03

# Aplicando Page Object nos testes

## Transcrição

No último vídeo discutimos sobre o padrão *Page Object*, bastante importante para organizarmos os códigos de testes, melhorarmos sua legibilidade e facilitarmos a manutenção. Nesse vídeo faremos uma refatoração do nosso `LoginTest` seguindo esse padrão, recomendado pelo próprio Selenium.

A ideia é que nossa classe não tenha mais acesso direto à API do Selenium, que deve ser isolada no Page Object de login. Por exemplo, não queremos mais ter acesso ao Webdriver ou a `URL_LOGIN`, que são especificos da API.

Um Page Object nada mais é do que uma classe que representa uma página. Como estamos testando a funcionalidade de login, que é uma página da nossa aplicação, teremos uma classe `PaginaDeLogin`, `LoginPage` ou algo do tipo, e toda manipulação da página de login ficará dentro dessa classe.

Criaremos em `LoginTest` uma variável do tipo `LoginPage` chamada `paginaDeLogin`. Como a classe `LoginPage` não existe, vamos criá-la no próprio pacote `br.com.alura.leilao.login`.

```
public class LoginTest {

    private LoginPage paginaDeLogin;
//...
```

COPIAR CÓDIGO

```
package br.com.alura.leilao.login;

public class LoginPage {
}
```

COPIAR CÓDIGO

O Page Object, portando, é uma classe que representa determinada página. Moveremos para esta classe os códigos específicos da API do Selenium, como a URL de login e o `WebDriver`.

```
public class LoginPage {

    private static final String URL_LOGIN = "http://localhost:808
    private WebDriver browser;
}
```

COPIAR CÓDIGO

Teremos alguns erros de compilação nos métodos do JUnit `LoginTest`, já que anteriormente estávamos instanciando o driver do Chrome, acessando a URL de login e assim por diante. Moveremos esses códigos para a classe `LoginPage`, mas removendo as anotações do JUnit - lembre-se: os Page Objects não possuem acesso ao JUnit, tanto anotações quanto *asserts*.

```
public class LoginPage {

    private static final String URL_LOGIN = "http://localhost:808
    private WebDriver browser;

    public static void beforeAll() {
        System.setProperty("webdriver.chrome.driver", "/driver
    }
```

```
    public void beforeEach(){
        this.browser = new ChromeDriver();
        browser.navigate().to(URL_LOGIN);
    }
}
```

COPIAR CÓDIGO

Ao invés das anotações, podemos declarar um construtor e mover tais códigos para ele. Assim, toda vez que a `LoginPage` for instanciada, nossas propriedades serão instanciadas e as ações serão feitas normalmente.

```
public class LoginPage {

    private static final String URL_LOGIN = "http://localhost:808
    private WebDriver browser;

    public LoginPage() {

        System.setProperty("webdriver.chrome.driver", "/drivers/c
        this.browser = new ChromeDriver();
        browser.navigate().to(URL_LOGIN);
    }
}
```

COPIAR CÓDIGO

A partir de agora, o método `beforeEach()` simplesmente intanciará o nosso Page Object, que abstrairá todas as configurações.

```
public class LoginTest {
```

```
    private LoginPage paginaDeLogin;

    @BeforeEach
    public void beforeEach() {
        this.paginaDeLogin = new LoginPage();
    }

    @AfterEach
    public void afterEach(){
        this.browser.quit();
    }
```

COPIAR CÓDIGO

No `afterEach()` nos estávamos fechando o browser, mas não temos mais acesso a ele. Resolveremos todas as situações desse tipo transformando-as em métodos de `LoginPage` que poderão ser acessados da classe de teste. Nesse caso, teremos um método `paginaDeLogin.fechar()` que será criado no Page Object.

```
public class LoginTest {

    private LoginPage paginaDeLogin;

    @BeforeEach
    public void beforeEach() {
        this.paginaDeLogin = new LoginPage();
    }

    @AfterEach
    public void afterEach(){
        this.paginaDeLogin.fechar();
    }
```

COPIAR CÓDIGO

```
public class LoginPage {

    private static final String URL_LOGIN = "http://localhost:808
    private WebDriver browser;

    public LoginPage() {

        System.setProperty("webdriver.chrome.driver", "/drivers/c
        this.browser = new ChromeDriver();
        browser.navigate().to(URL_LOGIN);
    }

    public void fechar() {
        this.browser.quit();
    }
}
```

