01

Melhorando nosso código de teste com Page Objects

Transcrição

Começando deste ponto? Você pode fazer o DOWNLOAD (https://s3.amazonaws.com/caelum-online-public/PM-74/testes-de-sistema-cap2.zip) do projeto completo do capítulo anterior e continuar seus estudos a partir deste capítulo.

Já vimos que o Selenium facilita e muito nossa vida. Com o que já sabemos hoje, podemos escrever muitos métodos de teste e testar diferentes formulários web. Precisamos agora trabalhar para que nosso código de teste não se torne mais complicado do que deveria.

Veja o método de teste abaixo:

```
public class UsuariosSystemTest {

    private WebDriver driver;

    @Before
    public void inicializa() {
        driver = new FirefoxDriver();
    }

    @Test
    public void deveAdicionarUmUsuario() {
        driver.get("http://localhost:8080/usuarios/new");

        WebElement nome = driver.findElement(By.name("usuario.nome"));
        WebElement email = driver.findElement(By.name("usuario.email"));

        nome.sendKeys("Ronaldo Luiz de Albuquerque");
        email.sendKeys("ronaldo2009@terra.com.br");

        nome.submit();

        assertTrue(driver.getPageSource()
        .contains("Ronaldo Luiz de Albuquerque"));
        assertTrue(driver.getPageSource()
        .contains("ronaldo2009@terra.com.br"));

    }

    @After
    public void encerra() {
        driver.close();
    }

}
```

O código de testes é simples de ler. Mas poderia ser melhor e mais fácil. E se conseguíssemos escrever dessa forma?

```
    @Test
    public void deveAdicionarUmUsuario() {

        usuarios.novo()
        .cadastra("Ronaldo Luiz de Albuquerque", "ronaldo2009@terra.com.br");

        assertTrue(usuarios.existeNaListagem(
                "Ronaldo Luiz de Albuquerque", "ronaldo2009@terra.com.br");

    }
```

Veja só como o teste é bem mais legível ! É muito mais fácil de ler e entender o que o teste faz. Agora precisamos implementar. Veja que a declaração da variável `usuarios` foi omitida do código acima. A ideia é que essa variável representa "a página de usuários". Veja as ações que ela contém: `novo()` para ir para a página de novo usuário, e `existeNaListagem()` , que verifica se um usuário está lá.

Vamos escrever então uma classe que representa a página de listagem de usuários e contém as operações descritas anteriormente. Para implementá-las, basta fazer uso do Selenium, igual fizemos nos nossos testes até então:

```
class UsuariosPage {

    public void visita() {
        driver.get("localhost:8080/usuarios");
    }

    public void novo() {
        // clica no link de novo usuario
        driver.findElement(By.linkText("Novo Usuário")).click();
    }

    public boolean existeNaListagem(String nome, String email) {
        // verifica se ambos existem na listagem
        return driver.getPageSource().contains(nome) &&
                driver.getPageSource().contains(email);
    }

}
```

Ótimo! Veja que escrevemos basicamente o mesmo código que escrevemos anteriormente, mas dessa vez os escondemos em uma classe específica. Mas esse código ainda não funciona. Precisamos do `driver` do Selenium. Mas, ao invés de instanciar um driver dentro da classe, vamos receber esse driver pelo construtor. Dessa forma, ainda conseguimos fazer uso dos métodos `@Before` e `@After` do JUnit para abrir e fechar o driver e, quando tivermos mais classes iguais a essa (para cuidar das outras páginas do nosso sistema), para compartilhar o mesmo driver entre elas:

```
class UsuariosPage {

    private WebDriver driver;

    public UsuariosPage(WebDriver driver) {
        this.driver = driver;
    }
```

```
    public void visita() {
        driver.get("localhost:8080/usuarios");
    }

    public void novo() {
        // clica no link de novo usuario
        driver.findElement(By.linkText("Novo Usuário")).click();
    }

    public boolean existeNaListagem(String nome, String email) {
        // verifica se ambos existem na listagem
        return driver.getPageSource().contains(nome) &&
                driver.getPageSource().contains(email);
    }

}
```

Excelente! Já conseguimos clicar no link de Novo Usuário e já conseguimos verificar se o usuário existe na página. Falta fazer agora o preenchimento do formulário. A pergunta é: onde devemos colocar esse comportamento? Na classe `UsuariosPage`? A grande ideia por trás do que estamos tentando fazer é criar uma classe para cada diferente página do nosso sistema! Dessa forma, cada classe ficará pequena, e esconderá todo o código responsável por usar a página. Ou seja, precisamos criar a classe `NovoUsuarioPage`.

Ela será muito parecida com nossa classe anterior. Ela também deverá receber o driver pelo construtor, e expor o método `cadastra()`, que preencherá o formulário e o submeterá:

```
class NovoUsuarioPage {

    private WebDriver driver;

    public NovoUsuarioPage(WebDriver driver) {
        this.driver = driver;
    }

    public void cadastra(String nome, String email) {
        WebElement txtNome = driver.findElement(By.name("usuario.nome"));
        WebElement txtEmail = driver.findElement(By.name("usuario.email"));

        txtNome.sendKeys(nome);
        txtEmail.sendKeys(email);

        txtNome.submit();

    }

}
```

Ótimo! Precisamos agora chegar nesse `NovoUsuarioPage`. Mas quando chegamos nela? Quando clicamos no link "Novo Usuário". Ou seja, o método `novo()`, depois de clicar no link, precisa retornar um `NovoUsuarioPage`:

```
    public NovoUsuarioPage novo() {
        // clica no link de novo usuario
```

```
            driver.findElement(By.linkText("Novo Usuário")).click();
            // retorna a classe que representa a nova pagina
            return new NovoUsuarioPage(driver);
        }
```

Agora, de volta ao nosso teste, temos o seguinte código:

```
public class UsuariosSystemTest {

    private WebDriver driver;
    private UsuariosPage usuarios;

    @Before
    public void inicializa() {
        this.driver = new FirefoxDriver();
        this.usuarios = new UsuariosPage(driver);
    }

    @Test
    public void deveAdicionarUmUsuario() {

        usuarios.visita();
        usuarios.novo()
        .cadastra("Ronaldo Luiz de Albuquerque", "ronaldo2009@terra.com.br");

        assertTrue(usuarios.existeNaListagem(
                "Ronaldo Luiz de Albuquerque", "ronaldo2009@terra.com.br"));

    }

    @After
    public void encerra() {
        driver.close();
    }

}
```

Veja só como nossos testes ficaram mais claros! E o melhor, eles não conhecem a implementação por baixo de cada uma das páginas! Essa implementação está escondida em cada uma das classes específicas!

Sabemos que nosso HTML muda frequentemente. Se tivermos classes que representam as nossas várias páginas do sistema, no momento de uma mudança de HTML, basta que mudemos na classe correta, e os testes não serão afetados! Esse é o poder do encapsulamento, um dos grandes princípios da programação orientada a objetos, bem utilizada em nossos códigos de teste.

A idéia de escondermos a manipulação de cada uma das nossas páginas em classes específicas é inclusive um padrão de projetos. Esse padrão é conhecido por Page Object. Pense em escrever Page Objects em seus testes. Eles garantirão que seus testes serão de fácil manutenção por muito tempo.