UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

DESENVOLVIMENTO E REUTILIZAÇÃO DE TESTES AUTOMATIZADOS EM APLICAÇÕES WEB

TRABALHO DE GRADUAÇÃO

Lucas Antunes Amaral

Santa Maria, RS, Brasil

2014

DESENVOLVIMENTO E REUTILIZAÇÃO DE TESTES AUTOMATIZADOS EM APLICAÇÕES WEB

Lucas Antunes Amaral

Trabalho de Graduação apresentado ao Curso de Ciência da Computação da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para a obtenção do grau de

Bacharel em Ciência da Computação

Orientadora: Profª. Drª. Andrea Schwertner Charão

Santa Maria, RS, Brasil

2014

Universidade Federal de Santa Maria
Centro de Tecnologia
Curso de Ciência da Computação

A Comissão Examinadora, abaixo assinada,
aprova o Trabalho de Graduação

DESENVOLVIMENTO E REUTILIZAÇÃO DE TESTES AUTOMATIZADOS EM APLICAÇÕES WEB

elaborado por
Lucas Antunes Amaral

como requisito parcial para obtenção do grau de
Bacharel em Ciência da Computação

COMISSÃO EXAMINADORA:

Andrea Schwertner Charão, Drª.
(Presidente/Orientadora)

, Profª. Dr. (UFSM)

, Prof. Dr. (UFSM)

Santa Maria, de Outubro de 2014.

RESUMO

Trabalho de Graduação
Curso de Ciência da Computação
Universidade Federal de Santa Maria

DESENVOLVIMENTO E REUTILIZAÇÃO DE TESTES AUTOMATIZADOS EM APLICAÇÕES WEB

AUTOR: LUCAS ANTUNES AMARAL
ORIENTADORA: ANDREA SCHWERTNER CHARÃO
Local da Defesa e Data: Santa Maria, de Outubro de 2014.

A constante busca pela qualidade de uma solução em forma de software, fez com que empresas do ramo de desenvolvimento, aderissem e enxergassem a importância da realização de testes automatizados em seus sistemas. A partir deste cenário, surgiram inúmeras ferramentas e frameworks para suprir esta demanda, que se propõe a ampliar a otimização de tempo e eficácia das aplicações implementadas, visando uma garantia maior na qualidade das mesmas. Contudo, é sabido que criar um novo teste para cada nova funcionalidade ou demanda do sistema, se torna muito custoso, sendo necessário um grande desprendimento de recursos humanos. Assim, este trabalho objetiva apresentar scripts de testes automatizados para sistemas web, que possam, de maneira mais genérica e com poucas alterações, serem reutilizados, de forma escalável, para novos casos de testes, que sigam o mesmo escopo.

Palavras-chave: Qualidade de Software. Testes automatizados de Software. Linguagens de Programação. Selenium HQ. Cucumber.

SUMÁRIO

1 INTRODUÇÃO

Dada a atual conjectura do mercado de desenvolvimento de software, é fundamental para que uma aplicação se mantenha viva de forma competitiva, apresentar diferenciais ao seu público alvo. Desta maneira a área de qualidade de software ganha cada vez mais espaço dentro das empresas de TI e, em especial, as ferramentas e metodologias de teste de software ganham maior visibilidade.

Os testes de software podem ocorrer em todas as etapas do desenvolvimento e de diferentes formas, contudo, sempre objetivam atender na totalidade os requisitos do sistema e, simultaneamente, amplificar a qualidade da solução codificada. São inúmeras as vantagens de se utilizar testes automatizados ao invés dos testes manuais em uma aplicação. Apesar de, aparentemente, ser mais prático e rápido realizar um teste manual, a cada nova alteração em um módulo do sistema, o teste tem que ser todo refeito e a tendência que novos erros sejam gerados até mesmo em funcionalidades já testadas é enorme, problema este que não ocorre quando a abordagem escolhida é a automatização dos testes.

Mesmo apresentando grandes vantagens, os testes automatizados demandam um grande custo inicial em sua codificação e, com isso, aumenta o envolvimento da equipe de qualidade. Pensando nesse problema, podemos buscar formas alternativas para que se possa usufruir de todas estas virtudes dos testes automatizados, e, ao mesmo tempo, utilizar de forma eficiente os recursos disponíveis em uma instituição.

