Um estudo sobre as propriedades pilares do Cabelenium

Andrew Takeshi Tanaka de Vita

Orientador: Ricardo Terra

Departamento de Ciência da Computação
Universidade Federal de Lavras (UFLA)

andrew.vita@estudante.ufla.br

Abstract. *Due to the rapid technological growth, the development of web systems has become more sophisticated. In order to ensure the quality of the applications, Industry has been using automated test frameworks to perform functional tests and hence certifying that the system behaves as expected. In view of these circunstances, this paper focuses on Cabelenium, an automated functional testing framework, built on top of Selenium. We present the framework by describing the following core properties that motivates its adoption in a real-world software company: (i) portability, which is the ability to transfer the framework to other web-based environments; (ii) reuse, which allows a method to easily invoke any other method in the test project; (iii) data consistency, which allows us to ensure the data actually persisted in the database; (iv) test atomicity, which consists in tests are not leaving traces of its execution after they finish; and (v) traceability, which allows to textually monitor the events that occur during the test execution.*

Resumo. *Devido ao rápido crescimento tecnológico, a criação de sistemas web se tornou mais sofisticada. Dessa forma, para garantir o funcionamento e qualidade das aplicações, foi-se necessário utilizar frameworks de testes automatizados para realizar verificações dos dados, então, certificando que o sistema se comporta como originalmente projetado. Diante disso, este artigo é focado no Cabelenium, um framework de testes funcionais automatizados, criado a partir do Selenium. O objetivo deste documento é apresentar as propriedades pilares do framework Cabelenium que, motivaram a sua adoção na empresa em que se foi concebido: (i) portabilidade que é a capacidade de transferir o framework para outros ambientes de trabalho; (ii) reúso que permite um método invocar facilmente qualquer outro método do projeto de teste; (iii) consistência de dados que permite assegurar os dados de fato persistidos no banco de dados; (iv) atomicidade dos testes que consiste em um teste não deixar vestígios de sua execução após sua finalização; e, por fim, (v) rastreabilidade que permite monitorar textualmente os eventos que ocorrem durante a execução do teste.*

1. Introdução

Testes automatizados são de suma importância para garantir a integridade de uma aplicação (BERNARDO; KON, 2008). Desse modo, é comum o uso de *frameworks* de testes automatizados, pois garantem que o sistema está funcionando como originalmente projetado, apenas executando os testes logo após uma atualização (PFLEEGER, 2004). Diante disso, o Selenium é muito utilizado no desenvolvimento de testes automatizados por ser um *framework* de código aberto, por facilmente prover soluções diferentes de simulação de acessos a um navegador e por realizar verificações de resposta de uma aplicação (ATEŞOĞULLARI; MISHRA, 2020).

No entanto, em um sistema de testes, ao utilizar apenas o Selenium, pode surgir dificuldades em alguns aspectos, por exemplo, a geração de códigos sem padrão com a possibilidade de muitas variações. Por consequência, os testes ficam propensos a verbosidade e a redundância de operações (algoritmos iguais, porém feitos de formas diferentes). Logo, isso dificulta a manutenção, devido à extensão do código, e o reúso, pela falta de padrão (TERRA, 2020). Além disso, percebe-se que não basta apenas verificar se o fluxo de operações da aplicação corresponde com o que foi originalmente projetado, é preciso outras verificações (por exemplo, verificações de consistências no banco de dados).

Diante dos problemas supracitados, este artigo apresenta as seguinte propriedades pilares do Cabelenium, um *framework* de testes funcionais automatizados, baseado no Selenium: (i) portabilidade que é a capacidade de transferir o framework para outros ambientes de trabalho; (ii) reúso que permite um método invocar facilmente qualquer outro método do projeto de teste; (iii) consistência de dados que permite assegurar os dados de fato persistidos no banco de dados; (iv) atomicidade dos testes que consiste em um teste não deixar vestígios de sua execução após sua finalização; e, por fim, (v) rastreabilidade que permite monitorar textualmente os eventos que ocorrem durante a execução do teste. Em suma, as propriedades pilares são melhorias nos aspectos em que o Selenium de fato não se propõe a realizar. Desse modo, demonstra-se que a criação do Cabelenium contribui para complementar o Selenium.

O artigo está organizado com a seguinte estrutura: a Seção 2 introduz os conceitos e termos que permeiam por todo o trabalho. A Seção 3 expõe como usar o Cabelenium e seus componentes. A Seção 4 apresenta decisões que motivaram a concepção dos componentes. A Seção 5 apresenta trabalhos relacionados. Enfim, a Seção 6 relata os resultados obtidos e trabalhos futuros.

2. Revisão da Literatura

Esta seção apresenta o que significa teste de unidade (Seção 2.1), em seguida, teste funcional automatizado (Seção 2.2). Por fim, apresentação do Selenium e exemplo de sua utilização (Seção 2.3).

2.1. Teste de unidade

Teste de unidade é uma verificação de um funcionalidade ou componente de um sistema de software. Sua construção se baseia, por exemplo, em um método de uma classe, desenvolvido para executar uma funcionalidade ou componente da tela de um sistema web que deve retornar um resultado esperado e, assim realizar uma comparação se o valor recebido é o igual ou similar ao esperado. Dessa forma, tem-se uma garantia de que

essa unidade do software funciona como o planejado, não precisando testar manualmente enquanto o sistema está em funcionamento, bastando executar os testes de unidades e caso algum vir a falhar, evidenciará a parte do sistema que está com problemas (OLAN, 2003).

Diante disso, conforme apontado por Bernardo e Kon (2008), é o tipo de teste inicial em sistemas, o mais usado e o mais importante. Principalmente, quando se começa a desenvolver um sistema complexo que oferece diversas funcionalidades, da qual precisa que cada componente esteja em atividade, testando cada componente isoladamente primeiro, conforme Pfleeger (2004) diz. Assim sendo, o teste de unidade é essencial para que qualquer sistema tenha uma confiabilidade.

2.1.1. Teste de regressão

Conforme um sistema se desenvolve, é recomendado executar os testes de unidades que anteriormente já obtiveram sucesso. A prática de executar novamente os testes para verificar se os componentes do sistema ainda correspondem com o que foi originalmente projetado é chamado teste de regressão (WAHL, 1999).

De acordo com Pfleeger (2004), situações como fazer atualizações para tratar *bugs* ou adicionar uma nova funcionalidade são ideais para o uso de teste de regressão, pois verificará testes que previamente obtinham sucesso; continuam executando sem apresentar erros. Ao implementar teste de regressão, realiza-se com as etapas seguintes: 1) adição de código; 2) revisão: testando os componentes que o código inserido interage; 3) executar novamente os testes de unidades já criados (teste de regressão); e 4) testar os testes criados para a nova inserção de código.

Enfim, como visto, testes de regressão são frequentes para uma manutenção ativa dos sistemas e devem ser realizados sempre que houver uma atualização ou mudança de especificação (WAHL, 1999).

2.1.2. JUnit

Ao implementar testes de unidades, pode-se usar o *framework* JUnit[1], desenvolvido para linguagem Java. Assim, para executar um teste completo, são realizadas duas etapas: 1) a configuração do teste, ou seja, preparo de um objeto ou retorno de uma função, por exemplo; e 2) a execução do próprio teste. Também, faz-se análise e visualiza-se o resultado (RIEHLE, 2008).

O Código 1 ilustra um exemplo de teste com JUnit que verifica os *status* de um estoque. Primeiro há um preparo da configuração do teste, em que se cria um produto que tem por parâmetro o tipo do produto (linha 3), criação do estoque passando os dados por parâmetro o nome do armazém e a quantidade limite de produtos (linha 4) e inserindo um produto no estoque (linha 5). Por fim, verifica se a capacidade do estoque está no limite (linha 6) e depois, tentou-se colocar mais um produto (linha 8). Por fim, verifica se o estoque contém mais do que é permitido (linha 9).

[1] https://junit.org/

Código 1. Exemplo de teste com JUnit

```
@Test
public void exemploEstoque() {
    Produto televisao = new Produto("televisao");
    Estoque est = new Estoque("Armazem1", 1);
    est.armazena(televisao);
    assertEquals("Cheio", est.getCapacidade());
    Produto estante = new Produto("estante");
    est.armazena(estante);
    assertEquals("Acima do permitido", est.getCapacidade());
}
```

2.2. Teste funcional automatizado

Teste funcional é um teste em que não há conhecimento sobre a operação interna do programa, o analista concentra-se nas funções que o software contemplará. Baseado na especificação, determinam-se as saídas esperadas para um determinado conjunto de dados (SILVA; ALVES; BRUNO, 2015) por isso, esses testes são de caixa-preta ou teste comportamental (PRESSMAN; MAXIM, 2014). Diferentemente do teste não-funcional que testa o desempenho da aplicação, verificando o que não está relacionado com as regras de negócio ou funcionalidades, o teste funcional, no entanto, é o começo do teste do sistema, em que não se precisa saber quais componentes estão sendo executados, mas como cada funcionalidade de um aplicativo se comporta (PFLEEGER, 2004). O objetivo é garantir que o software atenda a todos os requisitos especificados ou de componentes a serem testados (CORREIA; SILVA, 2004).

Teste automatizado é qualquer teste em que o seu funcionamento não requer a intervenção humana. Também podem ser utilizados para conhecer os efeitos colaterais de ferramentas e *frameworks* (BERNARDO, 2011. p. 221). Segundo Bernardo e Kon (2008, p. 2) , "testes automatizados são programas ou *scripts* simples que exercitam funcionalidades do sistema sendo testado e fazem verificações automáticas nos efeitos colaterais obtidos." Dessa forma, podem ser reproduzidos repetidamente quando houver possibilidade.

Por fim, teste funcional automatizado é o termo usado para descrever o processo de criação de *scripts* de testes automatizados. Esse procedimento normalmente é realizado usando um software projetado para capturar e monitorar o teste de software. Antes do desenvolvimento da abordagem de teste funcional automatizado, os desenvolvedores de software dependiam de testes manuais em aplicativos de software (GROCEVS; PROKOFJEVA, 2016).

2.3. Selenium

Selenium[2] é um *framework* de teste funcional automatizado que usa *scripts* para execução de testes diretamente dentro de um navegador. Ele facilita o uso, pois pode ser feito em diversas linguagens (e.g., Java, Perl, C#, Ruby e Python), a partir de *scripts*, possibilitando a montagem de cenários de testes automatizados em várias plataformas como Linux, Windows e MacOS e ser usado em vários navegadores como Firefox, Chrome e Edge (RAMYA; SINDHURA; SAGAR, 2017).

O *framework* oferece uma interface amigável para criar e executar suítes de testes, uma estrutura robusta, flexível e extensível que suporta automação de testes em diversos

[2]https://www.selenium.dev

conjuntos de sistemas web em diversos domínios. Geração programática de casos de teste funcionais, integração para agrupamento dinâmico e flexível de suítes de teste (ANGMO; SHARMA, 2014).

De acordo com Wang (2010), o Selenium pode gerar testes que ao estarem executando, ocorre uma simulação tal como seria de um usuário de um sistema web, por exemplo, estar logando em um sistema web e acessar interagir com os elementos da tela.

Como visto, o Selenium por ter essa capacidade de realizar testes funcionais automatizados permite que se faça testes de regressão, fazendo-se repetição dos testes criados para atualizações posteriores em um sistema. No entanto, conforme se desenvolve um sistema, os testes de uma tela de sistema web podem ficar extensos, por exemplo, operações simples como a criação e busca no sistema, ou até senão repetitivo. Principalmente se for usar o recurso *record and playback*, que gera códigos desnecessários, desse modo atrapalhando no reúso (ATEŞOĞULLARI; MISHRA, 2020; TRABOUSSY, 2021).

