O padrão Page-Object

Jailma Januário da Silva

Leonardo Massayuki Takuno

Resumo

Os objetivos desta parte são: (I) Conhecer um padrão de projeto de testes. (II) Entender o padrão Page Object. (III) Modelar ações a serem realizadas nas páginas web utilizando o padrão Page Object (IV) Construir um teste utilizando o padrão Page Object. (V) Entender as vantagens de utilização do padrão Page Object.

Introdução

O desenvolvimento de um sistema, ao longo do tempo, acumula um conjunto grande de testes. Com o aumento dos testes, torna-se difícil realizar mudanças no código dos testes - uma simples mudança pode causar falhas em diversos testes, mesmo que a aplicação execute apropriadamente. Às vezes, estes problemas são inevitáveis, mas quando ocorrem devem ser rapidamente corrigidos (SELENIUM, 2021).

Este texto apresenta o padrão de projetos para automação de testes chamado *Page Object*. Um padrão de projeto é uma solução geral para problemas recorrentes durante o desenvolvimento de *software*. Segundo MUTHUKADAN (2018), um *page object* representa uma área onde os testes interagem dentro da página web. A ideia desse padrão é encapsular os atributos e as ações que podem ser realizadas em cada página web, separando-os da especificação de testes, e assim, facilitando a manutenção. Os benefícios do uso do *page object* são:

- Facilita a leitura dos casos de teste;
- Cria um código reusável que pode ser compartilhado por múltiplos testes;

- Reduz a quantidade de código duplicado;
- Caso a interface tenha mudança, a correção das mudanças ocorrem somente em um lugar.

Características do padrão *page object*

Para construir testes utilizando o padrão *page object* é importante entender três características a saber:

- **Encapsulamento**: O objeto page object tem a responsabilidade de fornecer acesso aos elementos web da página em teste, os quais sofrem ações que são implementadas por meio de métodos.
- **Abstração**: Os detalhes da implementação não são importantes para o usuário do *page object*. Portanto, saber se o localizador de um elemento web é por *id*, ou se é por *tag* é irrelevante.
- **Separação da responsabilidade**: É preciso separar a responsabilidade da função de teste do *page object*. A função de teste tem a responsabilidade de preparar os dados, executar o script de teste e verificar os resultados. Enquanto que o *page object* fornece métodos para realizar os testes dos elementos *web* por meio da interação com a API *Selenium Webdriver*.

Teste em uma página de busca

O primeiro exemplo consiste em realizar uma pesquisa na página de busca DuckDuckGo. Como qualquer outra ferramenta de busca, o usuário fornece as palavras que deseja consultar e obtém um conjunto de links para os websites que tenham algum tipo de ligação com essa palavra. Para isso, serão utilizados as seguintes etapas:

1. Navegar até a página inicial do DuckDuckGo: http://www.duckduckgo.com.
2. Informar a palavra que se deseja realizar a consulta.
3. Verificar que:

   a. Os resultados aparecem na página de resultado.

   b. A palavra informada aparece na barra de busca.

   c. Pelo menos um resultado da busca contém a palavra informada.

Para implementar estas etapas, observe a Codificação 9.1.

Codificação 9.1. test_busca_ddg.py

```
import time

from selenium import webdriver
from selenium.webdriver.common.by import By

from webdriver_manager.chrome import ChromeDriverManager
from selenium.webdriver.chrome.service import Service


def test_busca_site_duckduckgo():
    servico = Service(ChromeDriverManager().install())
    navegador = webdriver.Chrome(service=servico)

    navegador.get('http://www.duckduckgo.com')

    palavras_busca = 'Page Object'

    caixa = navegador.find_element(By.ID, 'searchbox_input')
    caixa.send_keys(palavras_busca)
    caixa.submit()
    time.sleep(5)

    links = navegador.find_elements(By.PARTIAL_LINK_TEXT,
palavras_busca)
```

```
    assert len(links) > 0

            xpath = f"//*[@class='react-results--main']//*[contains(text(),
'{palavras_busca}')]"
    resultados = navegador.find_elements(By.XPATH, xpath)
    assert len(resultados) > 0

    caixa_de_busca = navegador.find_element(By.ID, 'search_form_input')
    assert caixa_de_busca.get_attribute('value') == palavras_busca

    navegador.close()
    navegador.quit()
```

Fonte: do autor, 2022.

