Testes de Sistema - parte 1

Site:	Moodle institucional da UTFPR	Impresso por:	MARLENE VASCONCELOS MORAES DE OLIVEIRA
Curso:	CETEJ158A - Teste de Software - J29 (2023_03) J30A (2024_01)	Data:	domingo, 8 set. 2024, 18:11
Livro:	Testes de Sistema - parte 1		

Descrição

Estamos muito acostumados a testar nossas aplicações de maneira manual. Empresas geralmente possuem imensos roteiros de script, e fazem com que seus analistas de teste executem esses scripts incansavelmente. Mas quais os problemas dessa abordagem?

Índice

1. Introdução

Imagine agora um sistema web que toma conta de leilões, domínio com que trabalhamos até agora. Nele, o usuário da empresa pode adicionar novos usuários, cadastrar leilões e efetuar lances. Essa aplicação foi desenvolvida em Java e está pronta para ser executada.

A aplicação não é muito grande: possui cadastro de usuários, leilões e lances. Mas, apesar da pequena quantidade de funcionalidades, pense na quantidade de cenários que você precisa testar para garantir seu funcionamento.

Se pensarmos em todos os cenários que devemos testar, percebemos que teremos uma quantidade enorme! Quanto tempo uma pessoa leva para executar todos esses cenários? Agora imagine o mesmo problema em uma aplicação grande. Testar aplicações grandes de maneira manual leva muito tempo! E, por consequência, custa muito caro.

Na prática, o que acontece é que, como testar sai caro, as empresas optam por não testar! No fim, entregamos software com defeito para nosso cliente! Precisamos mudar isso. Se removermos a parte humana do processo e fizermos com que a máquina execute o teste, resolvemos o problema: a máquina vai executar o teste rápido, repetidas vezes, e de graça!

A grande questão é: como ensinar a máquina a fazer o trabalho de um ser humano? Como fazê-la abrir o browser, digitar valores nos campos, preencher formulários, clicar em links etc.?

Para isso, faremos uso do Selenium. O Selenium é um framework que facilita e muito a vida do desenvolvedor que quer escrever um teste automatizado. A primeira coisa que uma pessoa faria para testar a aplicação seria abrir o browser. Com Selenium, precisamos apenas da linha a seguir:

```
WebDriver driver = new FirefoxDriver();
```

Nesse caso, estamos abrindo o Firefox! Em seguida, entraríamos em algum site. Vamos entrar no Google, por exemplo, usando o método get():

```
driver.get("http://www.google.com.br/");
```

Para buscar o termo “UTFPR”, precisamos digitar no campo de texto. No caso do Google, o nome do campo é “q”.

Para descobrir, podemos fazer uso do Inspector, no Chrome (ou do Firebug no Firefox). Basta apertar Ctrl + Shift + I (ou F12), e a janela abrirá. Nela, selecionamos a lupa e clicamos no campo cujo nome queremos descobrir. Ele nos levará para o HTML, onde podemos ver name="q".

Com Selenium, basta dizermos o nome do campo de texto, e enviarmos o texto:

```
WebElement campoDeTexto = driver.findElement(By.name("q"));
campoDeTexto.sendKeys("UTPFR");
```

Agora, basta submetermos o form! Podemos fazer isso pelo próprio campoDeTexto:

```
campoDeTexto.submit();
```

Pronto! Juntando todo esse código em uma classe Java simples, temos:

```
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.firefox.FirefoxDriver;

public class TesteAutomatizado {

    public static void main(String[] args) {
// abre firefox
        WebDriver driver = new FirefoxDriver();
// acessa o site do google
        driver.get("http://www.google.com.br/");
// digita no campo com nome "q" do google
        WebElement campoDeTexto
                = driver.findElement(By.name("q"));
        campoDeTexto.sendKeys("UTFPR");
// submete o form
        campoDeTexto.submit();
    }
}
```

Se você não entendeu algum método invocado, não se preocupe. Vamos estudá-los com mais detalhes nos próximos capítulos. Nesse momento, rode a classe! Acabamos de fazer uma busca no Google de maneira automatizada!

Execute o teste novamente. Veja agora como é fácil, rápido e barato! Qual a vantagem? Podemos executá-los a tempo! Ou seja, a cada mudança que fazemos em nosso software, podemos testá-lo por completo, clicando apenas em um botão. Saberemos em poucos minutos se nossa aplicação continua funcionando!

Quantas vezes não entregamos software sem tê-lo testado por completo?

2. Automatizando o primeiro teste de sistema

Imagine uma tela de cadastro padrão, onde o usuário deve preencher um formulário qualquer. E que, ao clicar em “Salvar”, o sistema devolve o usuário para a listagem com o novo usuário cadastrado.