O Código 2 apresenta um teste funcional automatizado desenvolvido com Selenium que é um exemplo do que foi visto no Código 1. São realizadas operações de *login* de usuário (linhas 3-5), em seguida criação de um produto (linhas 7-10), criação de um estoque (linhas 12-17), logo após, inserção do produto criado (linhas 19-27) e verificação da capacidade do armazém (linhas 29-33). Por fim, adição de mais um produto e verificação do estoque novamente (linhas 35-43).

Código 2. Exemplo de teste funcional automatizado com Selenium

```
1  @Test
2  public void testVincular(){
3    DRIVER.get("http://teste_armazem.union");
4    DRIVER.findElement(By.className("input-email")).sendKeys("user@test.com",
         Key.TAB);
5    DRIVER.findElement(By.className("input-pass")).sendKeys("", Key.RETURN);
6
7    DRIVER.findElement(By.id("produto")).click();
8    DRIVER.findElement(By.id("novo")).click();
9    DRIVER.findElement(By.id("nomeProd")).sendKeys("televisao");
10   DRIVER.findElement(By.id("salvarProd")).click();
11
12   DRIVER.findElement(By.id("armazem")).click();
13   DRIVER.findElement(By.id("novo")).click();
14   DRIVER.findElement(By.id("nomeArmazem")).sendKeys("Armazem1");
15   DRIVER.findElement(By.id("Capacidade")).sendKeys("1");
16   DRIVER.findElement(By.id("salvar")).click();
17   assertEquals("Confirmado", DRIVER.findElement(By.id("popUp_msg")).getText());
18
19   DRIVER.findElement(By.id("armazem")).click();
20   DRIVER.findElement(By.id("nomeArmazem")).sendKeys("Armazem1");
21   DRIVER.findElement(By.id("pesquisar")).click();
22   DRIVER.findElement(By.id("elementoTabela")).click();
23   DRIVER.findElement(By.id("addProduto")).click();
24   DRIVER.findElement(By.id("input-prod")).sendKeys("tv", Key.RETURN);
25   DRIVER.findElement(By.id("elementoProd")).click();
26   DRIVER.findElement(By.id("salvar")).click();
27   assertEquals("Confirmado", DRIVER.findElement(By.id("popUp_msg")).getText());
28
29   DRIVER.findElement(By.id("armazem")).click();
30   DRIVER.findElement(By.id("nomeArmazem")).sendKeys("Armazem1");
31   DRIVER.findElement(By.id("pesquisar")).click();
32   DRIVER.findElement(By.id("elementoTabela")).click();
33   assertEquals("Cheio", DRIVER.findElement(By.id("status")).getText());
34
35   DRIVER.findElement(By.id("produto")).click();
36   DRIVER.findElement(By.id("novo")).click();
37   DRIVER.findElement(By.id("nomeProd")).sendKeys("estante");
```

```
38      DRIVER.findElement(By.id("salvarProd")).click();
39      DRIVER.findElement(By.id("addProduto")).click();
40      DRIVER.findElement(By.id("input-prod")).sendKeys("estante", Key.RETURN);
41      DRIVER.findElement(By.id("elementoProd")).click();
42      DRIVER.findElement(By.id("salvar")).click();
43      assertEquals("Mais do que permitido",
            DRIVER.findElement(By.id("status")).getText());
44  }
```

Como visto, pelo exemplo, facilitaria a programação dos testes com a criação de métodos que possibilite o reúso, conforme a funcionalidade. E apesar de não ser um problema do projeto do Selenium, ter a liberdade de elaborar *scripts* de testes sem um padrão, no entanto, para um ambiente arquitetural, isso o deterioraria. Por isso, existem *frameworks* como o Galen Framework (PRAZINA et al., 2019) e o Cabelenium (TERRA, 2020) que trazem módulos de funcionalidades padronizadas, evitando flexibilidades de variações e repetições blocos de códigos.

3. CABELENIUM

Esta seção é dedicada a explicar o que é o Cabelenium, os seus principais componentes e como pode ser usado. As informações contidas nesta seção são em sua maioria advindas do artigo de Traboussy (2021) modificadas para adequação do propósito deste trabalho acadêmico.

Inspirado e baseado no Selenium (e em linguagem Java), o *framework* Cabelenium foi desenvolvido para remover a flexibilidade que pode favorecer o aparecimento de problemas, principalmente relacionados à manutenibilidade e à evolução dos testes (TERRA, 2020). Mais importante, o Cabelenium promove um desenvolvimento orientado à funcionalidade (termo explicado na Seção 3.3) e provê suporte para consistência em banco de dados (detalhado na Seção 3.5).

3.1. Exemplo Motivador

Esta seção ilustra um exemplo motivador de como o Cabelenium pode ser utilizado em um sistema web básico de cadastro de usuários que fundamentalmente gerencia pessoas físicas.

O Código 3 ilustra um teste funcional usando o Cabelenium, em que há presença da anotação `@Clean` que significa que ocorrerá uma limpeza antes e depois da execução desse teste, dos dados de pessoa física, desse modo, os dados criados na execução serão apagados (linhas 4-6). E também, há um sistema de auditoria que exibe mensagens de *logs* (ou registros de execução) nos testes (linhas 8, 17, 22, 26 e 30), depois é feito o cadastro de uma pessoa (linhas 10-16). Em seguida, foi verificada a consistência no banco de dados (linhas 18-20). Nas linhas 23 a 25, foi feita a busca do usuário para colocar em modo de edição. Assim, pode-se alterar o nome e salvar conferindo a mensagem na tela (linhas 27-29) e, por fim, validar o dado com a consistência de dados (linhas 31-33).

Código 3. Exemplo de Teste Funcional com Cabelenium

```
1  public class PessoaTestCase extends PlugWebRunnerTestCase {
2      ...
3      @Test
4      @Clean({
5          "PessoaFisica:76965738022"
6      })
7      public void fluxoSimples() {
```

```
8         LOGGER.trace("Iniciando processo de criar pessoa física");
9         Long cpfInternal = 76965738022L;
10        this.asYouDesire("novo")
11            .addParam("nome", "Tester")
12            .addParam("cpf", cpfInternal)
13            .addParam("conhecido", "Teste novo")
14            .addParam("numeroRG", "40.966.641-8")
15            .addParam("email", "teste@associate.com.br")
16            .run();
17        LOGGER.trace("Realizando consistência de dados do usuário");
18        this.assertResultInDBMS(new SQL("pessoafisica001.sql")
19            .addParam("cpf", 76965738022L), 76965738022L, "Tester", "Teste novo",
20                    "teste@associate.com.br", 409666418);
21
22        LOGGER.trace("Buscando pessoa e colocando em modo de edição");
23        this.asYouDesire("modoEdicao")
24            .addParam("cpf", cpfInternal)
25            .run();
26        LOGGER.trace("Alterando o dado nome");
27        this.writeById("inputNome", "Tester editado");
28        this.click(Botao.SALVAR);
29        this.assertMessagePanelContainsText("Registro alterado com sucesso.");
30        LOGGER.trace("Realizando consistência de dados após alteração");
31        this.assertResultInDBMS(new SQL("pessoafisica001.sql")
32            .addParam("cpf", 76965738022L), 76965738022L, "Tester editado",
33                "Teste novo", "teste@associate.com.br", 409666418);
34    }
35    ...
36 }
```

A função `asYouDesire` faz chamada de métodos em qualquer classe do projeto de teste. Conforme mostrado no Código 3, os métodos `novo` e `modoEdicao`, respectivamente, nas linhas 10 a 16 e 23 a 25. Assim, pode-se observar a implementação da função `novo` no Código 4. Inicialmente, entra-se no menu de pessoa (linha 4), clica na opção para criação de um usuário (linha 5), coloca os dados nos campos (linhas 6-10), salva (linha 11) e verifica a mensagem no painel (linha 12). Na função `modoEdicao` busca-se o dado criado para ficar em modo de edição (linhas 16-19). As funções utilizadas são chamadas de *core*, elaboradas para interagir com *front-end* de um sistema web.

Código 4. Fragmento da classe `PessoaTestCase`

```
1  public class PessoaTestCase extends PlugWebRunnerTestCase {
2      ...
3      public void novo() {
4          this.openMenu(Menu.PESSOA);
5          this.click(Botao.NOVO);
6          this.writeById("inputNome", this.nome);
7          this.selecionarPF("btSel_fornecedor", this.cpf);
8          this.writeById("inputConhecido", this.conhecido);
9          this.selectOption("inputNumeroRg", this.numeroRG);
10         this.writeById("inputEmail", this.email);
11         this.click(Botao.SALVAR);
12         this.assertMessagePanelContainsText("Registro incluído com sucesso.");
13     }
14
15     public void modoEdicao(Long cpf) {
16         this.openMenu(Menu.PESSOA);
17         this.writeById("inputCpf", cpf);
18         this.click(Botao.PESQUISAR);
19         this.assertInputValueById("inputCpf", cpf);
20     }
21     ...
22 }
```

É importante salientar que o intuito dos exemplos é evidenciar a simplicidade na utilização do *framework*. Além disso, poder depreender as características próprias do

Cabelenium (Seção 3.2). Portanto, detalhes de como os processos internos são realizados, são expostos nas seções seguintes.

3.2. Propriedades pilares do Cabelenium

O Cabelenium possui características que são os pilares das motivações de suas melhorias com base no Selenium. Assim, são apresentadas e exemplificadas as seguintes propriedades pilares:

- *Portabilidade*: o objetivo é facilitar a transferência do *framework* para diferentes ambientes (MåRTENSSON, 2006). Isso faz com que seus componentes tenham fácil adaptação para interagir com o sistema web. No Cabelenium, primeiramente, deve-se utilizar suas funções próprias. Essa rigidez previne que não tenha testes acoplados diretamente com o Selenium, justamente para não comprometer a portabilidade. Por exemplo, conforme pode ser observado no Código 4, as implementações dos métodos `novo` (linhas 3-13) e `modoEdicao` (linhas 15-20) utilizam apenas das funções *core* do Cabelenium, isto é, não estabelecem nenhuma dependência direta com a API do Selenium. Logo, alteração no *front-end* da aplicação não traz impactos nos métodos supracitados, uma vez que a reimplementação dos métodos *core* para o novo *front-end* garante o devido funcionamento dos métodos de casos de testes.

- *Reusabilidade*: o objetivo é promover reúso. Para isso, testes, métodos e componentes criados podem ser utilizados em todo o ambiente do projeto de teste. Por exemplo, conforme o Código 3 nas linhas 8 a 14, a função `asYouDesire` pode ser utilizada em todo o projeto de teste e permite efetuar outras operações além do `novo`, por exemplo, `modoEdicao` (linhas 21-23). Isso significa que todos os métodos de uma classe de teste podem ser chamados em outros testes utilizando apenas a função `asYouDesire`. Outro detalhe, modificações internas nos métodos criados não afetam o reúso desde que mantenham-se inalterados os parâmetros e os atributos implementados.

- *Consistência de dados*: o objetivo é garantir a consistência e integridade dos dados salvos no banco de dados (BD). Por exemplo, conforme o Código 3, para certificar se os dados estão salvos e persistentes, a consistência de dados foi chamada para verificar os dados de uma pessoa criada (linhas 16-18) e posteriormente, os dados recém alterados de uma pessoa existente (linhas 29-31). Para isso, na função de consistência de dados, foi necessário informar a consulta `SQL` e atribuir os parâmetros de tal consulta. Por exemplo, na verificação se uma pessoa foi devidamente persistida (linhas 16-18), é informado que a consulta `SQL` está no arquivo "pessoafisica001.sql" e que o parâmetro cpf é o conteúdo da variável `cpfInternal`. Como resultado, espera-se uma única linha (quíntupla) com os valores informados.