1.1 Objetivos

1.1.1 Objetivo Geral

Este trabalho tem como objetivo principal apresentar um conjunto de casos de teste e testes que possam ser reutilizados de forma otimizada em novas funcionalidades de uma aplicação ou em sistemas que sigam os mesmos padrões e comportamento dos softwares conhecidos.

1.1.2 Objetivos Específicos

- Apontar diferenças de casos de testes;
- Gerar testes automatizados reaplicáveis em novos casos;

- Gerar casos de testes genéricos para um escopo definido.

1.2 Justificativa

A qualidade de software é uma das variáveis essenciais para que um projeto de software tenha sucesso. Sendo assim, torna-se cada vez mais necessário a inserção de testes automatizados em projetos web, agregando aos mesmos uma maior confiabilidade e redução nos possíveis erros que o sistema possa apresentar. Para que haja a possibilidade de aumentar a qualidade dos sistemas, sem que seja necessário uma maior demanda de recursos humanos para a área de qualidade, podemos adotar práticas de reuso de códigos de testes, visando maximizar a produtividade e eficiência, além de, simultaneamente, obter um produto final com uma garantia de qualidade superior.

2 FUNDAMENTOS E REVISÃO DE LITERATURA

Neste capítulo, serão apresentados conceitos relativos os conteúdos abordados neste trabalho, descrevendo, qualidade de software, ferramentas de teste de software, assim como o reuso de testes.

2.1 Qualidade de Software

A Qualidade de software é uma subárea oriunda da engenharia de software, que tem com foco central apresentar metodologias, procedimentos e métricas que garantam a qualidade no processo de desenvolvimento de um sistema. Apesar de ocorrer no processo, a Qualidade de software objetiva obter qualidade no produto final, e, com isso, consiga contemplar na totalidade os requisitos tratados com o cliente ao longo do processo. **?** (**?**) afirma que a qualidade de software está diretamente relacionada a um gerenciamento rigoroso de requisitos, uma gerência efetiva de projetos e em um processo de desenvolvimento bem definido, gerenciado e em melhoria contínua. Afirma também, que atividades de verificação e uso de métricas para controle de projetos e processo também estão inseridas nesse contexto, contribuindo para tomadas de decisão e para antecipação de problemas.

Mas como medir a qualidade de um sistema em questão? Para responder este questionamento, Garvin (**?**) propõe o conceito que ele chama de oito dimensões, que seriam em ordem, qualidade do desempenho, qualidade dos recursos, confiabilidade, conformidade, durabilidade, facilidade de manutenção, estética e percepção. Segundo **?**, atendendo a estes oito critérios, o sistema apresentará qualidade. **?** (**?**) complementa a definição de Garvin mesclando-a com a norma ISO 9126, e aponta os fatores críticos para o sucesso neste caso, como sendo: Intuição, Eficiência, Robustez e Riqueza.

2.1.1 Qualidade do processo

A qualidade de software é largamente determinada pela qualidade dos processos utilizados para o desenvolvimento. Deste modo, a melhoria da qualidade de software é obtida pela melhoria da qualidade dos processos (**?**).

2.1.2 Qualidade do produto

Existe uma relação direta entre qualidade de produto e qualidade do processo, pois, para obtenção da qualidade do produto final, faz-se necessário adquirir primeiramente qualidade nos processos que compõem o desenvolvimento do mesmo. Avaliar a qualidade de um produto de software é verificar, através de técnicas e atividades operacionais o quanto os requisitos são atendidos. Tais requisitos, de uma maneira geral são a expressão das necessidades, explicitados em termos quantitativos ou qualitativos, e têm por objetivo definir as características de um software, a fim de permitir o exame de seu atendimento (**?**).

2.1.3 Testes de Software

É a atividade responsável por apresentar os erros existentes em um determinado programa, por isso, pode ser vista como uma atividade destrutível, pois visa expor os defeitos para depois corrigir os mesmos, e, de preferência, em um estágio inicial. Quanto mais tarde um defeito for identificado mais caro fica para corrigi-lo e mais, os custos de descobrir e corrigir o defeito no software aumentam exponencialmente na proporção em que o trabalho evolui através das fases do projeto de desenvolvimento (**?**). O teste possibilita também, validar se os requisitos inicias do sistema, alinhados pelos *stakeholders*, estão contemplados em sua plenitude.

