com frequência, os desenvolvedores copiam esses trechos de código exatamente como estão e os usam em sua aplicação para avançar no desenvolvimento. Nesse caso, não há nenhuma validação para saber se esse é o código mais otimizado possível ou se ele é realmente apropriado para o contexto. Isso resulta em uma aplicação de software sem flexibilidade e de difícil manutenção.

As causas de uma programação do tipo copiar e colar (copy-and-paste) ou cortar e colar (cut-and-paste) são:

- desenvolvedores iniciantes não acostumados a escrever código ou que não saibam como desenvolver;
- correção de bug ou avanços rápidos no desenvolvimento;
- duplicação de código por necessidade de uma estrutura de código ou de uma padronização entre os módulos;
- falta de pensamento em longo prazo ou planejamento antecipado.

As consequências da programação cortar e colar ou copiar e colar incluem:

- tipos de problemas semelhantes entre aplicações de software;
- custos mais altos de manutenção e ciclo de vida mais longos para bugs;
- base de código menos modular, com o mesmo código executando em várias linhas;
- problemas de herança que já existiam antes.

Antipadrões na arquitetura de software

A arquitetura de software é uma parte importante da arquitetura de sistemas em geral. Enquanto a arquitetura de sistemas tem como foco aspectos como design, ferramentas e hardware, entre outros, a arquitetura de software está voltada para a modelagem do software, que deve ser bem compreendida pelas equipes de desenvolvimento e de testes, por gerentes de produtos e outros stakeholders (pessoas-chave). Essa arquitetura desempenha um papel crucial para determinar o sucesso da implementação e no modo como o produto funcionará para os clientes.

Discutiremos alguns dos antipadrões do nível de arquitetura que observamos no mundo real no desenvolvimento e na implementação de arquiteturas de software.

Reinventando a roda

Com frequência, ouvimos os líderes de tecnologia falarem de NÃO reinventar a roda. O que isso quer dizer essencialmente? Para alguns, pode significar reutilização de código ou de bibliotecas. Na verdade, isso aponta para a reutilização de arquitetura. Por exemplo, você resolveu um problema e concebeu uma solução no nível de arquitetura. Se você se deparar com um problema semelhante em qualquer outra aplicação, o processo de raciocínio (arquitetura ou design) desenvolvido anteriormente deve ser reutilizado. Não há motivos para rever o mesmo problema e conceber uma solução, o que, essencialmente, significa reinventar a roda.

As causas que levam à reinvenção da roda são:

- ausência de uma documentação ou de um repositório central que discutam problemas no nível de arquitetura e soluções implementadas;
- falta de comunicação entre líderes de tecnologia na comunidade ou na empresa;
- construir do zero é o processo seguido pela empresa; basicamente, há processos imaturos, sem uma sólida implementação e aderência ao processo.

As consequências desse antipadrão incluem:

- soluções em demasia para resolver um problema-padrão, com muitas delas não tendo recebido a atenção merecida;
- mais tempo e utilização de recursos para a equipe de engenharia, resultando em estouros no orçamento e mais tempo para alcançar o mercado (time to market);
- uma arquitetura de sistema fechada (a arquitetura é útil apenas para um produto), duplicação de esforços e gerenciamento de riscos precário.

Vendor lock-in

Como sugere o nome do antipadrão, as empresas de produto tendem a ser dependentes de algumas tecnologias oferecidas pelos fornecedores. Essas tecnologias estão muito amarradas a seus sistemas, a ponto de ser difícil afastar-se delas.

A seguir, apresentamos as causas de um Vendor Lock-in (Dependência de Fornecedor):

- familiaridade com as pessoas que têm autoridade na empresa fornecedora e possíveis descontos na compra da tecnologia;
- tecnologia escolhida com base em ofertas de marketing e vendas, e não na avaliação da tecnologia;
- uso de uma tecnologia comprovada (indicando que o retorno sobre os investimentos com essa tecnologia realmente foram elevados na experiência anterior) no projeto atual, mesmo quando ela não é apropriada para as necessidades ou os requisitos do projeto;
- especialistas em tecnologia/desenvolvedores já estão treinados para usar essa tecnologia.

As consequências de um Vendor Lock-in são estas:

- ciclos de lançamento e de manutenção do produto de uma empresa são diretamente dependentes do ciclo de lançamento do fornecedor;
- o produto é desenvolvido em torno da tecnologia, e não com base nos requisitos do cliente;
- o tempo para o produto alcançar o mercado não é confiável e não atende às expectativas do cliente.

Design by Committee

Às vezes, com base no processo de uma organização, um grupo de pessoas se reúne e faz o design de um sistema em particular. A arquitetura de software resultante muitas vezes é complexa ou fica abaixo do padrão, pois envolve muitos processos de raciocínio, e os especialistas em tecnologia, que podem não ter o conjunto de habilidades adequado nem a experiência para fazer o design dos

produtos, impõem suas ideias.

As causas do Design by Committee (Design por Comitê) são:

- o processo da empresa envolve a aprovação da arquitetura ou do design por muitos stakeholders;
- não há nenhum ponto de contato ou arquiteto único responsável pelo design;
- as prioridades do design são definidas pelo marketing ou por especialistas em tecnologia, e não pelo feedback do cliente.

Os sintomas observados com esse antipadrão incluem:

- pontos de vista conflitantes entre desenvolvedores e arquitetos, mesmo após o design ter sido concluído;
- design excessivamente complexo, muito difícil de ser documentado;
- qualquer mudança na especificação ou no design passa por revisão de muitas pessoas, resultando em atrasos na implementação.

Resumo

Para sintetizar este capítulo, conhecemos os antipadrões, vimos o que são eles e como são classificados. Entendemos que os antipadrões podem estar relacionados ao desenvolvimento ou à arquitetura de software. Vimos antipadrões de ocorrência comum e conhecemos suas causas, os sintomas e as consequências. Estou certo de que você aprenderá com isso e evitará situações desse tipo em seu projeto.

É isso, pessoal; este foi o último capítulo do livro. Espero que vocês tenham gostado dele e que o livro tenha ajudado a aperfeiçoar suas habilidades. Desejo o melhor a todos vocês!