- *Autocontido*: o objetivo é limpar todos os dados criados por um teste executado. Por exemplo, conforme o Código 3, para garantir a limpeza da pessoa física persistida no cenário de teste, utilizou-se da anotação `@Clean` (linhas 2-4) com valor da entidade e um identificador sendo "PessoaFisica:76965738022", indicando a

entidade (pessoa física) e seu identificador único (cpf). Logo, antes e depois da execução desse teste, tal limpeza é automaticamente realizada.

- *Rastreabilidade*: o objetivo é registrar informações de execução (*logs*), como escrita de campos, cliques, assertivas, consultas `SQL`, etc. Enfim, auxiliar na identificação do problema e a possível origem. Além disso, pode-se utilizar `LOGGER` para ter conhecimento da operação. Por exemplo, conforme o Código 3, o `LOGGER` destacou em cada parte do código, informações de operações realizadas posteriormente (linhas 6, 15, 20, 24 e 28); similarmente como comentários de um algoritmo.

3.3. Orientação à funcionalidade

O Cabelenium estipula como boa prática que, para cada funcionalidade do sistema, cria-se uma classe de teste em que:

- *Atributos*: representam os campos da tela e o atributo que identifica unicamente o registro principal da funcionalidade é anotado com `@Id`.

- *Métodos*: representam *métodos úteis* ou *métodos de caso de teste*. Por um lado, *métodos úteis* são aqueles que fomentam o reúso, por exemplo, `novo` que cria um novo registro preenchendo os campos da tela com os valores dos atributos e verificando se foi corretamente persistido e o método `edita` que busca um registro específico e a deixa em modo de edição. Por outro lado, *métodos de caso de teste* são aqueles que de fato proveem testes funcionais e são anotados com `@Test`.

O Código 5 ilustra um teste funcional em que o produto `TV` é cadastrado (linha 11) e 15 etiquetas são impressas (linha 12). Assim, são recebidas 15 unidades de `TV` do PJ01 (linha 14) e armazenadas no armazém (linha 15). Nesse ponto, verifica-se se tem-se 15 unidades de `TV` disponíveis no estoque (linha 16). Por fim, planeja-se uma expedição de 14 unidades de `TV` para o PJ02 (linha 18). Logo após finalizar o planejamento, verifica-se se tem-se uma única unidade de `TV` disponível no estoque e 14 reservadas (linha 19).

Código 5. Exemplo de Teste Funcional usando Json

```
1  @Test
2  @Clean({
3      "Produto:TV",
4      "ImprimirEtiqueta:TV",
5      "Recebimento:REC01",
6      "IniciarRecebimento:REC01",
7      "Armazenar:REC01",
8      "Expedicao:EXP01",
9  })
10 public void planejarExpedicao() {
11     this.asYouDesire(ProdutoTestCase.class, "novo").single("TV.json").run();
12     this.novaImpressaoEtiquetas("TV", "COBERTO", 15);
13
14     this.novoRecebimentoCompleto("REC01", "[1;TV;15]", "PJ01");
15     this.novoArmazenamentoCompleto("REC01");
16     this.assertResultInDBMS(new SQL("estoque.sql").addParam("codigo", "TV"), 15, 0);
17
18     this.novaExpedicao("EXP01", "[1;TV;14]", "COBERTO", "PJ02");
19     this.assertResultInDBMS(new SQL("estoque.sql").addParam("codigo", "TV"), 1, 14);
20 }
```

Na linha 11 do Código 5, é invocado o método `novo` da classe `ProdutoTestCase` passando o arquivo JSON descrito no Código 6.

Código 6. JSON de um Produto

```
1 {
2     "codigo"                          : "TV",
3     "nome"                            : "Televisão",
4     "cnpj"                            : 39789102000173,
5     "tipo"                            : "Inteiro"
6 }
```

Conforme pode ser observado no Código 7, o Cabelenium inicializa os atributos `codigo`, `nome`, `cnpj` e `tipo` e invoca o método útil `novo`.[3] A partir de métodos de maior abstração – porém usando as funcionalidades do Selenium – entra-se na função novo da funcionalidade (linhas 11-12), preenche os campos (linhas 14-18) e salva o registro (linha 20) confirmando mensagem em tela (linha 21) e na base de dados (linha 22). É importante ressaltar que todo esse processo é realizado na linha 11 do Código 5.

Código 7. Fragmentos da classe `ProdutoTestCase`

```
1  public class ProdutoTestCase extends PlugWebRunnerTestCase {
2      ...
3      @Id private String codigo;
4      private Long nome;
5      private Long cnpj;
6      private String tipo;
7      @Default private String unidade = "un";
8      ...
9
10     public void novo() {
11         this.openMenu(Menu.PRODUTO);
12         this.click(Botao.NOVO);
13
14         this.selectMany("checkTipoProduto", this.tipo);
15         this.writeById("inputNome", this.nome);
16         this.writeById("inputCodigo", this.codigo);
17         this.selectOption("comboUnidade", this.unidade);
18         this.selecionarPJ("btSel_fornecedor", this.cnpj);
19
20         this.click(Botao.SALVAR);
21         this.assertMessagePanelContainsText("Registro incluído com sucesso.");
22         this.assertResultInDBMS(new SQL("produto001.sql").addParam("codigo",
               this.codigo), this.nome, this.cnpj, this.tipo, this.unidade);
23     }
24
25     public void modoEdicao(String codigo) {
26         this.openMenu(Menu.PRODUTO);
27         this.writeById("inputCodigo", codigo);
28         this.click(Botao.PESQUISAR);
29         this.assertTextAtSearchResultTable(0, "Código", codigo);
30         this.clickOnDefaultSearchResultTable(0);
31         this.assertInputValueById("inputCodigo", codigo);
32     }
33     ...
34 }
```

Mais importante, o método `novo` é genérico o suficiente para ser reutilizado por diversas outras funcionalidades que requerem a criação de um produto, tal como o planejamento de uma expedição (ou qualquer função que se tenha produto) ilustrado na Seção 3.3 (referindo-se ao Código 5). Além disso, o método `modoEdicao` (linhas 25-32) pode

[3] Atributos que possuem um valor padrão devem ser inicializados e anotados com `@Default`, por exemplo, o campo de unidade na linha 7 do Código 7. Porém, caso um valor seja informado para o atributo, o valor padrão é sobrescrito.

ser utilizado por diversos casos de teste em que um produto deva ser alterado ou excluído, por exemplo. Por fim, vários outros método úteis podem ser desenvolvidos – tais como `excluir`, `inativar`, `transferir`, etc. – dependendo das particularidades do teste.

3.4. Funções utilitárias

A criação de casos de testes devem ser criados – sempre que possível – a partir de invocações de *métodos úteis* na própria classe e de outras classes. Essa prática fomenta o reúso.

No caso de rotinas complexas que sejam largamente utilizadas dentro de um sistema, o Cabelenium sugere que sejam generalizadas e se tornem *funções utilitárias*. O teste funcional do Código 5 utiliza de quatro *funções utilitárias* denominadas `novaImpressaoEtiquetas`, `novoRecebimentoCompleto`, `novoArmazenamentoCompleto` e `novaExpedicao`. Por exemplo, a *função utilitária* `novoRecebimentoCompleto` é um atalho para o planejamento de um recebimento e o processo de recebimento de fato de todos os itens previamente planejados, conforme pode ser observado no Código 8.

Código 8. Função utilitária `novoRecebimentoCompleto`

```
1  public void novoRecebimentoCompleto(String idRecebimento, String produtos, String nomeArmazem) {
2      this.novoPlanejamentoRecebimento(idRecebimento, produtos, nomeArmazem, 1);
3
4      Set<Integer> idEtiquetas = new HashSet<Integer>();
5      for (ProdutoInfo pi : this.getProdutosInfo(produtos)) {
6          idEtiquetas.addAll(Consultas.getEtiquetasPorProduto(pi.getCodigoProduto()));
7      }
8      this.asYouDesire(IniciarRecebimentoTestCase.class, "novo")
9                      .addParam("identificadorExterno", idRecebimento)
10                     .addParam("idEtiquetas", idEtiquetas)
11                     .run();
12 }
```

Mais detalhadamente, o Código 8 utiliza outra *função utilitária* que faz o planejamento (linha 2), busca todas as etiquetas de cada produto (linhas 5-7) e então invoca o *método útil* `novo` (que de fato faz o recebimento) da classe `IniciarRecebimentoTestCase` passando o identificador do recebimento e as etiquetas dos produtos que estão sendo recebidos (linhas 8-11).

É importante salientar que a prática de (i) *funções utilitárias* chamando outras *funções utilitárias* ou *métodos úteis* ou (ii) *métodos úteis* chamando outros *métodos úteis* ou mesmo *funções utilitárias* visam fomentar cada vez mais o reúso dentro do projeto de teste.

3.5. Consistência em banco de dados

A garantia de que o que é feito no sistema é corretamente persistido é algo crítico em diversos tipos de sistemas de software. O Cabelenium, por sua vez, provê nativamente formas de verificação em banco de dados.

Basicamente, o método `assertResultInDBMS` recebe como entrada um objeto SQL e um arranjo de arranjo de objetos que deveriam ser a lista de tuplas retornadas pelo Sistema Gerenciador de Banco de Dados (SGBD).

O teste funcional do Código 5 chama duas vezes o *script* `estoque.sql` com o parâmetro `codigo` sendo "TV" e esperando uma única tupla [15,0] como resultado.[4] Como

[4]Como o retorno é uma única tupla, o recurso varargs do Java permite que seja passado fora da definição de um arranjo de objetos. (ORACLE. . . , 2021)

pode ser observado pelo Código 9, o 15 refere-se à quantidade disponível e 0 à quantidade reservada por outros planejamentos de expedições.

Código 9. SQL de verificação de consistência de estoque

```
1 select QTDE_DISPONIVEL, QTDE_RESERVADA from ESTOQUE e
2             inner join PRODUTO p on e.ID_PRODUTO = p.ID
3         where
4             p.CODIGO = ':codigo'
```

É importante salientar que tanto as *funções utilitárias* quanto os *métodos úteis* devem incluir verificações de consistência genéricas. Isso implica em verificações de consistência sendo sempre realizadas em diferentes casos de teste o que provê uma maior confiabilidade nos testes. Por exemplo, toda vez ao se criar um produto (método útil `novo`, Código 7), é verificado se foi salvo conforme os valores informados (linha 22).

3.6. Casos de teste autocontidos

O Cabelenium prevê que todos os casos de testes sejam autocontidos. Isso garante que a execução do caso de teste não gere efeitos colaterais em outros testes.

Portanto, todos os casos de testes devem incluir uma anotação `@Clean` com tudo o que foi criado e deve ser excluído. Essa limpeza ocorre após a execução do teste para garantir que o sistema volte ao seu estado pré-teste. Por preciosismo, a limpeza ocorre também antes da execução para garantir não haver vestígios de uma falha não controlada durante uma execução prévia desse mesmo caso de teste ou de outros casos de testes que podem compartilhar mesmos nomes de identificadores únicos.

O teste funcional do Código 5 (linhas 2-9) tem seis *scripts* de testes sendo executados antes e depois de sua execução. Por exemplo, o "`Produto:TV`" indica que o Cabelenium executará a limpeza a partir do arquivo `produto-clean.sql` (Código 11) passando "`TV`" como parâmetro (`:codigo`). Particularmente para a limpeza de produto, são excluídos os registros de estoque (um para cada armazém) e depois é excluído o produto em si. É importante mencionar que, no decorrer do projeto, os *scripts* de limpeza tendem a crescer.