Apesar de não ser possível, através de testes, provar que um programa está correto, os testes, se conduzidos sistemática e criteriosamente, contribuem para aumentar a confiança de que o software desempenha as funções especificadas e evidenciar algumas características mínimas do ponto de vista da qualidade do produto (**?**). Sendo assim, se faz essencial o mapeamento de um processo de testes para que se possa criar garantias e métricas que reduzam os erros, maximizando a qualidade, **?** (**?**) descreve o processo de teste, como sendo a composição de quatro macro etapas: Planejamento, projeto, execução e acompanhamento dos testes de unidade.

2.1.4 Estratégias de Testes

A Estratégia de testes se caracteriza pela definição da abordagem geral a ser aplicada nos testes, descrevendo como o software será testado, identificando os níveis de testes que serão aplicados, os métodos, técnicas e ferramentas a serem utilizada (**?**). Existem muitas estratégias mas podemos destacar entre elas: Teste de unidade, integração, sistema, aceitação e regressão, funcional e carga. Se pode também, mesclar mais de uma estratégia com o intuito de reduzir os

possíveis defeitos que o sistema venha a ter.

2.2 Ferramentas para teste de software

Existem muitas ferramentas desenvolvidas para realização de testes de software web, neste trabalho serão descritas duas(2) dentre elas, que serão as ferramentas utilizadas no desenvolvimento dos testes estudados.

2.2.1 Cucumber

Cucumber é um ferramenta de desenvolvimento de testes, voltado para sistemas web, que adota uma linguagem de alto nível bem próxima à uma linguagem natural e tem suas origens fixadas sobre a metodologia BDD (Behavior Driven Development). Cucumber é escrita em linguagem Ruby, mas pode ser utilizada para executar especificações de aplicações escritas em qualquer linguagem (**?**).

A escolha dessa ferramenta, baseou-se na fácil transcrição dos requisitos do sistema para a linguagem em questão, tornando possível conferir se os requisitos estão contemplados pelas funções e métodos descritos no sistema *web*. **?**(**?**) compara Cucumber com o *software* Capybara e afirma que o primeiro apresentar um código mais legível e amigável. Uma observação é que o código com Capybara faz referência para vários detalhes de implementação, enquanto o código do Cucumber reserva isso apenas para os *Steps* e não para o arquivo de feature (**?**).

A ferramenta funciona basicamente através da leitura de arquivos com a extensão feature, os quais descrevem em linguagem natural uma funcionalidade e casos de teste, conhecidos como cenários. Como os testes estão escritos em uma linguagem natural, e não de programação, Cucumber precisa pesquisar pelo código associado aos passos que formam o cenário em arquivos auxiliares (**?**). Cucumber executa seus arquivos *.feature*, e esses arquivos contêm especificações executáveis escritos em uma linguagem chamada Gherkin (**?**), que, possui um layout bem definido. Inicia pela descrição de uma funcionalidade, que por sua vez possui cenários, onde, um cenário é descrito da seguinte forma: *Dado* alguma condição *Quando* outra condição *E* terceira condição *Então* faça algo, conforme ilustra a figura **??**.

```
Feature: Sign up

  Scenario: Successful sign up
    Given I am on the homepage
    When I follow the sign up link
    And I fill out the form with valid details
    Then I should receive a confirmation email
    And I should see a personalized greeting message
```

Figura 2.1 – Um exemplo de código cucumber.

2.2.2 Selenium HQ

É um framework open source, utilizado para automatização de testes funcionais em aplicações web (**?**). Segundo **?** (**?**), se trata de uma ferramenta de fácil uso e eficiente para desenvolver casos de teste, permitindo os testes de aceitação ou funcional, regressão e de desempenho. Selenium trabalha como um plugin do navegador Firefox, o mesmo traz muita praticidade pois permite que se possa capturar cliques e valores digitados transformando-os em um caso de teste. Ele é composto por quatro ferramentas: Selenium IDE, Selenium Grid, Selenium RC e Selenium WebDrive.