A funcionalidade funciona atualmente, mas nossa experiência nos diz que futuras alterações no sistema podem fazer com que a funcionalidade pare. Vamos automatizar um teste para o cadastro de um novo usuário. O cenário é o mesmo que faríamos ao testar de maneira manual.

```
public static void main(String[] args) {

}
```

A primeira parte do teste manual é entrar na página de cadastro de usuários. Vamos fazer o Selenium abrir o Firefox nessa página. Suponha que o endereço seja http://localhost:8080/usuarios/new:

```
WebDriver driver = new FirefoxDriver();
driver.get("http://localhost:8080/usuarios/new");
```

Nessa página, precisamos cadastrar algum usuário, por exemplo, “Ronaldo Luiz de Albuquerque” com o e-mail “ronaldo2009@terra.com.br”. Para preencher esses valores de maneira automatizada, precisamos saber o nome dos campos de texto para que o Selenium saiba aonde colocar essa informação!

Aperte CTRL + U (no Firefox e Chrome) ou Ctrl + F12 (no Internet Explorer) para exibir o código-fonte da página. Veja que o nome dos campos de texto são “usuario.nome” e “usuario.email”. Com essa informação em mãos, precisamos encontrar esses elementos na página e preencher com os valores que queremos:

```
// encontrando ambos elementos na pagina
WebElement nome = driver.findElement(By.name("usuario.nome"));
WebElement email = driver.findElement(By.name("usuario.email"));
// digitando em cada um deles
nome.sendKeys("Ronaldo Luiz de Albuquerque");
email.sendKeys("ronaldo2009@terra.com.br");
```

Veja o código. Para encontrarmos um elemento, utilizamos o método driver.findElement. Como existem muitas maneiras para encontrar um elemento na página (pelo id, nome, classe CSS etc.), o Selenium nos provê uma classe chamada By que tem um conjunto de métodos que nos ajudam a achar o elemento. Nesse caso, como queremos encontrar o elemento pelo seu nome, usamos By.name("nome-aqui").

Tudo preenchido! Precisamos submeter o formulário! Podemos fazer isso de duas maneiras. A primeira delas é clicando no botão que temos na página. Ao olhar o código-fonte da página novamente, é possível perceber que o id do botão de Salvar é btnSalvar. Basta pegarmos esse elemento e clicar nele:

```
WebElement botaoSalvar = driver.findElement(By.id("btnSalvar"));
botaoSalvar.click();
```

Uma outra alternativa mais simples ainda é “submeter” qualquer uma das caixas de texto! O Selenium automaticamente procurará o form na qual a caixa de texto está contida e o submeterá! Ou seja:

```
nome.submit();
// email.submit(); daria no mesmo!
```

Se tudo der certo, voltamos à listagem de usuários. Mas, desta vez, esperamos que o usuário Ronaldo esteja lá. Para terminar nosso teste, precisamos garantir de maneira automática que o usuário adicionado está lá. Para fazer esses tipos de verificação, utilizaremos um framework muito conhecido do mundo Java, que é o JUnit. O JUnit nos provê um conjunto de instruções para fazer essas comparações e ainda conta com um plugin que nos diz se os testes estão passando ou, caso contrário, quais testes estão falhando!

Para garantir o usuário na listagem, precisamos procurar pelos textos “Ronaldo Luiz de Albuquerque” e “ronaldo2009@terra.com.br” na página atual. O Selenium nos dá o código-fonte HTML inteiro da página atual, através do método driver.getPageSource(). Basta verificarmos se existem o nome e e-mail do usuário lá:

```
boolean achouNome = driver.getPageSource().contains("Ronaldo Luiz de Albuquerque");
boolean achouEmail = driver.getPageSource().contains("ronaldo2009@terra.com.br");
```

Sabemos que essas duas variáveis devem ser iguais a true. Vamos avisar isso ao JUnit através do método assertTrue() e, dessa forma, caso essas variáveis fiquem com false, o JUnit nos avisará:

```
assertTrue(achouNome);
assertTrue(achouEmail);
```

Lembre-se que, para o método assertTrue funcionar, precisamos fazer o import estático do método import static org.junit.Assert.assertTrue;.

Devemos agora encerrar o Selenium:

```
driver.close();
```

Por fim, para que o JUnit entenda que isso é um método de teste, precisamos mudar sua assinatura. Todo método do JUnit deve ser público, não retornar nada, e deve ser anotado com @Test. Veja:

```
@Test
public void deveAdicionarUmUsuario() {
// ...
}
```

Observe que usamos o nome do método para explicar o que ele testa. Essa é uma boa prática.