Código 10. SQL de limpeza de produto

```
1 --Apagando Estoque
2 delete from ESTOQUE where IDATIVO IN (
3     select p.ID from PRODUTO p where p.CODIGO = ':codigo'
4 );
5
6 --Apagando o produto em si
7 delete from PRODUTO where codigo = ':codigo';
```

Código 11. SQL de limpeza de produto

```
1 select  pf.NUMERO_IDENTIFICACAO, p.NOME, p.CONHECIDO, p.EMAIL, pf.REGISTRO_GERAL
2     from PESSOA p
3     inner join PESSOAFISICA pf on pf.IDPESSOA=p.ID
4     where
5         pf.CPF = :cpf
```

Ainda mais importante, a limpeza permite otimizar correções. Tomando ainda como exemplo o teste funcional do Código 5, assuma que o teste falhe e o desenvolvedor corrija um *bug* na tela de planejamento de expedição. Ao invés de executar todo o teste para uma aferição temporária, ele pode executar apenas o *clean* de planejamento de expedição (linha 8) e reexecutar somente as linhas 18 a 19.

4. Decisões

Esta seção apresenta as decisões de projeto do *framework* Cabelenium em relação às propriedades pilares (Seção 3.2). Inicialmente, são apresentadas as motivações de projeto da portabilidade de *front-end* (Seção 4.1), as decisões em orientação à funcionalidade (Seção 4.2), as verificações de bancos de dados (Seção 4.3), testes autocontidos (Seção 4.4) e, por fim, *logs* do sistema (Seção 4.5).

4.1. Portabilidade de *front-end*

O Cabelenium possui um conjunto de funções mínimas, denominadas funções *core*, que contemplam todo o espectro de iterações entre usuário e sistema. Essas funções interagem diretamente com a interface web, o que implica em um acoplamento direto com a tecnologia utilizada no *front-end*. O contrato de portabilidade do Cabelenium estabelece que todos os testes devem ser elaborados com o uso apenas de funções *core*. Isso garante que, em caso de troca de tecnologia utilizada no *front-end*, a implementação dos testes existentes não sofrerão impactos, desde que sejam reimplementadas as funções da Tabela 1 para a nova tecnologia.

Tabela 1. Funções *core*, com descrição de seu funcionamento interno.

Método	Descrição
login ()	A partir da *url* de *login*, configurada no arquivo de propriedades[9], escreve nos campos (tipo *input*) o nome do usuário e senha para então realizar e o *login*.
logout ()	Obtém a *url* de *logout* (do arquivo de propriedades[9]) para sair da sessão corrente.
openMenu (*Menu*[10])	Abre a página web com o dado *id*, contido na estrutura de dados *Menu*.
assertContainsTextById (*id*, *text*)	Verifica se o texto *text* está contido dentro do elemento HTML identificado pelo atributo *id*.
assertInputValueById (*id*, *text*)	Verifica se o texto *text* é estruturalmente igual ao conteúdo do elemento HTML identificado pelo atributo *id*.
writeById (*id*, *text*)	Escreve o texto *text* dentro do elemento HTML (tipo *input*) identificado pelo atributo *id*.
selectOption (*id*, *text*)	Seleciona a opção com o rótulo *text* do elemento HTML (tipo *combo*) identificado pelo atributo *id*.
selectMany (*id*, *text*...[11])	Seleciona as opções com rótulo(s) $text_1$, $text_2$, ..., $text_n$ do elemento HTML (tipo *checkbox*) identificado pelo atributo *id*.
getTextById (*id*)	Obtém texto dentro do elemento HTML identificado pelo atributo *id*.
clickByClassName (*className*)	Clica no elemento HTML que contém a classe *classeName*.
clickById (*id*)	Clica no elemento HTML identificado pelo atributo *id*.
exists (*id*)	Verifica se há ao menos um elemento HTML identificado pelo atributo *id*.
isEnabled (*id*)	Verifica se o elemento HTML identificado pelo atributo *id* está habilitado.
waitForLoadingJustInCase ()	Aguarda até que não existam funções em execução em segundo plano.
click (*Botao*[12])	Clica no elemento HTML que contenha o dado *id*, contido na estrutura de dados *Botao*.
getInputValueById (*id*)	Obtém o conteúdo dentro do elemento HTML identificado pelo atributo *id*.
clearById (*id*)	Apaga todo o texto do elemento HTML (tipo *input*) identificado pelo atributo *id*.
uploadFile (*id*, *filename*)	Envia o arquivo *filename* para elemento HTML (tipo *file*) identificado pelo atributo *id*.

É provável que algum sistema web tenha *loading* durante a execução dos testes. Assim, é recomendado que funções que utilizam *assert* (que realizam verificação de valores), contenham uma implementação que se previnam dessa eventualidade. Por exemplo, inserir no código o método `waitForLoadingJustInCase` antes de alguma interação que

envolva busca ou verificação de algum elemento HTML pode evitar o problema do elemento HTML, no momento, não estar acessível.

O Código 12 ilustra um teste de CRUD de um sistema de uma loja que cria um produto (linhas 5–10) e verifica a mensagem de confirmação na tela (linha 11). Em seguida, busca o produto criado para editar o nome e verifica a mensagem de confirmação de alteração (linhas 14–20). Por fim, exclui o produto e verifica a mensagem de confirmação de exclusão (linhas 23–28).

Código 12. Exemplo de teste automatizado de CRUD com produto

```
1  @Test
2  @Clean({ "Produto:CELULAR-08" })
3  public void fluxoSimples() {
4          LOGGER.trace("Cria produto");
5          this.openMenu(Menu.PRODUTO);
6          this.click(Botao.NOVO);
7          this.writeById("inputCpf", "333.444.343-08");
8          this.writeById("inputIdentExterna", "CELULAR-08");
9          this.writeById("inputNome", "Celular Z8 zony");
10         this.clickById("btConfirmar");
11         this.assertMessagePanelContainsText("Produto salvo com sucesso");
12 }
```

Conforme o Código 12, em uma eventual troca de tecnologia do *front-end*, o teste continuará funcionando. Para isso, basta que sejam implementadas as funções *core* na nova tecnologia exatamente como descritas na Tabela 1.

Como um outro exemplo, o Código 13 ilustra um teste funcional de CRUD de um sistema de uma biblioteca, em que se cadastra um livro (linhas 5–10) e verifica a mensagem de confirmação na tela (linha 11). Em seguida, busca o livro recém criado para editar o título e verifica a mensagem de confirmação de alteração (linhas 14–20). Por fim, exclui o livro e verifica a mensagem de confirmação de exclusão (linhas 23–29).

Código 13. Exemplo de teste automatizado de CRUD de livro

```
1  @Test
2  @Clean({ "Livro:123146" })
3  public void fluxoSimples() {
4          LOGGER.trace("Cadastrar um livro");
5          this.openMenu(Menu.LIVROS);
6          this.click(Botao.NOVO);
7          DRIVER.findElement(By.id("inputNumeroSerie")).sendKeys(123146L, Key.TAB);
8          DRIVER.findElement(By.id("inputTitulo")).sendKeys("Lusíadas", Key.TAB);
9          DRIVER.findElement(By.id("inputAutor")).sendKeys("Luiz Vaz de Camões");
10         this.clickById("btConfirmar");
11         this.assertMessagePanelContainsText("Registro salvo com sucesso");
12
13         LOGGER.trace("Buscar e editar o título do livro");
14         this.openMenu(Menu.LIVROS);
15         DRIVER.findElement(By.id("inputNumeroSerie").sendKeys( 123146L, Key.TAB);
16         this.click(Botao.PESQUISAR);
17         this.assertInputValueById("inputNumeroSerie", 123146L);
18         DRIVER.findElement(By.id("inputTitulo")).sendKeys( "Os Lusíadas");
19         this.clickById("btConfirmar");
20         this.assertMessagePanelContainsText("Registro alterado com sucesso.");
21
```

[9]Os dados são oriundos do arquivo de propriedades que contém informações do sistema web como o endereço do sistema, o endereço do banco de dados e senha do usuário.

[10]As informações *id* e título, são providos pelo Menu do tipo enum.

[11]Representa um arranjo de textos.

[12]A informação *id* é provida pelo do tipo enum Botao.

```
22          LOGGER.trace("Excluir um Livro");
23          this.openMenu(Menu.LIVROS);
24          DRIVER.findElement(By.id("inputNumeroSerie")).sendKeys(123146L);
25          this.click(Botao.PESQUISAR);
26          this.assertInputValueById("inputNumeroSerie", 123146L);
27          this.click(Botao.EXCLUIR);
28          this.clickById("btConfirmar");
29          this.assertMessagePanelContainsText("Registro excluído com sucesso.");
30  }
```

Nos exemplos supramencionados, é possível notar que os casos de teste desenvolvidos apenas com funções *core* continuarão funcionando em caso de troca da tecnologia de *front-end*. Porém, no Código 13, a equipe de qualidade terá dificuldades em todas as eventuais trocas de tecnologia, justamente pelo acoplamento direto com o Selenium. Logo, não apenas reimplementará as funções *core*, como também terá de refazer as operações, como inserir dados nos campos do tipo *input* (linhas 7, 8, 9, 15, 18 e 24), conforme a nova tecnologia de *front-end*.

Enfim, conforme pode ser observado, o Código 13 é similar ao Código 12, porém utilizado em um sistema diferente. Assim, pode-se concluir que o Cabelenium pode ser portado para outros sistemas, justamente pela propriedade de portabilidade das funções *core*. Pois, para realizar os testes em outros ambientes, basta utilizá-las e, assim, estarão realizando tais operações e verificações, como o exemplo do Código 12, visto que, a prática supracitada do Código 13 rompe com o contrato de portabilidade.

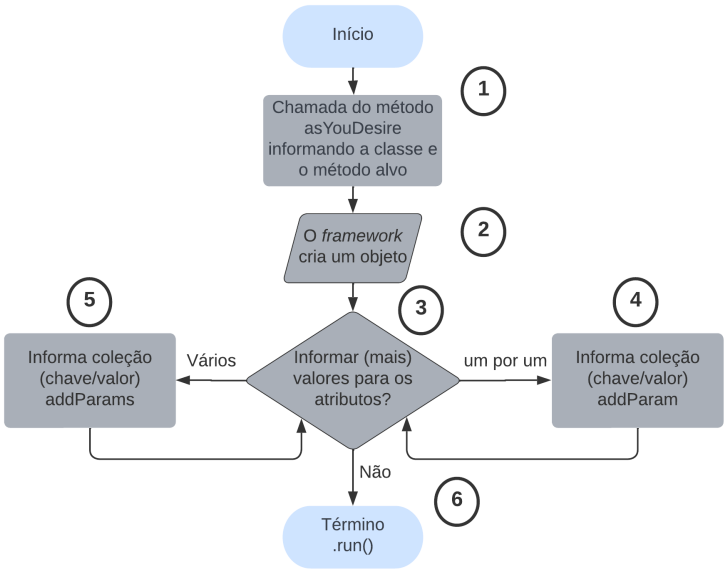
Mais importante, pode-se concluir que o Cabelenium é um *framework* de caixa-cinza, ou seja, ocorre um balanceamento de caixa-preta e caixa-branca, extensão e flexibilidade (KRAJNC; HERICKO, 2003). Para portar o *framework* para outra tecnologia de *front-end*, basta implementar os métodos *core* e os demais componentes continuam inalterados, i.e., nenhum impacto no seu funcionamento. É importante mencionar que é possível inclusive desacoplar-se do Selenium (como supracitado).

4.2. Orientação à funcionalidade

O Cabelenium possui um método denominado `asYouDesire` que permite chamar métodos de outras classes do projeto de teste. O objetivo é promover o reúso dos *métodos de casos de testes (ou úteis)*, principalmente, os métodos `novo` e `modoEdicao`. Isso garante que, além de conseguir reutilizar métodos de outras classes, alterações internas nesses métodos não geram impactos nos testes, desde que não se alterem os atributos da classe (Seção 3.2).