A função **test_busca_duckduckgo()** implementa as etapas de testes seguindo o padrão *Arrange-Act-Assert*. Note que o teste inicia-se com a etapa *Arrange*, que é a preparação do teste:

- Iniciar o driver do Google Chrome e acessar a página do DuckDuckGo, como apresenta a seguir:

```
    servico = Service(ChromeDriverManager().install())
    navegador = webdriver.Chrome(service=servico)
    navegador.get('http://www.duckduckgo.com')
```

É importante atentar-se que esta instrução não aguarda a página do DuckDuckGo ser carregada, ela simplesmente faz a requisição get para o link.

- Definir as palavras a serem buscadas pelo DuckDuckGo:

```
    palavras_busca = 'Page Object'
```

A etapa Act que informa as palavras de busca e solicita que o DuckDuckGo realize a consulta. O primeiro passo para automatizar interações com a página web é localizar o elemento web que deve sofrer a ação. Para isso, Act deve buscar a caixa de texto utilizando a instrução **find_element()**, como apresenta a seguir:

caixa = navegador.find_element(By.ID, 'searchbox_input')

Para determinar qual é o localizador, é necessário verificar a estrutura da página HTML. Sobre o campo de busca, acione o menu suspenso utilizando o botão direito do mouse, e escolha a opção "Inspect". Esse elemento tem um atributo id com o valor **'searchbox_input'**. Observe a Figura 9.1 que apresenta a estrutura HTML da página de busca do DuckDuckGo.

O método **find_element()** devolve a referência ao campo de busca e atribui a instância do elemento web na variável caixa. Com isso, agora é possível realizar interações com esse elemento web. O método **send_keys()** envia uma sequência de teclas pressionadas para o elemento campo de busca, como se um usuário digitasse no teclado. Verifique a chamada da função a seguir:

```
caixa.send_keys(palavras_busca)
caixa.submit()
```

Após o envio de algumas palavras ao campo de busca, o método **submit()** é invocado, o que sinaliza o DuckDuckGo que ele deve realizar a busca das palavras informadas.

Figura 9.1. Estrutura HTML da página DuckDuckGo

Fonte: do autor, 2022.

A etapa de Assert ocorre em três partes neste exemplo. Na primeira parte, o *script* procura um elemento web com o ID de nome "links", que por sua vez possui um elemento filho *div* para cada link do resultado. Utiliza-se o buscar por PARTIAL_LINK_TEXT para encontrar todos os resultados da consulta. Note que o método é o **find_elements()**, com elements no plural, que é um método que devolve uma lista de elementos. Após isso, o script verifica se o tamanho da lista é maior do que zero, ou seja, a consulta trouxe algum resultado.

```
        links = navegador.find_elements(By.PARTIAL_LINK_TEXT, palavras_busca)
    assert len(links) > 0
```

Embora a consulta tenha trazido alguns links, o script ainda deve verificar se o esse resultado está adequado. Para isso, verifica-se se as palavras submetidas para a consulta encontram-se nos resultados. Pode-se utilizar o XPath para identificar quais são os links resultados que possuem as palavras submetidas nas consultas. Neste trecho do script, o XPath busca uma div com ID igual a "*links*", então procura por descendentes que contenham as palavras da consulta. Após isso, verifica se o método **find_elements()** devolveu algum resultado.

```
        xpath = f"//*[@class='react-results--main']//*[contains(text(), '{palavras_busca}')]"
    resultados = navegador.find_elements(By.XPATH, xpath)
    assert len(resultados) > 0
```

Como na *assertion* anterior, este trecho do script verifica se pelo menos um resultado foi encontrado. É apenas um teste de sanidade. Poderia ser mais rígido, como por exemplo, verificar se cada link possui as palavras de consulta. Porém nem sempre os resultados trazem as palavras exatamente da mesma forma que foi enviada na consulta. Alguns resultados podem ter letras maiúsculas. Os localizadores e a lógica precisariam ser muito mais complicados para uma verificação avançada. Logo, uma simples verificação será o suficiente.de texto