A utilização de comandos no Selenium consiste em digitar o comando seguido de dois parâmetros tal como por exemplo *verifyText //div//a[2] Login*. Dependendo do comando, os parâmetros poderão ser opcionais, aliás, alguns comandos não necessitam de parâmetros para serem executados (**?**).

Selenium foi escolhida como umas das ferramentas no desenvolvimento deste projeto pelo fato de ser um *framework open source* que possui um vasto leque de ferramentas para testes de sistemas web. Outro fator importante para sua escolha, é a possibilidade de interação do Selenium HQ com o Cucumber. Ela se destaca entre as demais ferramentas gratuitas pelo fato de ser a mais completa, permitindo integração com várias linguagens, outros *frameworks*, além de suportar inúmeros navegadores e sistemas operacionais (**?**).

2.3 Reuso de testes

Visando melhor aproveitar os recursos existentes em uma instituição e ainda assim apresentar garantias na qualidade do produto/serviço entregues aos clientes, desenvolvedores do mundo todo, começaram a apresentar teses e modelos que buscam criar testes genéricos e pa-

dronizados. Um padrão é um pedaço de informação instrutiva e nomeada, que captura a estrutura essencial e *"insights"* de uma família bem sucedida de soluções aprovadas para um determinado problema, o qual surge em um determinado contexto (**?**).

? diz que por razões histórias a área de desenvolvimento de software não atingiu a maturidade que outras áreas da engenharia atingiram, complementa afirmando, que apesar disso, é inegável que algum avanço tenha sido alcançado, pois a forma de realização dessa atividade evoluiu de uma atividade realizada de forma quase artesanal, para um processo de desenvolvimento bem estruturado e que, nos melhores casos, contempla inclusive atividades de gerência e avaliação da qualidade (**?**). Com isso, a reutilização dos testes já codificados, se mostra uma importante prática no desenvolvimento e que ainda apresenta muitas incógnitas e possibilidades para as equipes de TI, principalmente na geração de casos de testes que possam ser reutilizados em situações que apresentem um padrão parecido com os casos já conhecidos.

3 DESENVOLVIMENTO

Abaixo, serão descritas as atividades desenvolvidas afim de alcançar os objetivos propostos por este trabalho. Será apresentada a estrutura básica dos sistemas escolhidos para inserção dos testes automatizados, assim como trechos dos códigos desenvolvidos e estrutura das soluções encontradas ao longo do desenvolvimento do projeto.

3.1 Escolha dos sistemas

Para desenvolver os testes automatizados deste trabalho, foram escolhidos 2 sistemas web desenvolvidos na linguagem de programação Java, com suas interfaces desenvolvidas em *JSP*, contando com funções *Ajax* e *Javascript*.

3.2 Organização do Código - Solução 1

Em um primeiro momento, com objetivo de automatizar os testes de uma aplicação e reduzir o trabalho na codificação de novos testes para as equipes de desenvolvimento e testes de uma empresa, criou-se uma abordagem baseada em anotações de classes Java com o intuito de mapear qual seriam as classes e métodos utilizados nos formulários do software que deveriam ser testados.

A ideia inicial seria informar através de uma anotação, qual deveria ser o teste executado para cada método mapeado com a anotação em questão, sendo assim, inserindo uma anotação sobre um método, se saberia qual propriedade aquele método corresponderia em um determinado formulário, assim como o tipo do mesmo (input, text, select, radiobuttom, checkbox) e o valor que deveria ser inserido no mesmo. Informações como o formulário que deveria ser testado e o valor de aceitação para o teste seriam inseridos na anotação da classe. Para realizar esta tarefa foi descrita a seguinte classe de anotação.

3.2.1 `Teste.java`

A interface `Teste.java` procura com seus métodos mapear as ações que serão necessárias para a execução de um teste *Selenium* utilizando elementos da classe *WebDriver*, conforme apresentaremos mais adiante neste experimento.