O fluxograma da Figura 1 ilustra a sequência de passos que ocorre quando se faz uma chamada do método `asYouDesire`. Inicialmente, informa-se a classe e o método alvo (passo 1). Assim, o Cabelenium cria um objeto, com essas informações (passo 2). Para a execução do método alvo, pode ser necessária a inicialização de atributos do objeto da classe (passo 3). Se sim, existem duas formas. Uma em que é informado individualmente o nome do atributo e seu valor (passo 4). A outra é uma otimização em que é informado um mapa onde cada entrada representa o nome do atributo e seu valor (passo 5). Por fim, se não houver mais valores a serem passados, executa-se o método *run*. Então, são feitas verificações se os atributos existem na classe e, também, se os tipos dos valores são correspondentes aos dos atributos. Caso satisfeitas as condições, executa-se o método alvo com os valores atribuídos aos atributos (passo 6).

Figura 1. Fluxograma de execução de um método utilizando o `asYouDesire`.



O Código 14 ilustra um caso de teste que envolve a venda de um produto. No exemplo, o método `asYouDesire` (linhas 5-10) realiza a invocação do método `novo` para criação de um registro de produto. Para isso, foi passada a definição literal da classe produto (obtida do Código 7, Seção 3.3) e o nome do método (linha 5). Ainda, para ao método `addParam`, foram passados os nomes dos atributos e seus valores utilizados no `novo` (linhas 6-9). Note que não foi necessário passar o valor do atributo `unidade` (linha 7 do Código 7, Seção 3.3), porque o seu valor é definido como padrão, definido com a anotação `@Default`. No entanto, os demais atributos devem ser instanciados para invocação do método sem erros. Em seguida, é feito o registro de cadastro de venda (linhas 12-17) utilizando o código do produto criado (linha 14), assim relacionando o produto com o cliente.

Código 14. Exemplo teste de venda de um produto utilizando `asYouDesire`

```
1  @Test
2  @Clean({ "Produto:CELULAR-08" })
3  public void fluxoSimples() {
4          LOGGER.trace("Cria produto");
5          this.asYouDesire( ProdutoTestCase.class, "novo")
6              .addParam("cnpj", 54059036000170L)
7              .addParam("codigo", "CELULAR-08")
8              .addParam("nome", "Celular Z8 zony")
9              .addParam("tipo", "Inteiro")
10             .run();
11
12         this.asYouDesire("novo")
13             .addParam("cpf", 33344422270L)
14             .addParam("codigo", "CELULAR-08")
15             .addParam("nome", "Luiz Fraga")
16             .addParam("valor", 4550)
17             .run();
18 }
```

Com essas informações, é possível compreender o funcionamento básico da execução do método *asYouDesire*. Basicamente, se estiver na classe do método alvo, não há necessidade de passar a definição da classe na chamada do método (Código 14, linha 12).

Caso tenha atributos que necessitam de atribuição de valores, utilizar o *addParam* (por exemplo, no Código 14, linhas 6 a 9). Por fim, a invocação do método *run* dispara a execução do método.

Conforme a Tabela 2, os métodos mais utilizados com o `asYouDesire` para execução de um método alvo são: `addParam` e `run`, são os métodos básicos para qualquer teste e são simples de utilizá-las. O `addParams` é uma otimização do *addParam* que, ao invés de passar os dados por cascata (por exemplo, no Código 14 nas linhas 6 a 9), usa-se uma coleção de pares (Código 15).

Tabela 2. Funções utilizados com `asYouDesire` e suas descrições.

Método	Descrição
run ()	Executa o método desejado para o teste, a partir do nome passado para o *asYouDesire*.
addParam (*attributeName*, *value*)	Atribui o valor *value* ao atributo com o nome *attributeName*.
addParams (*map*)	Invoca o método addParam supracitado para cada entrada no mapa.
single (*jsonfileName*)	Atribui o valor ao atributo com o mesmo nome, para uma única entrada de um arranjo, obtido do arquivo JSON.
multiple (*jsonfileName*)	Atribui os valores aos seus atributos com os mesmos nomes, para cada entrada de um arranjo, obtido do arquivo JSON.

O Código 15 ilustra um fragmento que exemplifica como utilizar o método `addParams`. Assim, se instancia um objeto (chave/valor) e adiciona-se valores (linhas 1-5). Por fim, passa como parâmetro um objeto para o método `novo` da classe `ProdutoTestCase` (linhas 7-9). Pode-se percebe que, executa-se a mesma operação do Código 14, nas linhas 5 a 10.

Código 15. Fragmento de código, utilizando o `addParams`

```
1 Map<String,Object> map = new HashMap<>();
2 map.put("cnpj" , 54059036000170L);
3 map.put("codigo" , "CELULAR-08");
4 map.put("nome" , "Celular Z8 zony");
5 map.put("tipo" , "Inteiro");
6
7 this.asYouDesire(ProdutoTestCase.class, "novo")
8     .addParams(map)
9     .run();
```

Os métodos `single` e `multiple` foram criados para facilitar a passagem de dados por parâmetro. Assim, basta criar um arquivo JSON contendo os dados com nome do atributo e seu valor; se for utilizar o `single` (por exemplo, na linha 11 da Seção 3.3, Código 5) ou um arranjo contendo objetos (conforme o Código 17).

O Código 16 ilustra um fragmento que utiliza como base o Código 5 (Seção 3.3), porém, cria-se dois registros de televisão, conforme o Código 17. Ou seja, para cada elemento contido no arranjo do JSON, é executado o método alvo utilizando esses dados.

Código 16. Fragmento de código, utilizando o `multiple`

```
1 this.asYouDesire(ProdutoTestCase.class, "novo" )
2     .multiple("TVs.json")
3     .run();
```

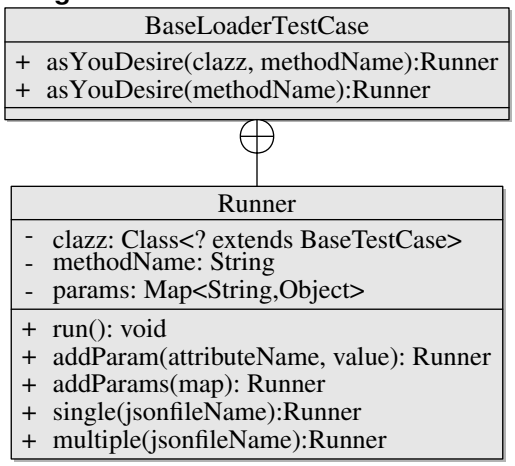
Na linha 1 do Código 16, é invocado o método `novo` da classe `ProdutoTestCase` passando o arquivo JSON (linha 2), pelo método `multiple`.

Código 17. JSON com arranjo de Produtos

```
[
    {
        "codigo" : "TV-01",
        "nome" : "Televisão Plasma",
        "cnpj" : 39789102000173,
        "tipo" : "Inteiro"
    },
    {
        "codigo" : "TV-02",
        "nome" : "Televisão LCD 4K",
        "cnpj" : 39789102000173,
        "tipo" : "Inteiro"
    }
]
```

O diagrama de classe da Figura 2 ilustra que o método *asYouDesire* é pertencente a classe *BaseLoaderTestCase*. Pode-se perceber que o método possui uma sobrecarga de métodos, isto é, a sua chamada pode ser feita de duas formas: (i) passando a referência literal da classe e o nome do método ou (ii) apenas o nome do método, caso a invocação, seja realizada na própria classe.

Figura 2. Diagrama de classe do *Runner* e do `asYouDesire`.



O *Runner* em sua definição é uma classe interna da *BaseLoaderTestCase* cujos atributos recebem os dados essenciais de um método. Os atributos são: *clazz*, para receber a definição da classe; *methodName*, para receber o nome do método a ser executado; e *params*, para receber os dados de nome dos atributos (chave) e os seus valores (valor). Os métodos são justamente aqueles utilizados para execução de um método com o *asYouDesire*, já previamente descritos na Tabela 2. Enfim, a utilidade do objeto da classe *Runner* é armazenar os dados para invocação de um método.

Basicamente, os métodos `addParam` e `addParams` retornam o *this* de forma a permitir o encadeamento de chamadas. Isso significa que o objeto do tipo *Runner* é alterado adicionando mais itens ao atributo *params*, quando houver novamente o uso desses métodos, como no Código 14 (linhas 6-9 e 13-16). Ou seja, a classe *Runner* é o meio pelo qual pode-se configurar execuções de qualquer objeto de uma classe. Além disso, conforme a Figura 1, sua criação é realizada logo após a chamada do método *asYouDesire* (passo 2).

Desse modo, percebe-se que o foco do reúso do método *asYouDesire* segue o conceito do *don't repeat yourself* (*dry*), ou seja, evitar repetição de blocos de códigos

(HUNT; THOMAS, 2000). Assim, cada método de uma classe no projeto de teste é chamada apenas pelo *asYouDesire*, evitando assim a repetição da chamada de construtores para cada objeto. Além disso, isso simplifica a criação de novas classes porque não precisa criar vários construtores com parâmetros diferentes; não existindo as suas sobrecargas. Por fim, isso melhora o aspecto de variação de códigos sem padrão, o que o Selenium não trata.

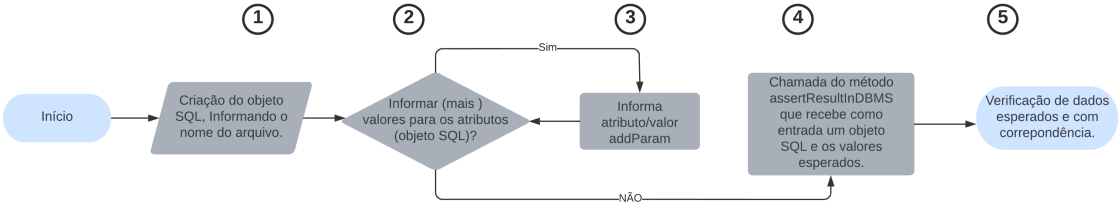
Mais importante, o método `asYouDesire` foi projetado utilizando os princípios do padrão de projeto *Builder* adaptado com programação reflexiva. O padrão *Builder* separa a construção de um objeto (encapsulamento) e, desse modo, permite que seja construído por etapas (GAMMA et al., 2000; FREEMAN et al., 2009). A técnica de programação reflexiva consiste em utilizar uma metaclasse para fazer reflexão sobre si mesmo e modificar informações sobre a classe, suas instâncias, atributos e métodos de uma classe em tempo de execução (GONÇALVES, 2012; MASUHARA et al., 1992). Portanto, pode-se perceber o padrão de projeto, conforme o fluxograma da Figura 1, na utilização do método *asYouDesire*, pois os passos para invocação de um método alvo se resumem em uma preparação do objeto de uma classe. Por fim, a reflexão, pode ser percebida com o uso do método `run` pois, em sua execução, cria-se usando reflexão um objeto da classe *className* e inicializa cada atributo em *params*.

4.3. Consistência de dados

O Cabelenium provê nativamente o recurso de verificação de banco de dados. O objetivo é garantir a consistência e integridade dos dados salvos no banco de dados (Seção 3.2). Para isso, utiliza-se o método denominado *assertResultInDBMS* que recebe como parâmetro um objeto SQL e os valores esperados como resultado da consulta (Seção 3.5).

O fluxograma da Figura 3 ilustra a sequência de passos que ocorre quando se faz a chamada do método `assertResultInDBMS`. Inicialmente, ocorre a criação do objeto SQL que possui como valor de entrada o nome do arquivo que contém a consulta (passo 1). Assim, se há necessidade de passar valores para consulta (passo 2), utiliza-se o método `addParam`, passando o nome do parâmetro na consulta e o seu respectivo valor (passo 3). Se não, faça-se a chamada do método recebendo como entrada um objeto SQL e os valores esperados (passo 4). Por fim, é realizada a verificação dos dados da consulta com os dados passados como parâmetro do método `assertResultInDBMS` (passo 5).

Figura 3. Fluxograma de execução do método `assertResultInDBMS`.