Nosso método agora ficou assim:

```
import static org.junit.Assert.assertTrue;
import org.junit.Test;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.firefox.FirefoxDriver;

public class UsuariosSystemTest {

    @Test
    public void deveAdicionarUmUsuario() {
        WebDriver driver = new FirefoxDriver();
        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();
        boolean achouNome = driver.getPageSource()
                .contains("Ronaldo Luiz de Albuquerque");
        boolean achouEmail = driver.getPageSource()
                .contains("ronaldo2009@terra.com.br");
        assertTrue(achouNome);
        assertTrue(achouEmail);
        driver.close();
    }
}
```

O que acabamos de fazer é o que chamamos de teste de sistema. Ou seja, é um teste que exercita a aplicação do ponto de vista do usuário final, como um todo, sem conhecer seus detalhes internos. Se sua aplicação é web, então o teste de sistema é aquele que navegará pela interface web, interagir com ela, os dados serão persistidos no banco de dados, os serviços web serão consumidos etc.

A vantagem desse tipo de teste é que ele é muito parecido com o teste que o usuário faz. No teste de unidade, garantíamos que uma classe funcionava muito bem. Mas nosso software em produção faz uso de diversas classes juntas. É aí que o teste de sistema entra; ele garante que o sistema funciona quando “tudo está ligado”.

2.1. Novamente, as vantagens do teste automatizado

São várias as vantagens de termos testes de sistema automatizado:

- O teste automatizado é muito mais rápido do que um ser humano.
- A partir do momento em que você escreveu o teste, você poderá executá-lo infinitas vezes a um custo baixíssimo.
- Mais produtividade, afinal, você gastará menos tempo testando (escrever um teste automatizado gasta menos tempo do que testar manual- mente diversas vezes a mesma funcionalidade) e mais tempo desenvolvendo.
- Bugs encontrados mais rápido pois, já que sua bateria de testes roda rápido, você a executará a todo instante, encontrando possíveis partes do sistema que deixaram de funcionar devido a novas implementações.

  O Selenium é uma excelente ferramenta para automatizar os testes. Sua API é bem clara e fácil de usar, além da grande quantidade de documentação que pode ser encontrada na internet.

3. Boas práticas: Page Objects

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 a seguir:

```
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 desta 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. Repare que a declaração da variável usuarios foi omitida do código. A ideia é que essa variável representa “a página de usuários”. Ela contém as seguintes ações: novo() para ir para a página de novo usuário, e existeNaListagem(), que verifica se um usuário está lá.

Vamos escrever 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 usuário
        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 anteriormente, mas dessa vez os escondemos em uma classe específica. Só que esse código ainda não funciona. Precisamos do driver do Selenium. Mas, em vez 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 {

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

    public void novo() {
// clica no link de novo usuário
        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 con- seguimos verificar se o usuário existe na página. Falta fazer agora o preenchi- mento 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 isso acontece? 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 ideia 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.

4. Testando formulários complexos

Vamos continuar testando nossa aplicação; agora é a vez do cadastro de leilão. Veja que essa tela é bem mais complexa: ela contém combos, checkboxes etc. Será que o Selenium consegue lidar com todos esses elementos?

A resposta é: sim! Vamos lá! Vamos escrever nosso Page Object que lida com a tela de Novo Leilão. Para adicionarmos um novo leilão, precisamos passar as seguintes informações: nome, valor inicial, usuário e marcar se o objeto é usado.

Para preencher uma caixa de texto, já sabemos como fazer:

```
        WebElement nome = driver
                .findElement(By.name("leilao.nome"));
        WebElement valor = driver
                .findElement(By.name("leilao.valorInicial"));
        nome.sendKeys("Dono do Produto");
        valor.sendKeys("123,45");
```

Já o usuário é um combobox. Para selecionarmos uma opção no combo, devemos fazer uso da classe Select. Com esse objeto em mãos, podemos pedir para que ele selecione uma determinada opção. Por exemplo, se quisermos selecionar o usuário “João” (nome que aparece no combo), fazemos:

```
Select usuario = new Select(driver.findElement(By.name("leilao.usuario.id")));
usuario.selectByVisibleText("João");
```

Com o checkbox, é mais fácil. Basta clicarmos nele:

```
WebElement usado = driver.findElement(By.name("leilao.usado");
usado.click();
```

Pronto! Agora sabemos como preencher nosso formulário por inteiro. Vamos escrever nosso Page Object:

```
public class NovoLeilaoPage {

    private WebDriver driver;