Por fim, o script verifica se as palavras submetidas para consulta ainda estão na caixa de busca, na página de resultados. Para isso, é necessário buscar novamente o elemento web que representa a caixa de busca, e verificar se o seu conteúdo é igual ao texto utilizado para realizar a busca.

```
    caixa_de_busca = navegador.find_element(By.ID, 'search_form_input')
```

```
    assert caixa_de_busca.get_attribute('value') == palavras_busca
```

Agora, execute o pytest de acordo com a Figura 9.2.

Figura 9.2. Execução do teste

Fonte: do autor, 2022.

Para converter este teste no padrão *page object*, organize os arquivos como apresenta a Figura 9.3, com uma pasta de nome page, para classes do tipo *page object*, e outra pasta para testes, chamada tests.

Figura 9.3. Organização dos testes com padrão *page object*

Fonte: do autor, 2022.

Na pasta pages, crie um arquivo chamado **pagina_busca_ddg.py** com a classe PaginaBuscaDDG, conforme a Codificação 9.2.

Codificação 9.2. pagina_busca_ddg.py

```
from selenium.webdriver.common.by import By

class PaginaBuscaDDG:
    URL = 'https://www.duckduckgo.com'

    def __init__(self, navegador):
        self.navegador = navegador

    def carregar(self):
        self.navegador.get(self.URL)

    def buscar(self, palavras_busca):
        caixa = self.navegador.find_element(By.ID, 'searchbox_input')
        caixa.send_keys(palavras_busca)
        caixa.submit()
```

```
    def conta_resultados(self):
        xpath = f'//*[@class="react-results--main"]/li'
        links = self.navegador.find_elements(By.XPATH, xpath)
        return len(links)

    def conta_palavras_nos_resultados(self, palavras_busca):
            xpath = f"//*[@class="react-results--main"]//*[contains(text(),
'{palavras_busca}')]"
        resultados = self.navegador.find_elements(By.XPATH, xpath)
        return len(resultados)

    def texto_na_caixa_de_busca(self):
                caixa_de_busca = self.navegador.find_element(By.ID,
'search_form_input')
        return caixa_de_busca.get_attribute('value')
```

Fonte: do autor, 2022.

Pela Codificação 9.2 pode-se observar que a classe PaginaBuscaDDG recebe no construtor a instância de um driver do Selenium na variável navegador. A partir daí, é possível carregar a página web, que neste caso é a página do DuckDuckGo. Pode-se, também, consultar uma palavra na página de busca. Além disso, a classe encapsula a verificação dos resultados, que poderiam estar modelados em uma outra classe só para avaliar a página de resultados da busca. No entanto, por simplicidade, a classe PaginaBuscaDDG incluiu a contagem dos resultados, a verificação das palavras nos links de resultados, e também, checa se o texto na caixa de busca é o mesmo que aquele que foi digitado na página de consulta.

Para o arquivo test_busca_ddg.py, observe a Codificação 9.3.

Codificação 9.3. test_busca_ddg.py

```
import pytest

from selenium import webdriver
from selenium.webdriver.common.by import By
```

```
from webdriver_manager.chrome import ChromeDriverManager
from selenium.webdriver.chrome.service import Service

from pages.pagina_busca_ddg import PaginaBuscaDDG

@pytest.fixture
def navegador():
    servico = Service(ChromeDriverManager().install())
    navegador = webdriver.Chrome(service=servico)
    yield navegador
    navegador.close()
    navegador.quit()

def test_busca_site_google(navegador):
    palavras_busca = 'Page Object'

    paginaBusca = PaginaBuscaDDG(navegador)
    paginaBusca.carregar()
    paginaBusca.buscar(palavras_busca)

    assert paginaBusca.conta_resultados() > 0
     assert paginaBusca.conta_palavras_nos_resultados(palavras_busca)
> 0
    assert paginaBusca.texto_na_caixa_de_busca() == palavras_busca
```

Fonte: do autor, 2022.