O Código 18 ilustra um caso de teste que envolve a verificação de dados de um registro de um produto. Assim, o método recebe como parâmetro um objeto SQL (linha 1) instanciado com nome do arquivo da consulta e utilizou o `addParam` para atribuir valores

aos parâmetros da respectiva instrução SQL (linha 2). Em seguida, foram passados os valores a serem comparados com os persistidos no banco de dados (linha 3). O exemplo utiliza como base a criação do produto como ocorre no Código 14 (Seção 4.2).

Código 18. Fragmento de código, consistência de dados de uma quadrutupla.

```
1 this.assertResultInDBMS(new SQL( "produto001.sql" )
2     .addParam( "codigo" , "CELULAR-08"),
3     "Celular Z8 zony", 54059036000170L, "Inteiro", "un");
```

Na linha 1 do Código 18, é passado para o objeto SQL com o nome da consulta da instrução. Assim, o Código 19 ilustra a consulta que recebe como parâmetro do ":codigo" o valor "CELULAR-08" e, por fim, retorna como resultado uma quádrupla da tabela "Produto".

Código 19. Consulta SQL, arquivo `produto001.sql`

```
1 select NOME, CNPJ, TIPO, UNIDADEPRIMARIA
2     from PRODUTO
3     where CODIGO = ':codigo';
```

Com essas informações, é possível compreender a facilidade de se fazer uma verificação da consistência de dados. Basicamente, a consulta não precisa ser alterada, basta as informações dos parâmetros e as linhas de consistências serem alteradas de acordo com cada instrução.

Conforme descrito na Tabela 3, o método `assertResultInDBMS` têm variações de passagem de dados por parâmetro: (i) verificando uma única linha, resultado da consulta (Código 18), (ii) verificando uma coleção de linhas (Código 20) ou (iii) verificando linhas de resultados organizados em arranjos (Código 21).

Tabela 3. Funções utilizados com `assertResultInDBMS` e suas descrições.

Método	Descrição
assertResultInDBMS (Sql, *values*...)	Consistência de dados de uma única linha.
assertResultInDBMS (Sql, *collection*)	Consistência de dados com valores contidos na coleção de arranjos (representando uma ou mais linhas de resultado).
assertResultInDBMS (Sql, *values*[]...)	Consistência de dados com valores organizados em arranjos (representando uma ou mais linhas de resultado).

O Código 20 ilustra um fragmento que exemplifica a verificação de dados, utilizando uma coleção. Assim, são adicionadas várias linhas de consistências iguais em um objeto (chave/valor) (linhas 2-4) e uma linha diferente (linha 5). Dessa forma, o objeto é utilizado como os dados esperados (linhas 7-8). É importante mencionar que essa variação é tipicamente usada quando se têm muitas linhas iguais.

Código 20. Fragmento de código, consistência de dados de uma coleção.

```
1 Set<Object[]> expected = new HashSet<>();
2 for (int i = 0; i < 5; i++) {
3     expected.add(new Object[] { "nomeComum", 1114445558887L} );
4 }
5 expected.add(new Object[] { "nomeDiferente", 1114445558887L} );
6
7 this.assertResultInDBMS(new SQL("pessoaExemplo.sql")
8     .addParam( "idColecao" , "MG00"), expected);
```

O Código 21 ilustra um fragmento que exemplifica a verificação de dados, utilizando arranjos de objetos. Assim, utilizando a mesma instrução SQL do Código 20

(linha 7), realiza-se uma verificação de consistência de dados (linhas 1-5) de duas linhas de verificação de dados diferentes (linhas 3-4). É importante mencionar que essa variação é tipicamente usada, quando se têm poucas linhas ou todas são diferentes.

Código 21. Fragmento de código, consistência de dados de linhas.

```
1 this.assertResultInDBMS(new SQL("pessoaExemplo.sql")
2     .addParam("idColecao", "MG-01"),
3     new Object[] { "Arret",  1114445558887L},
4     new Object[] { "Werdna", 1144050466444L}
5 );
```

Conforme previamente descrito, o método de verificação de consistência de dados (`assertResultInDBMS`) necessita do objeto da classe SQL. Todavia, em seu funcionamento interno, utiliza-se dos métodos da classe *Database* para buscar e, então, comparar as linhas de resultados do banco de dados.

4.3.1. Classe SQL

A classe SQL descreve uma instrução e seus parâmetros a serem executados. Conforme a Tabela 4, o método `addParam` atribui valores nos parâmetros contidos na instrução SQL.

Tabela 4. Funções utilizados pelo objeto `SQL` e suas descrições.

Método	Descrição
addParam (*key*, *value*)	Atribui o valor *value* ao parâmetro com o nome *key*.
done ()	Retorna uma instrução SQL pronta para execução. Isto é, substitui os parâmetros contidos na instrução com os valores obtidos pelo *addParam*.

O método `done` é responsável por realizar as substituições de parâmetros por valores (passados pelo `addParam`) para de fato retornar a instrução a ser executada. Por exemplo, conforme o Código 22 ilustra, utilizando o Código 9 (Seção 3.5), o resultado da substituição do parâmetro "codigo" por "TV".

Código 22. Exemplo de resultado de retorno do `done`.

```
1 select QTDE_DISPONIVEL, QTDE_RESERVADA from ESTOQUE e inner join PRODUTO p on
      e.ID_PRODUTO = p.ID  where  p.CODIGO = 'TV'
```

Dessa forma, uma única instrução SQL pode ser utilizada em outras consultas apenas substituindo valores de parâmetros. Ou seja, evita-se redundância na criação das mesmas consultas com valores diferentes.

4.3.2. Classe Database

A classe *Database* é responsável por realizar a execução de instruções SQL, como inserções (*insert*), atualizações (*update*), exclusões (*delete*) e as consultas (*select*) que, basicamente, são as mais comuns.

A Tabela 5 contém as descrições dos métodos da classe *Database*. É importante notar que o *runSqlStatement* tem uma sobrecarga passando uma lista de objetos SQL para

o caso de executar várias instruções SQL dentro de um contexto transacional. Ou seja, são executadas todas as instruções ou nenhuma (CHEN; BETAPUDI, 1994).

Assim, uma das utilizações desse método é feita pelo `@Clean` que executa várias instruções SQLs de forma a prover a limpeza dos dados. Todavia, remove-se todos os registros ou nenhuma limpeza é realizada (caso ocorra algum erro).

Tabela 5. Métodos da classe `Database` e suas descrições.

Método	Descrição
runSqlStatement (*sql*)	Executa a instrução contida no objeto *sql* (em caso de *update*, *insert* ou *delete*).
runSqlStatement (*List<SQL> sqls*)	Executa as instruções contida na lista de objetos *sql* (em caso de *update*, *insert* ou *delete*).
sqlMultiple (*sql*)	Retorna uma lista de linhas de resultado da consulta contida no objeto *sql*.
sqlSingle(*sql*)	Retorna uma linha de resultado da consulta, contida no objeto *sql*.

4.3.2.1. Exemplos de consultas

O Código 23 ilustra como a classe *Database* é utilizada com o método `sqlSingle` que retorna uma linha do resultado da consulta. Assim, ocorre uma atribuição à variável `nomeProduto`, da qual se recebe o valor da primeira posição do arranjo retornado (linha 1). É importante mencionar que essa variação é usada em consultas que retornam uma única linha e também utilizam apenas um valor, então, faz-se uso da indexação do arranjo.

Código 23. Código utilizando o `sqlSingle`

```
1 String nomeProduto = Database.sqlSingle(new
      SQL("nomeProduto.sql").addParam("idExterno", identificadorExternoProduto))[0];
```

Na linha 1 do Código 23, é invocado o método `sqlSingle` da classe `Database` passando um objeto SQL com o nome do arquivo da consulta do Código 24. Assim, ilustra-se uma instrução de consulta que busca o campo `nome` da tabela `Produto`, usando o identificador externo para a operação (linha 1).

Código 24. Exemplo da instrução "nomeProduto.sql"

```
1 select p.NOME from PRODUTO p where p.IDEXTERNO = ':idExterno';
```

O Código 25 ilustra como a classe *Database* é utilizada com o método `sqlMultiple`. Assim, nesse exemplo, ocorre a atribuição da variável `nomesProdutos` com uma lista de arranjos, ou seja, várias linhas do resultado da consulta (linhas 1-2).

Código 25. Código utilizando o `sqlMultiple`

```
1 List<Object[]> nomesProdutos = Database.sqlMultiple(new SQL("nomesProdutos.sql")
2     .addParam("idLojaExterno", "UFLA-BIB"));
```

Na linha 1 do Código 25, é invocado o método `sqlMultiple` da classe `Database` passando um objeto SQL com o nome do arquivo da consulta do Código 26. Assim, ilustra-se uma instrução de consulta que busca o campo nome da tabela `Loja`, usando-se o identificador externo para a operação (linha 4).

Código 26. Exemplo da instrução "nomesProdutos.sql"

```
1 select p.NOME
```

```
2       from Loja l
3       inner join PRODUTO p on p.ID = l.IDPRODUTO
4       where l.IDEXTERNO = ':idLojaExterno';
```

4.3.2.2. Exemplo de outras instruções

A classe *Database* é também utilizada por métodos do *framework* para executar a instrução SQL de limpeza de vestígios de dados, como ilustrado no Código 27.

Código 27. Código utilizando o `runSqlStatement`

```
1 Database.runSqlStatement(new SQL("produto-clean.sql"));
```

O Código 27 ilustra a invocação do método `runSqlStatement` da classe `Database`, passando um objeto SQL instanciado com o arquivo "produto-clean.sql". Assim, há alguns recursos que podem ser utilizados em uma instrução SQL, por exemplo, o Código 28 ilustra que o resultado do *select* é atribuído a uma variável (linhas 2-4). Dessa forma, é posteriormente utilizada para deletar os dados de `catalogoDadosAdc` (linha 11). Essa abordagem é uma forma de exclusão que visa o problema de exclusão múltipla, o que permite evitar a desabilitação de *foreign key*. Então, como visto, pode-se guardar um valor obtido através de uma consulta em uma variável e utilizá-lo posteriormente. Por fim, as demais linhas (6 a 9 e 13) são também executadas instruções de *delete*, seguindo a ordem de prioridade para não afetar as dependências de cada registro.

Código 28. Exemplo de *script utilizando* var

```
1  --Apagando Dados Adicionais
2  var idDadoAdicional = select id from dadoAdicional where idProduto IN (
3      select p.ID from PRODUTO p where p.IDEXTERNO = ':idExterno'
4  );
5
6  delete from dadoAdicional where and idProduto IN (
7      select p.ID from PRODUTO p
8          where p.IDEXTERNO = ':idExterno'
9  );
10
11 delete from catalogoDadosAdc where id IN (:idDadoAdicional);
12
13 delete from PRODUTO where idExterno = ':idExterno';
```

Com as informações supracitadas, pode-se compreender que o método `runSqlStatement` realiza execução de várias instruções e também efetua a operação de atribuição a uma variável. Assim, pode-se apagar outras tabelas com a informação de um registro que fora anteriormente apagado.

4.4. Autocontido

O Cabelenium provê o recurso de remoção de dados criados durante a execução dos testes. Dessa forma, a partir de todos os dados criados por um "teste x", o objetivo do `@Clean` é remover antes da execução todos os dados que vão ser criados, garantindo que não existam esses dados. Também, após a execução, o objetivo é apagar todos os dados criados para evitar efeitos colaterais que podem afetar outros testes (Seção 3.6). Por isso, deve-se utilizar a anotação `@Clean` em todos os casos de testes.

O funcionamento do @Clean consiste em, por exemplo, antes de ser executado o método do Código 14 (Seção 4.2), é feita a limpeza dos registros que consiste no produto identificado por "CELULAR-08" (linha 2). Também, após a execução do método, é novamente realizado o processo do @Clean. Basicamente, busca-se o arquivo "produto-clean.sql", devido "Produto" estar contido na assinatura da anotação, passando para o parâmetro "codigo" o identificador "CELULAR-08". Ou seja, o Código 29 é o conteúdo do Código 11 (Seção 3.6), após a substituição supramencionado.