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

    public void preenche(String nome, double valor, String usuario, boolean usado) {
        WebElement txtNome
                = driver.findElement(By.name("leilao.nome"));
        WebElement txtValor
                = driver.findElement(By.name("leilao.valorInicial"));

        txtNome.sendKeys(nome);

        txtValor.sendKeys(String.valueOf(valor));
        WebElement combo
                = driver.findElement(By.name("leilao.usuario.id"));
        Select cbUsuario = new Select(combo);

        cbUsuario.selectByVisibleText(usuario);
        if (usado) {
            WebElement ckUsado
                    = driver.findElement(By.name("leilao.usado"));
            ckUsado.click();
        }
        txtNome.submit();
    }
}
```

Excelente. Podemos usar a mesma estratégia do teste anterior: criar um Page Object para representar a listagem de leilões e chegar na tela de Novo Leilão por ela. Vamos também já colocar um método existe() para nos dizer se existe um produto nessa listagem:

```
class LeiloesPage {

    private WebDriver driver;

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

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

    public NovoLeilaoPage novo() {
// clica no link de novo leilão
        driver.findElement(By.linkText("Novo Leilão")).click();
// retorna a classe que representa a nova página
        return new NovoLeilaoPage(driver);
    }

    public boolean existe(String produto, double valor, String usuario, boolean usado) {
        return driver.getPageSource().contains(produto)
                && driver.getPageSource().contains(String.valueOf(valor))
                && driver.getPageSource().contains(usado ? "Sim" : "Não");
    }
}
```

Vamos agora ao nosso teste. Devemos cadastrar um leilão:

```
public class LeiloesSystemTest {

    private WebDriver driver;
    private LeiloesPage leiloes;

    @Before
    public void inicializa() {

        this.driver = new FirefoxDriver();
        leiloes = new LeiloesPage(driver);
    }

    @Test
    public void deveCadastrarUmLeilao() {
        leiloes.visita();
        NovoLeilaoPage novoLeilao = leiloes.novo();
        novoLeilao.preenche("Geladeira", 123, "Paulo Henrique", true);
        assertTrue(
                leiloes.existe("Geladeira", 123, "Paulo Henrique", true)
        );
    }
}
```

Mas o problema é: de onde virá esse usuário? Precisamos cadastrar o usuário “Paulo Henrique” antes de cadastrarmos um leilão para ele. Como faremos isso? Será uma boa ideia já termos alguns dados populados, prontos para os testes?

Uma boa prática de testes é fazer com que a bateria de testes seja inteiramente responsável por montar o cenário necessário para o teste. Dessa forma, cada teste sabe qual cenário precisa montar. Ou seja, vamos fazer com que um usuário seja adicionado antes de adicionarmos um novo leilão. Para isso, usaremos o LeiloesPage:

```java
public class LeiloesSystemTest {

    private WebDriver driver;
    private LeiloesPage leiloes;

    @Before
    public void inicializa() {
        this.driver = new FirefoxDriver();
        leiloes = new LeiloesPage(driver);
        UsuariosPage usuarios = new UsuariosPage(driver);
        usuarios.visita();
        usuarios.novo().cadastra(
                "Paulo Henrique",
                "paulo@henrique.com");
    }

    @Test
    public void deveCadastrarUmLeilao() {
        leiloes.visita();
        NovoLeilaoPage novoLeilao = leiloes.novo();
        novoLeilao.preenche("Geladeira", 123, "Paulo Henrique", true);
        assertTrue(
                leiloes.existe("Geladeira", 123, "Paulo Henrique", true)
        );
    }
}
```

Pronto. O teste passa. Veja como é fácil testar formulários complexos, com diferentes tipos de entradas de dados. Além disso, lembre-se sempre de fazer com que seus testes sejam independentes, ou seja, eles devem ser responsáveis por todo o processo, desde a criação do cenário até a validação da saída.

5. Classes de teste pai

Teste é código. E por que não colocar todo código que é idêntico entre as várias classes de teste em um único lugar para facilitar reuuso? Perceba que todo Page Object contém um driver, um método visita() e uma URL. Podemos isolar tudo isso em uma classe pai. Agora, se todas elas forem implementações dessas classes abstratas, todas elas terão os mesmos comportamentos e maneiras de funcionar:

```
abstract class PageObject {

    protected WebDriver driver;

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

    public abstract String url();

    public void visita() {
        driver.go(url());
    }
}
```

O mesmo pode ser feito com as classes de teste. Todo teste começa com um @Before que abre o driver e um @After que o fecha. Podemos colocar esses métodos em uma classe pai, e o JUnit, inteligentemente os rodará mesmo que eles estejam na classe pai:

```
abstract class TesteDeSistema {

    @Before
    public void inicializa() {
// inicializa webdriver
    }

    @After
    public void finaliza() {
// finaliza webdriver
    }
}
```

Reutilizar código de infraestrutura de testes é sempre uma boa ideia.

6. Referência Bibliográfica

Extraído de:
Aniche, M. **Testes Automatizados de Software**. São Paulo: Casa do Código, 2015.