Perceba pela Codificação 9.3, que o teste aloca uma instância da classe PaginaBuscaDDG, passando para o construtor um driver do Selenium que está armazenada na variável navegador. Em seguida, realiza o método carregar, que solicita ao navegador o acesso à página principal do DuckDuckGo. Na sequência, submete palavras a serem buscadas para o método buscar. E finaliza o teste realizando três validações. O primeiro, verifica se a contagem de resultados foi maior do que zero. O segundo, conta

o número de vezes que as palavras submetidas à busca aparecem nos resultados dos links. O terceiro e último, verifica se na página de resultado, a caixa de busca contém a palavra de busca. Assim, o código fica mais limpo, mais fácil de entender e mais simples caso seja necessário realizar alguma manutenção.

Agora, execute o pytest conforme apresenta a Figura 9.4.

Figura 9.4. Execução do pytest com o *page object*

Fonte: do autor, 2022.

Trabalhando com alertas

O próximo exemplo, consiste em testar uma página com mensagens de alerta. Para isso, observe a página de exemplo de javascript que contém um exemplo de utilização de alertas. O endereço URL é https://www.w3schools.com/jsref/tryit.asp?filename=tryjsref_prompt, e contém uma tela conforme apresenta a Figura 9.5.

Figura 9.5. W3schools exemplo de mensagens de alerta

Fonte: do autor, 2022.

Observe que esta página é mais complexa do que os exemplos vistos até o momento. Para começar, pode-se observar três regiões, em que a primeira região contém o cabeçalho da página com uma barra de ferramentas, a segunda região contém uma área de código, e a terceira região é o local onde apresenta o resultado do código HTML que está contido na área de código. Ao acionar o botão com o texto "Try it", a página apresenta uma mensagem de alerta solicitando um texto de entrada, e como exemplo pré-definido, contém o texto "Harry Potter", conforme apresenta a Figura 9.6. Conforme se

pode observar, é possível acionar a opção *Cancel*, o qual cancela a operação, ou é possível acionar a opção OK, que confirma a operação. Neste caso, uma mensagem em inglês aparece na página, como apresenta a Figura 9.7. A mensagem, para este exemplo, é "Hello Harry Potter! How are you today?".

Figura 9.6. W3schools exemplo com alertas

Fonte: do autor, 2022.

Figura 9.7. Resposta ao botão OK

Fonte: do autor, 2022.

Para testar o botão "Try it", conforme as Figuras 9.6 e 9.7, é necessário realizar três passos:

1. Acesse o link: https://www.w3schools.com/jsref/tryit.asp?filename=tryjsref_prompt
2. Acesse a área onde se encontra o botão "Try it". Essa região é delimitada pela tag HTML *<iframe>*.
3. Acione o botão "Try it".
4. Acesse a região em que se encontra a janela de mensagem.
5. E, finalmente, acione o botão OK da janela de mensagem.

Para obter o ID do elemento web que representa a região onde se encontra o botão "Try it", acione o botão direito do mouse e selecione a opção "Inspect". Observe a Figura 9.8 e perceba que o ID do <iframe> é o iframeResult, o qual será utilizado no script para acessar o botão "Try it". Utilize o método **switch_to.frame()** para trocar para o frame iframeResult.

Figura 9.8. Obtendo o ID do frame

Fonte: do autor, 2022.

Para acionar o botão 'Try it', utilize o método find_element passando o XPATH, e após isso acione o método click(). Com isso uma janela de prompt surge na tela, de acordo Figura 9.6, o qual é possível fornecer um nome, que por padrão é "Harry Potter". Com opções de OK e Cancelar.

Para acessar a janela de prompt utilize o método **switch_to.alert()** e a partir daí utilize o método **accept()** para selecionar a opção OK. Observe a Codificação 9.4 o script de teste **test_js_prompt.py**.