Código 29. Exemplo de instrução modificada no @Clean

```
1 --Apagando Estoque
2 delete from ESTOQUE where IDATIVO IN (
3     select p.ID from PRODUTO p where p.CODIGO = 'CELULAR-08'
4 );
5
6 --Apagando o produto em si
7 delete from PRODUTO where codigo = 'CELULAR-08';
```

A Tabela 6 contém a descrição das anotações relacionadas ao processo de testes autocontidos. Por um lado, a anotação @NoCleanBefore não executa o @Clean antes da execução de um teste. É utilizada geralmente quando os registros já estão criados e necessita-se de uma execução parcial. Por outro lado, @NoCleanAfter não executa o @Clean depois da execução de um teste e usado geralmente para situações em que se precisam verificar dados criados na execução.

Tabela 6. Tabela de anotações

Método	Descrição
Clean	Anotação que garante que os testes sejam autocontidos.
NoCleanBefore	Anotação que garante que não haverá limpeza dos dados antes da execução do teste.
NoCleanAfter	Anotação que garante que não haverá limpeza dos dados ao término de execução do teste.

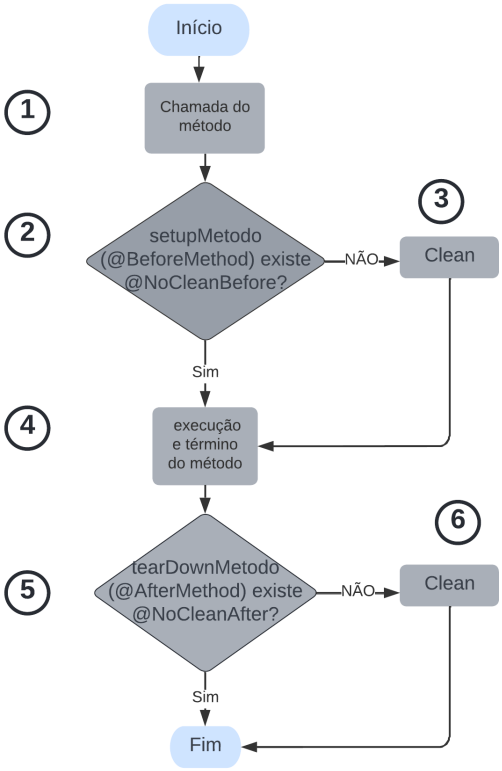
O fluxograma da Figura 4 ilustra o fluxo de execução de um teste. 1) inicialmente, faça-se a chamada de um método que possua a anotação do *@Clean*. 2) chamada do método setupMetodo customizado, mas provido pelo próprio JUnit, da qual se verifica se há presença do @NoCleanBefore. 3) caso não se tenha a anotação, executa-se o *clean*. 4) executam-se as instruções do teste. 5) Após a execução do método e pelo próprio recurso do JUnit, é feita a chamada do tearDownMetodo. E caso não haja anotação @NoCleanAfter, 6) é novamente executado o *clean*.

Mais importante, para cada instrução contida na anotação, é realizada a ordenação para se executar o *clean*. Por exemplo, houve a criação de dois registros: "departamento" e "funcionario". Como os registros de "departamento" estão vinculados a "funcionario" é importante que, os registros de "funcionario" sejam excluídos antes da outra tabela. Com isso, implica-se na necessidade de se ordenar as instruções, assim tendo ordem nas exclusões.

4.5. *Logs do sistema*

O Cabelenium provê nativamente o recurso de exibição de *logs* de execução. O objetivo é garantir a rastreabilidade das interações com sistema web e operações de métodos (Seção 3.2). Ou seja, todas as funções do Cabelenium já estão encarregadas de gerar informações de suas próprias operações, como uma auditoria.

Figura 4. Fluxograma de execução do `@Clean`.



A Tabela 7 contém a descrição dos tipos de *loggers* relevantes para programação dos testes. O `trace` e o `debug` são mais utilizados nos casos de testes. Já `info` e o `warn` são para programação do próprio Cabelenium. É importante mencionar que o mecanismo de *logs* é provido pela biblioteca Apache Log4j[5], utilizada para exibir as informações de execuções do Cabelenium.

Tabela 7. Tabela de *loggers*

Método	Descrição
trace	Utilizado para comentários sem passagem de dados de vaiáveis ou objetos.
debug	Utilizado para comentários com passagem de dados de vaiáveis ou objetos.
info	Utilizado para exibir informações do sistema.
warn	Utilizado para exibir informações de que as operações do sistema não estão sendo executadas ou não estão corretas.
error	Utilizado para exibir erros dos métodos do próprio *framework*.

É importante mencionar que há um nível de *log*, graduado segundo a sua necessidade, ordenado respectivamente a partir do *trace*, *debug*, *info*, *warn*, e terminando com *error*. Esse critério é estabelecido pela classe `LoggerFactory` (GUPTA, 2003), da qual os *logs* são oriundos.

O Código 30 ilustra a exibição do *log* de uma execução de um teste executado. Assim, foram exibidas informações do sistema (linhas 1-7), como pode ser observado mos-

[5]<https://logging.apache.org/log4j>

tra a versão do *driver* do *Chrome* (linha 3), a versão do Java (linha 5) e foi executando com o navegador *Chrome* no sistema operacional *Linux* (linha 6). Ao executar de fato o método gerou as informações do nome método (linhas 11-13) e dos registros de limpeza (linhas 14-19). Mais importante, mostrou as interações com a interface web (linhas 22-32). Por exemplo, acessou o menu (linha 22), fez assertivas (linha 23, 31 e 32), clicou em botões (linha 24, 28 e 30), selecionou uma opção dentro de um elemento *radio* (linha 25) e fez-se escritas nos campos (linhas 26, 27 e 29). Por fim, novamente a execução do `@Clean` (linhas 34-39) e mensagens de conclusão (linhas 40-42).

Código 30. Exemplo de log de um test

```
1  [RemoteTestNG] detected TestNG version 7.1.0
2  INFO  BaseTestCase URL[] DB[jdbc:mysql://XXX.XX.XXX.XXX:3306/gtiwms?autoReconnect=true&useSSL=false] USER[user]
3  Starting ChromeDriver 98.0.4758.80 () on port
4  ChromeDriver was started successfully.
5  INFO  BaseTestCase Java Version 1.8.0_312
6  INFO  BaseTestCase Browser chrome version 98.0.4758.80 running on LINUX
7  INFO  BaseTestCase PLUG Version 9.16.5
8  TRACE PlugWebUtilitiesTestCase write :: id=j_username value=user@site
9  TRACE PlugWebUtilitiesTestCase write :: id=j_password value=admin
10 TRACE PlugWebUtilitiesTestCase click :: id=btnLogin
11 DEBUG BaseTestCase ****************************************************
12 INFO  BaseTestCase Running ProdutoTestCase:fluxoSimples
13 DEBUG BaseTestCase ****************************************************
14 TRACE SQL Creating SQL from produto-clean.sql
15 DEBUG Database Running produto-clean.sql with { idExterno=TUPRD001}
16 DEBUG Database A variável idEpcApagar não foi inicializada durante a execução do script SQL
17 DEBUG Database OK(0) [delete from ESTOQUE where IDPRODUTO IN ( select a.ID from PRODUTO a where a.IDEXTERNO =
       'TUPRD001')]
18 DEBUG Database OK(0) [delete from PRODUTO where idExterno = 'TUPRD001']
19 DEBUG Database COMMITTED
20 DEBUG BaseTestCase Configuração do tenant atualizada com sucesso!
21
22 TRACE PlugWebUtilitiesTestCase opening menu :: module=modulemenu_dadosBasicos menu=menu_produtos empresa=null
23 TRACE PlugWebUtilitiesTestCase assert :: 'GTI PLUG: Produtos' contains 'Produtos'
24 TRACE PlugWebUtilitiesTestCase click :: id=btNovo
25 TRACE PlugWebUtilitiesTestCase selectMany :: id=checkTipoProduto values=[Fracionado Variável]
26 TRACE PlugWebUtilitiesTestCase write :: id=inputNome value=Produto FV PRD
27 TRACE PlugWebUtilitiesTestCase write :: id=inputIdentExterna value=TUPRD001
28 TRACE PlugWebUtilitiesTestCase click :: id=btSel_fornecedor
29 TRACE PlugWebUtilitiesTestCase write :: id=inputCnpj value=39789102000173
30 TRACE PlugWebUtilitiesTestCase click :: id=btSalvar
31 TRACE PlugWebUtilitiesTestCase assert :: id=formCenter:msgs_formCenter text=Registro incluído com sucesso.
32 TRACE PlugWebUtilitiesTestCase assert :: 'Registro incluído com sucesso.' contains 'Registro incluído com
       sucesso.'
33
34 TRACE SQL Creating SQL from produto-clean.sql
35 DEBUG Database Running produto-clean.sql with { idExterno=TUPRD001}
36 DEBUG Database A variável idEpcApagar não foi inicializada durante a execução do script SQL
37 DEBUG Database OK(6) [delete from ESTOQUE where IDPRODUTO IN ( select a.ID from PRODUTO a where a.IDEXTERNO =
       'TUPRD001')]
38 DEBUG Database OK(1) [delete from PRODUTO where idExterno = 'TUPRD001']
39 DEBUG Database COMMITTED
40 DEBUG BaseTestCase Configuração do tenant atualizada com sucesso!
41 INFO  BaseTestCase Method ProdutoTestCase:fluxoSimples took 23.278 s.
42 INFO  BaseTestCase ***O DRIVER DO CHROME FOI ENCERRADO COM SUCESSO***
```

Mais importante, o Cabelenium possui *logs* implementados internamente, exibidos por meio do `LOGGER`, como visto no exemplo (Código 30). Ou seja, não há necessidade de inserir mais `LOGGER` em um teste, para que as informações textuais sejam exibidas.

5. Trabalhos Relacionados

Gojare, Joshi e Gaigawae (2015) apresentaram uma avaliação do Selenium a partir de uma estrutura de teste própria, criada em conjunto com TestNG. O objetivo é extrair resultados das análises das funcionalidades de cada ferramenta a partir do teste de regressão. Dessa forma, conclui-se que o Selenium é um *framework* adequado para aquilo que se propõe, isto é, garantir que um teste automatizado possa ser verificado a partir da sua geração de relatório e reduzir o tempo de programação de *scripts*. Portanto, pode-se compreender que foi adequada a escolha do Selenium como base do Cabelenium.

Kumar e Saxena (2015) apresentaram uma implementação que simula usuários acessando o *e-mail*, usando o Selenium e o TestNG. O objetivo é explicar como desenvolver e usar o Selenium em Java (para testar o acesso na aplicação web e aos dados de usuários em um arquivo *xls*) com o TestNG (para gerar relatórios de execução). Dessa forma, concluiu-se que o *framework* gerou um bom resultado, enquanto se utiliza os dados advindos do arquivo. Portanto, pode-se reforçar que foi adequada a escolha do Selenium como base do Cabelenium, pois, através de seus recursos, obtém-se variadas possibilidades de interação com ambientes web.

Singh e Sharma (2015) apresentaram um comparativo entre as ferramentas de testes Selenium e Sahi Pro. O objetivo é auxiliar testadores ou analistas de qualidade na melhor escolha conforme os recursos e requisitos de cada sistema. O método consiste em realizar comparações entre as ferramentas, considerando o tempo de execução, a eficiência de gravação e reprodução, o navegador e compatibilidade de plataforma, os relatórios de resultados, a facilidade de aprendizado e o custo. Dessa forma, concluiu-se que ambas as ferramentas possuem bons resultados, então a decisão está em função das especificações providas pelo ambiente, em que uma das ferramentas estará em execução. Portanto, embora este documento utilize Selenium, uma potencial escolha poderia ser também o Sahi Pro.