Abaixo temos uma classe que demonstra a forma como ocorre as anotações utilizando a

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.METHOD})
public @interface Teste {
    String getUrl() default "";

    //findElement
    String getCampo() default ""; //campo html do formulário
    String getIdentificador() default "id"; //Informar se deve buscar um id, name, class ou
        css

    boolean isSelect() default false;

    String getValor() default ""; //utilizado como sendKeys e selectText
    boolean click() default false;
    boolean submit() default false;
    boolean limpar() default false;

    String getTipoAssert() default "igual";
    String getCampoAssert() default "";
    String getIdentificadorAssert() default "id";
    String getValorEsperadoAssert() default "";
    String getAtributoCampoComparacaoAssert() default "texto";

    boolean fazerLogin() default false;
    String getLogin() default "colegiado";
    String getSenha()default "";
}
```

Figura 3.1 – Interface Teste

interface Teste.

3.2.2 `TesteFormulario.java`

Para realizar o teste propriamente dito, criou-se nos projetos utilizados, uma classe de testes denominada `TesteFormulaio.java`, onde a mesma utilizaria as anotações do *framework JUnit* para executar os testes necessários. A classe desenvolvida possui um método principal denominado `testaFormularios`, onde a ideia principal do mesmo seria buscar de forma recursiva, todas as classes do projeto anotadas pela interface Teste que criamos, e para cada uma das mesmas cria-se elementos *webDrive* para executar as ações indicadas pela anotação.

Para tornar o método `testaFormularios` e seus sub métodos mais genéricos, utilizou-se *Java Reflection* para tratar todas as classes e métodos simplesmente como objetos, possibilitando assim, o reuso do mesmo para qualquer teste necessário dentro deste projeto. Vemos, a seguir, o código desta classe:

```
@Teste(getUrl = "/cadastro-disciplina.htm", getCampo = "salvar", click = true
        ,getIdentificadorAssert = TestePropriedades.IDENTIFICADOR_CSS, getCampoAssert = "h4",
            getValorEsperadoAssert = "Sucesso!")
public class Disciplina {
    private Long id;
    private String codigo;
    private String nome;
    private Integer cargaHoraria;
    private Boolean preAprovada;
    private Boolean ativa;
    private Instituicao instituicao;

    @Teste(getCampo = "ativa1", click = true)
    public Boolean getAtiva() {
        return ativa == null || ativa;
    }

    public void setAtiva(Boolean ativa) {
        this.ativa = ativa;
    }

    @Teste(getCampo = "preAprovada1", click = true)
    public Boolean getPreAprovada() {
        return preAprovada != null && preAprovada;
    }

    public void setPreAprovada(Boolean preAprovada) {
        this.preAprovada = preAprovada;
    }

    @Teste(getCampo = "cargaHoraria", getValor = "60", isSelect = true)
    public Integer getCargaHoraria() {
        return cargaHoraria;
    }

    public void setCargaHoraria(Integer cargaHoraria) {
        this.cargaHoraria = cargaHoraria;
    }

    @Teste(getCampo = "nome", getValor = "Disciplina teste")
    public String getNome() {
        return nome;
    }
}
```

Figura 3.2 – Classe anotada com @Teste

3.2.2.1 `testaFormularios()`

Função principal da classe de testes, responsável por, de forma iterativa, buscar em cada classe que possui a anotações Teste do projeto em questão.

3.2.2.2 `executaTeste()`

Método responsável por encontrar no browser o campo necessário e passá-lo para o método que irá executar as ações necessárias.

```
@Test
public void testaFormularios() {
  for (Class classe : getCarregaClasses()) {
        Teste testeClasse = TestePropriedades.teste(classe);
        if (testeClasse.fazerLogin()) {
            LoginTeste.login(TestePropriedades.urlSistema, testeClasse.getSenha(),
                testeClasse.getLogin(), webDriver);
        }
        if (!testeClasse.getUrl().equals("")) {
            webDriver.get(TestePropriedades.urlSistema + testeClasse.getUrl());
        }
        for(Method metodo: classe.getDeclaredMethods()) {
            Teste teste = TestePropriedades.teste(metodo);
            if (teste != null) {
                executaTeste(teste, true);
            }
        }
        executaTeste(testeClasse, false);
        System.out.println("Formulário da classe " + classe.getName() + " testado!");
  }
}
```

```
public void executaTeste(Teste teste, Boolean submeteUrl) {
    WebElement webElement = getEncontraCampo(teste.getIdentificador(), teste.getCampo());
    if (webElement != null) {
        executaAcoes(teste, webElement, submeteUrl);
    }
}
```

3.2.2.3 `executaAcoes()`

Recebe como parâmetro a anotação atual e o elemento html(WebDriver), e com isso executa as ações necessárias.