Codificação 9.4. test_js_prompt.py

```
from selenium.webdriver import Chrome
from selenium.webdriver.common.by import By
from webdriver_manager.chrome import ChromeDriverManager
from selenium.webdriver.chrome.service import Service

def test_js_prompt():
    servico = Service(ChromeDriverManager().install())
    driver = Chrome(service=servico)
                                                    URL          =
'https://www.w3schools.com/jsref/tryit.asp?filename=tryjsref_prompt'
    driver.get(URL)
    driver.switch_to.frame('iframeResult')
    driver.find_element(By.XPATH, "//button['Try it']").click()
    driver.switch_to.alert.accept()
    resultado = driver.find_element(By.ID, 'demo').text
    assert 'Hello Harry Potter! How are you today?' == resultado
    driver.close()
```

Fonte: do autor, 2022.

Execute o pytest e verifique pela Figura 9.9 a execução dos testes.

Figura 9.9. Execução do pytest

Fonte: do autor, 2022.

Os testes estão passando, porém o código de teste está dependente das instruções do Selenium. Para melhorar o código de teste, é possível modelar com o Padrão Page Object. Para isso, defina a estrutura de pastas conforme a Figura 9.10.

Figura 9.10. Estrutura de pastas

Fonte: do autor, 2022.

Para o arquivo **pagina_window_prompt.py** observe a Codificação 9.5.

Codificação 9.5. pagina_window_prompt.py

```
from selenium.webdriver.common.by import By

class PaginaWindowPrompt:
    URL = 'https://www.w3schools.com/jsref/tryit.asp?filename=tryjsref_prompt'

    def __init__(self, navegador):
        self.navegador = navegador

    def carregar(self):
        self.navegador.get(self.URL)

    def trocar_frame(self):
        self.navegador.switch_to.frame('iframeResult')

    def clicar_no_botao_try_it(self):
        self.navegador.find_element(By.XPATH, "//button['Try it']").click()

    def clicar_no_botao_ok(self):
```

```
        self.navegador.switch_to.alert.accept()

    def obter_texto_resultado(self):
        return self.navegador.find_element(By.ID, 'demo').text
```

Fonte: do autor, 2022.

Para o arquivo test_js_promp.py observe a Codificação 9.6.

Codificação 9.6. test_js_prompt2.py

```
from selenium.webdriver import Chrome

from selenium.webdriver.common.by import By
from pages.pagina_window_prompt import PaginaWindowPrompt

from webdriver_manager.chrome import ChromeDriverManager
from selenium.webdriver.chrome.service import Service


def test_js_prompt_hello_harry_potter():
    servico = Service(ChromeDriverManager().install())
    navegador = Chrome(service=servico)
    pagina = PaginaWindowPrompt(navegador)
    pagina.carregar()
    pagina.trocar_frame()
    pagina.clicar_no_botao_try_it()
    pagina.clicar_no_botao_ok()
    resultado = pagina.obter_texto_resultado()
    assert 'Hello Harry Potter! How are you today?' == resultado
```

Fonte: do autor, 2022.

Utilizando-se dos métodos implementados na classe PaginaWindowPromt é possível observar, pela Figura 9.4, que o código de teste fica mais legível e mais simples de entender. Um dos benefícios do ganho de legibilidade é a manutenção do código de teste. Outro benefício, é a facilidade para identificar

falhas, ou bugs. Um terceiro benefício inclui o encapsulamento da tecnologia Selenium ao acessar elementos web, que é um tanto confuso para quem está iniciando nos testes. O código no Padrão Page Object ajuda a entender detalhes da tecnologia. Além disso, qualquer programador que começar no projeto pode utilizar do código de teste para entender as regras do negócio que estão evidenciadas nos códigos de testes e aprender sobre a tecnologia do Selenium caso ainda não saiba.

Agora, em vez de apenas selecionar a opção OK da janela de diálogo, suponha que você queira testar a entrada de dados do alerta e fornecer um nome para teste. Observe a Codificação 9.7.