COPIAR CÓDIGO

Agora refatoraremos nossos métodos de teste. As linhas em que manipulamos a navegação na aplicação também virarão métodos, o que tornará nossos cenários mais legíveis. Começaremos extraindo as linhas em que preenchemos o formulário de login para um método `paginaDeLogin.preencheFormularioDeLogin()`, passando como parâmetros um login e uma senha.

```
@Test
public void deveriaEfetuarLoginComDadosValidos() {
    paginaDeLogin.preencheFormularioDeLogin("fulano", "pass");
    browser.findElement(By.id("login-form")).submit();
    Assert.assertFalse(browser.getCurrentUrl().equals(URL_LOGIN));
    Assert.assertEquals("fulano", browser.findElement(By.id("u
}
```

COPIAR CÓDIGO

Moveremos para ele o código que estava no método de teste, alterando os parâmetros recebidos para `username` e `password`, tornado-o mais flexível.

```
public void preencheFormularioDeLogin(String username, String pas
    browser.findElement(By.id("username")).sendKeys(username);
    browser.findElement(By.id("password")).sendKeys(password);
}
```

COPIAR CÓDIGO

Dessa forma, toda vez que quisermos preencher o formulário de login, bastará chamarmos esse método. Em seguida, faremos o mesmo processo para o envio do formulário, que se tornará o método `paginaDeLogin.efetuaLogin()`.

```
@Test
public void deveriaEfetuarLoginComDadosValidos() {
    paginaDeLogin.preencheFormularioDeLogin("fulano", "pass");
    paginaDeLogin.efetuaLogin();
    Assert.assertFalse(browser.getCurrentUrl().equals(URL_LOGIN)
    Assert.assertEquals("fulano", browser.findElement(By.id("usua
}
```

COPIAR CÓDIGO

```
public void efetuaLogin() {
    browser.findElement(By.id("login-form")).submit();

}
```

COPIAR CÓDIGO

Repare que nossos *asserts* estão verificando elementos da página. Também extrairemos essas verificações para métodos específicos, começando pela verificação da página de login, que se tornará o método `paginaDeLogin.isPaginaDeLogin()`.

```
@Test
public void deveriaEfetuarLoginComDadosValidos() {
    paginaDeLogin.preencheFormularioDeLogin("fulano", "pass");
    paginaDeLogin.efetuaLogin();
    Assert.assertFalse(paginaDeLogin.isPaginaDeLogin());
    Assert.assertEquals("fulano", browser.findElement(By.id("usua
}
```

COPIAR CÓDIGO

```
public boolean isPaginaDeLogin() {
    return browser.getCurrentUrl().equals(URL_LOGIN);
}
```

COPIAR CÓDIGO

O próximo *assert* se refere ao nome do usuário logado, que será retornado pelo método `paginaDeLogin.getNomeUsuarioLogado()`.

```
@Test
public void deveriaEfetuarLoginComDadosValidos() {
    paginaDeLogin.preencheFormularioDeLogin("fulano", "pass");
    paginaDeLogin.efetuaLogin();
    Assert.assertFalse(paginaDeLogin.isPaginaDeLogin());
```

```
    Assert.assertEquals("fulano", paginaDeLogin.getNomeUsuarioLog
}
```

COPIAR CÓDIGO

```
public Object getNomeUsuarioLogado() {
    return browser.findElement(By.id("usuario-logado")).getText()
}
```

COPIAR CÓDIGO

Devemos nos lembrar que, em `naoDeveriaLogarComDadosInvalidos()`, essa busca por usuário nos retornava uma exceção da API do Selenium, algo que também deve ter removido da classe de teste. Para resolvermos isso, podemos encapsular o corpo do método em um `try/catch` que, caso não consiga recuperar a string com o nome do usuário logado, pegará a `NoSuchElementException` e retornará nulo ( `null` ) para o teste.

```
public Object getNomeUsuarioLogado() {
    try {
        return browser.findElement(By.id("usuario-logado")).getTe
    } catch (NoSuchElementException e) {
        return null;
    }
}
```