Socolowski, Alarcon e Antonio (2013) apresentaram o *framework* FlexTest que utiliza testes do Selenium como uma das tarefas de execução. O objetivo é, por meio do *framework* desenvolvido, executar testes funcionais para aplicações web. O funcionamento do FlexTest é, a partir de um arquivo *csv* que contém os comandos de execução, executar um *script* de teste gravado pelo Selenium e, também realizar listas de verificações de consistência de dados. Dessa forma, conclui-se que o resultado da aplicação do FlexTest foi satisfatório, mas tem que considerar o ambiente em que será implementado e as suas necessidades. Este documento também demonstra que o Selenium suporta ser integrado com outras ferramentas, além de ser usado como base do projeto do FlexTest, tal como no Cabelenium.

Wang e Xu (2009) apresentaram uma estrutura de teste que combina o *framework* Selenium com o FitNesse. O objetivo é obter sucesso com a integração dos *frameworks* de forma que o FitNesse supra o Selenium em aspectos em que ele não se propõe a realizar. Dessa forma, conclui-se que o resultado da integração foi satisfatório atingindo o objetivo proposto. Este documento também descreve e apresenta um *framework* que realizou melhorias em aspectos em que o Selenium não se propõe a realizar.

Petrova-Antonova, Ilieva e Manova (2015) apresentaram o *framework* TASSA, um *framework* de orquestrações de serviços web para testes funcionais e não funcionais. O objetivo é apresentar as decisões de projeto e o seu funcionamento. Assim, o seu fluxo de execução consiste em três fases: i) a partir de um arquivo BPEL – que contém os modelos de criação dos testes e as transformações de processos de negócio – servir de entrada para o início da execução; ii) gerar os testes conforme o modelo assumido; e iii) executar os testes e exibir os seus resultados. Foi realizado um estudo de caso e, por meio das análises, o TASSA tem capacidade de realizar o que foi proposto, além de ter bom desempenho. Portanto, é possível compreender que o esquema de funcionamento do TASSA pode ser uma possibilidade de uso para o Cabelenium, em específico, para ter suporte automati-

zado dos bancos de dados relacionais, de modo que a própria implementação interprete automaticamente os modelos a partir das instruções.

6. Conclusão

Testes automatizados podem garantir que um sistema esteja funcionando conforme o seu projeto, trazendo segurança em seu desenvolvimento (BERNARDO; KON, 2008). Desse modo, o Selenium é um *framework* que oferece diversas opções para que os testes automatizados sejam sucedidos. No entanto, ao usá-lo pode ocorrer problemas tais como códigos verbosos e com baixa manutenibilidade, o que dificulta o reúso (TERRA, 2020). Outra questão, para um sistema de teste, apenas verificar se o fluxo de operações ou funcionalidades estão operando como projetado pode não bastar, logo necessita-se de verificações dos dados. Ou seja, por exemplo, verificar se estão sendo salvos devidamente no banco de dados ou gerar alguma informação que permita fazer uma auditoria. Então, embora o Selenium não se proponha a suprir, existem necessidades – como as supracitadas – que são relevantes no desenvolvimento de testes funcionais.

Diante disso, este artigo apresentou o Cabelenium, um *framework* de testes funcionais automatizados desenvolvido a partir do Selenium. A sua contribuição consiste em adaptar-se nos aspectos das quais se têm dificuldades (com Selenium) na elaboração de testes e adicionando novas funcionalidades. Assim, as melhorias residem nas propriedades pilares (Seção 3.2). As características são a i) portabilidade, que consiste em se o *framework* for trocado de ambiente, mantenha-se os mesmos métodos, alterando apenas a programação que o integra com a tecnologia do *front-end* (Seção 4.1); ii) o reúso que, por meio do método *asYouDesire*, pode-se utilizar qualquer método das classes do projeto de teste, bastando alterar os parâmetros do método (Seção 4.2); iii) a consistência de dados que, por meio do método *assertResultInDBMS*, realizam-se verificações de valores esperados com os do banco de dados (Seção 4.3); iv) testes autocontidos que consistem em apagar apenas aqueles dados que foram criados durante a execução de um teste para não se ter efeitos colaterais provenientes desses dados na execução de um outro teste (Seção 4.4); e, por fim, v) a rastreabilidade, sendo a funcionalidade de registro de *log* que é associada a rastreabilidade que significa registrar qualquer ação (por exemplo, assertivas, consultas `SQL`, cliques e digitação em um campo) durante uma execução de um teste, então, auxiliando em uma auditoria (Seção 4.5). Além disso, o Cabelenium possui um menor tempo de desenvolvimento em relação ao Selenium (TRABOUSSY, 2021) devido ao uso de seus componentes que evitam a repetição de códigos e, consequentemente, promovem eficiência.

Trabalhos futuros seriam: (1) realizar um estudo de caso de portabilidade do *framework* para outro ambiente de trabalho, por exemplo, em outra empresa diferente da qual o Cabelenium foi atualmente projetado (TERRA, 2020). Dessa forma, demonstrando, através desse estudo, a propriedade de portabilidade; (2) adaptá-lo para integração com outras tecnologias de *front-end*; (3) efetuar uma troca para outro banco de dados relacional, como PostgreSQL e, verificar seu funcionamento com consistência no banco de dados e do `Clean`. Desse modo, estendendo a portabilidade do *framework* e analisando as mudanças necessárias.

Referências

ANGMO, R.; SHARMA, M. Selenium tool: A web based automation testing framework. **International Journal of Emerging Technologies in Computational and Applied Sciences**, v. 4, n. 13, p. 351–355, 2014.

ATEŞOĞULLARI, D.; MISHRA, A. Automation testing tools: A comparative view. **International Journal on Information Technologies & Security**, v. 12, n. 4, p. 1–14, 2020.

BERNARDO, P. C. **Padrões de testes automatizados**. Dissertação (Mestrado) — Universidade de São Paulo, Programa de Pós-Graduação em Ciência da Computação, 2011. p. 221.

BERNARDO, P. C.; KON, F. A importância dos testes automatizados. In: . [S.l.]: Engenharia de Software Magazine, 1(3), 2008. p. 54–57.

CHEN, I.-R.; BETAPUDI, R. A petri net model for the performance analysis of transaction database systems with continuous deadlock detection. In: **4th ACM Symposium on Applied Computing (SAC)**. [S.l.: s.n.], 1994. p. 539–544.

CORREIA, S. A.; SILVA, A. R. d. Técnicas para construção de testes funcionais automáticos. In: **5th International Conference on the Quality of Information and Communications Technology (QUATIC)**. [S.l.: s.n.], 2004. p. 111–117.

FREEMAN, E. et al. **Use a Cabeça - Padrões de Projetos**. 2. ed. [S.l.]: Alta Books, 2009.

GAMMA, E. et al. **Padrões de Projeto: Solução Reutilizáveis de Software Orientado a Objeto**. 1. ed. [S.l.]: Bookman, 2000.

GOJARE, S.; JOSHI, R.; GAIGAWARE, D. Analysis and design of Selenium Webdriver automation testing framework. **Procedia Computer Science**, v. 50, n. 1, p. 341–346, 2015.

GONÇALVES, J. S. **O uso de programação reflexiva para o desenvolvimento de aplicações comerciais adaptativas**. Projeto de Iniciação Científica. Instituto Municipal de Ensino Superior de Assis, 2012. 1-31 p.

GROCEVS, A.; PROKOFJEVA, N. The capabilities of automated functional testing of programming assignments. **Procedia - Social and Behavioral Sciences**, v. 228, p. 457–461, 2016.

GUPTA, S. **Logging in Java with the JDK 1.4 Logging API and Apache log4j**. 1. ed. [S.l.]: Apress, Berkeley, 2003.

HUNT, A.; THOMAS, D. **The Pragmatic programmer : from journeyman to master**. 1. ed. [S.l.]: Addison-Wesley, 2000.

KRAJNC, A.; HERICKO, M. Classification of object-oriented frameworks. **The IEEE Region 8 EUROCON. Computer as a Tool.**, v. 2, n. 1, p. 57–61, 2003.

KUMAR, A.; SAXENA, S. Data driven testing framework using Selenium Webdriver. **International Journal of Computer Applications**, v. 118, n. 18, p. 1–6, 2015.

MASUHARA, H. et al. Object-oriented concurrent reflective languages can be implemented efficiently. **ACM Special Interest Group on Programming Languages Notices (ACM SIGPLAN Notice)**, v. 27, p. 127–144, 1992.

MåRTENSSON, F. **Software architecture quality evaluation**. 1. ed. [S.l.]: Blekinge Institute of Technology, 2006.

OLAN, M. Unit testing: Test early, test often. **Consortium for Computing Sciences in Colleges**, v. 19, n. 2, p. 319–328, 2003.

ORACLE Java Documentation. 2021. <https://docs.oracle.com/javase/8/docs/technotes/guides/language/varargs.html>. Acessado: 2021-11-01.

PETROVA-ANTONOVA, D.; ILIEVA, S.; MANOVA, D. Tassa: Testing framework for web service orchestrations. In: **10th International Workshop on Automation of Software Test (AST)**. [S.l.: s.n.], 2015. p. 8–12.

PFLEEGER, S. L. **Engenharia de Software: Teoria e prática**. 2. ed. [S.l.]: Pearson Prentice Hall, 2004.

PRAZINA, I. et al. Testing web page layouts using galen framework tests generated by the mockup tool. In: **27th Telecommunications Forum (TELFOR)**. [S.l.: s.n.], 2019. p. 1–4.

PRESSMAN, R. S.; MAXIM, B. R. **Software Engineering: a practitioner's approach**. 8. ed. [S.l.]: McGrawHill Education, 2014. 941 p.

RAMYA, P.; SINDHURA, V.; SAGAR, P. V. Testing using Selenium web driver. In: **2nd International Conference on Electrical, Computer and Communication Technologies (ICECCT)**. [S.l.: s.n.], 2017. p. 1–7.

RIEHLE, D. JUnit 3.8 documented using collaborations. **ACM SIGSOFT Software Engineering Notes**, v. 33, n. 5, p. 1–28, 2008.

SILVA, P. C. B. d.; ALVES, T. S.; BRUNO, E. A. Automação de testes funcionais: testes funcionais automatizados de software. **Revista de Ciências Exatas e Tecnologia**, v. 6, n. 6, p. 113–133, 2015.

SINGH, J.; SHARMA, M. Performance evaluation and comparison of Sahi Pro and Selenium Webdriver. **International Journal of Computer Applications**, v. 129, n. 8, p. 23–26, 2015.

SOCOLOWSKI, C.; ALARCON, A.; ANTONIO, A. T. d. Flextest – um framework flexível para a automação de testes. In: **X Encontro Anual de Computação (EnAComp)**. [S.l.: s.n.], 2013. p. 105–112.

TERRA, R. Tratativa de falhas e automação de testes funcionais: O caso do WMS GTI Plug. In: **XI Brazilian Conference on Software: Theory and Practice (CBSoft), Industry Track**. [S.l.: s.n.], 2020. p. 1–4.

TRABOUSSY, R. **Selenium record and playback vs Cabelenium: uma análise comparativa para automação de testes funcionais**. Trabalho de conclusão de curso de Ciência da Computação. Universidade Federal de Lavras, 2021. 1-28 p.

WAHL, N. J. An overview of regression testing. **ACM SIGSOFT Software Engineering Notes**, v. 24, n. 1, p. 69–73, 1999.

WANG, X.; XU, P. Build an auto testing framework based on selenium and fitnesse. In: **1st International Conference on Information Technology and Computer Science**. [S.l.: s.n.], 2009. v. 2, p. 436–439.

WANG, X.; ZHOU, B.; LI, W. Model based load testing of web applications. In: **3rd International Symposium on Parallel and Distributed Processing with Applications (ISPA)**. [S.l.: s.n.], 2010. p. 483–490.