Codificação 9.7. pagina_window_prompt.py 2

```
import pytest
from selenium.webdriver.common.by import By
from pages.pagina_window_prompt import PaginaWindowPrompt

from webdriver_manager.chrome import ChromeDriverManager
from selenium.webdriver.chrome.service import Service

@pytest.fixture
def pagina():
    servico = Service(ChromeDriverManager().install())
    navegador = Chrome(service=servico)
    pagina_window_promt = PaginaWindowPrompt(navegador)
    return pagina_window_promt

def test_js_prompt_hello_harry_potter(pagina):
    pagina.carregar()
    pagina.trocar_frame()
    pagina.clicar_no_botao_try_it()
    pagina.clicar_no_botao_ok()
    resultado = pagina.obter_texto_resultado()
```

```
    assert 'Hello Harry Potter! How are you today?' == resultado

def test_js_prompt_hello_jose(pagina):
    pagina.carregar()
    pagina.trocar_frame()
    pagina.clicar_no_botao_try_it()
    pagina.clicar_no_botao_ok_com_o_nome('Jose')
    resultado = pagina.obter_texto_resultado()
    assert 'Hello Jose! How are you today?' == resultado
```

Fonte: do autor, 2022.

Observe que agora, o arquivo pagina_window_prompt.py tem dois testes, então inclui-se uma fixture. O primeiro teste, executa da mesma forma que a Codificação 9.4. O segundo teste, informa o nome 'Jose' na caixa de diálogo e seleciona a opção OK por meio do método **clicar_no_botao_ok_com_o_nome('Jose')**, que se encontra na Codificação 9.8.

Codificação 9.8. pagina_window_prompt.py 3

```
from selenium.webdriver.common.by import By

class PaginaWindowPrompt:
    URL = 'https://www.w3schools.com/jsref/tryit.asp?filename=tryjsref_prompt'

    def __init__(self, navegador):
        self.navegador = navegador

    def carregar(self):
        self.navegador.get(self.URL)

    def trocar_frame(self):
        self.navegador.switch_to.frame('iframeResult')
```

```
    def clicar_no_botao_try_it(self):
        self.navegador.find_element(By.XPATH, "//button['Try it']").click()

    def clicar_no_botao_ok(self):
        self.navegador.switch_to.alert.accept()

    def obter_texto_resultado(self):
        return self.navegador.find_element(By.ID, 'demo').text

    def clicar_no_botao_ok_com_o_nome(self, nome):
        alert = self.navegador.switch_to.alert
        alert.send_keys(nome)
        alert.accept()
```

Fonte: do autor, 2022.

Considerações finais

Esta parte apresentou o padrão Page Object, que é um padrão utilizado para testes. Criou-se dois exemplos que modelam testes de interface em páginas web usando o Selenium e modelados com o Padrão Page Object. O primeiro exemplo acessou a página de busca DuckDuckGo utilizando o Padrão Page Object. E o segundo exemplo, realizou-se testes sobre uma janela de diálogo. Percebe-se a facilidade de entendimento do código de testes ao utilizar o padrão Page Object e também discutiu-se alguns benefícios obtidos na organização do código e melhoria da legibilidade. É necessário realizar vários testes em várias situações com o Selenium, e conforme você avança com os trabalhos de automação de testes, torna-se necessário a organização do código de testes, e o Page Object ajudará neste sentido.

Referências

MUTHUKADAN, B. **Selenium with python**. 2018. Disponível em: <https://selenium-python.readthedocs.io/index.html>. Acesso em: 10 jul. 2022.

PEIXOTO, R. **Selenium webdriver**: descomplicando testes automatizados com java. Casa do Código - Livros para o programador, 2018.

PYTEST. **Documentação versão de python 3.7+**. 2015. Disponível em: <https://docs.pytest.org/en/7.1.x/index.html>. Acesso em: 11 jun. 2022

PYTHON. **Documentação versão de python 3.10.5.** The python standard library. Development tools. unittest.mock - mock object library. Versão em inglês. Disponível em: <https://docs.python.org/3/library/unittest.mock.html>. Acesso: 08 jul. 2022.

RAGHAVENDRA, S. **Python testing with selenium**: learn to implement different testing techniques using selenium webdriver. India, Dharwad Karnataka: Appres(R), 2021.

SALE, D. **Testing Python:** applying unit testing, TDD, BDD, and accepting testing. Wiley; 2014.

SELENIUM. **Documentação** - O projeto selenium de automação de navegadores. The Selenium umbrella. 2021. Versão online. Disponível em: <https://www.selenium.dev/pt-br/documentation/>. Acesso em: 09 jul. 2022.