COPIAR CÓDIGO

Vamos refatorar o teste `naoDeveriaLogarComDadosInvalidos()` usando os novos métodos de `LoginPage` : usaremos os parâmetros `invalido` e `123` para simularmos o login inválido, alteraremos nosso *assert* para `assertTrue()`, verificando se ainda estamos na página de login, e usaremos um `assertNull()`

para verificarmos se o retorno de `getNomeUsuarioLogado()` corresponde ao esperado.

```
@Test
public void naoDeveriaLogarComDadosInvalidos() {
    paginaDeLogin.preencheFormularioDeLogin("invalido", "123");
    paginaDeLogin.efetuaLogin();

    Assert.assertTrue(paginaDeLogin.isPaginaDeLogin());
    Assert.assertNull(paginaDeLogin.getNomeUsuarioLogado());
}
```

COPIAR CÓDIGO

Em `naoDeveriaAcessarPaginaRestritaSemEstarLogado()`, teremos um método `navegaParapaginaDeLances()`. Dessa forma, não deixaremos exposta a URL da nova página no método de testes. Além disso, verificaremos se continuamos na página de login com `isPaginaDeLogin()`.

```
@Test
public void naoDeveriaAcessarPaginaRestritaSemEstarLogado() {
    paginaDeLogin.navegaParaPaginaDeLances();
    Assert.assertTrue(paginaDeLogin.isPaginaDeLogin());
    Assert.assertFalse(browser.getPageSource().contains("Dados do
}
```

COPIAR CÓDIGO

```
public void navegaParaPaginaDeLances() {
    this.browser.navigate().to("http://localhost:8080/leiloes/2");
}
```

COPIAR CÓDIGO

Também estavamos verificando se o código da fonte da página não continua o texto "Dados do Leilão". Faremos essa verificação com um método `paginaDeLogin.contemTexto()`, que receberá uma string e devolverá se ela existe ou não no código.

```
@Test
public void naoDeveriaAcessarPaginaRestritaSemEstarLogado() {
    paginaDeLogin.navegaParaPaginaDeLances();
    Assert.assertTrue(paginaDeLogin.isPaginaDeLogin());
    Assert.assertFalse(paginaDeLogin.contemTexto("Dados do Leilã
}
```

COPIAR CÓDIGO

```
public boolean contemTexto(String texto) {
    return browser.getPageSource().contains(texto);
}
```

COPIAR CÓDIGO

Esse método também estará presente no teste de falha de autenticação, mas dessa vez procurando por "Usuário e senha inválidos".

```
@Test
public void naoDeveriaLogarComDadosInvalidos() {
    paginaDeLogin.preencheFormularioDeLogin("invalido", "123");
    paginaDeLogin.efetuaLogin();

    Assert.assertTrue(paginaDeLogin.isPaginaDeLogin());
    Assert.assertFalse(paginaDeLogin.contemTexto("Usuário e senh
```

```
    Assert.assertNull(paginaDeLogin.getNomeUsuarioLogado());
}
```

COPIAR CÓDIGO

São os mesmos cenários de teste, mas agora extraídos para um Page Object que representa a nossa página de login, tornando nosso código mais legível. Além disso, se quisermos mudar para outra biblioteca que não seja o Selenium, é possível continuar usando o mesmo Page Object, sem interferir nas nossas verificações.

Ao rodarmos nosso testes, porém, receberemos um erro em `naoDeveriaLogarComDadosInvalidos()`. Isso acontece porque verificamos se estamos na página de login, mas, como vimos anteriormente, a página redirecionada na verdade é http://localhost:8080/login?error (http://localhost:8080/login?error).

Resolveremos esse problema criando um método `isPaginaDeLoginComDadosInvalidos()` que busca pela URL correta.

```
public boolean isPaginaDeLoginComDadosInvalidos() {
    return browser.getCurrentUrl().equals(URL_LOGIN + "?error");
}
```

COPIAR CÓDIGO

Existem diversas outras formas de fazermos essas construções, por exemplo recebendo uma string por parâmetro para completar a URL, e você é livre para experimentá-las. Executando novamente, nossos métodos passarão com sucesso.

Essa é a ideia geral dos Page Objects. Para cada página que quisermos testar, teremos um Page Object com métodos que manipulam essa página, preenchendo

formulários, clicando em elementos, recuperando e/ou devolvendo informações e assim por diante - tudo isso sem interferir, e sem sofrer interferência, da API do JUnit. Assim nosso código fica mais legível, mais fácil de fazer manutenção e com suas responsabilidades isoladas. Daqui para frente construiremos nossass funcionalidades utilizando esse padrão.