```
private void executaAcoes(Teste teste, WebElement webElement, Boolean submeteUrl) {
  if (!teste.getUrl().equals("") && submeteUrl) {
        webDriver.get(TestePropriedades.urlSistema + teste.getUrl());
    }
    if (teste.limpar()) {
        webElement.clear();
    }
    if (teste.isSelect()) {
        Select select = new Select(webElement);
        if (!teste.getValor().equals("")) {
            select.selectByVisibleText(teste.getValor());
        }
    } else if (!teste.getValor().equals("")) {
        webElement.sendKeys(teste.getValor());
    }
    if (teste.click()) {
        webElement.click();
    }
    if (teste.submit()) {
        webElement.submit();
    }
    if (!teste.getCampoAssert().equals("")) {
        executaComparacao(teste);
    }
}
```

3.2.2.4 executaComparacao()

Ultimo estágio do teste unitário, onde o mesmo confere se o valor esperado pelo formulário é o mesmo apresentado pelo browser após executar todas as ações mapeadas.

```
private void executaComparacao(Teste teste) {
  WebElement webElementAssert = getEncontraCampo(teste.getIdentificadorAssert(),
      teste.getCampoAssert());
    if (teste.getTipoAssert().equals(TestePropriedades.TIPO_ASSERT_IGUAL) &&
        !teste.getValorEsperadoAssert().equals("")) {
        assertEquals(teste.getValorEsperadoAssert(),
            getValorPropriedadeCampo(webElementAssert, teste));
    } else if (teste.getTipoAssert().equals(TestePropriedades.TIPO_ASSERT_DIFERENTE) &&
        !teste.getValorEsperadoAssert().equals("")) {
        assertNotEquals(teste.getValorEsperadoAssert(),
            getValorPropriedadeCampo(webElementAssert, teste));
    } else if (teste.getTipoAssert().equals(TestePropriedades.TIPO_ASSERT_COLECAO_IGUAL) &&
        !teste.getValorEsperadoAssert().equals("")) {
        //todo: preciso de um objeto e não string
    } else if (teste.getTipoAssert().equals(TestePropriedades.TIPO_ASSERT_VERDADEIRO)) {
        if (!teste.getValorEsperadoAssert().equals("")) {
            assertTrue(teste.getValorEsperadoAssert(), (Boolean)
                getValorPropriedadeCampo(webElementAssert, teste));
        } else {
            assertTrue((Boolean) getValorPropriedadeCampo(webElementAssert, teste));
        }
    } else if (teste.getTipoAssert().equals(TestePropriedades.TIPO_ASSERT_FALSO)) {
        if (!teste.getValorEsperadoAssert().equals("")) {
            assertFalse(teste.getValorEsperadoAssert(), (Boolean)
                getValorPropriedadeCampo(webElementAssert, teste));
        } else {
            assertFalse((Boolean) getValorPropriedadeCampo(webElementAssert, teste));
        }
    } else if (teste.getTipoAssert().equals(TestePropriedades.TIPO_ASSERT_NULO)) {
        if (!teste.getValorEsperadoAssert().equals("")) {
            assertNull(teste.getValorEsperadoAssert(),
                getValorPropriedadeCampo(webElementAssert, teste));
        } else {
            assertNull(getValorPropriedadeCampo(webElementAssert, teste));
        }
    } else if (teste.getTipoAssert().equals(TestePropriedades.TIPO_ASSERT_NAO_NULO)) {
        if (!teste.getValorEsperadoAssert().equals("")) {
            assertNotNull(teste.getValorEsperadoAssert(),
                getValorPropriedadeCampo(webElementAssert, teste));
        } else {
            assertNotNull(getValorPropriedadeCampo(webElementAssert, teste));
        }
    }
}
```

3.2.2.5 getValorpropriedadeCampo()

Retorna o elemento html encontrado no navegador utilizando as informações inseridas na anotação do método/classe.

3.2.2.6 getEncontraCampo()

Retorna o elemento html procurado.

```
private Object getValorPropriedadeCampo(WebElement webElement, Teste teste) {
  if (teste.getAtributoCampoComparacaoAssert()
    .equals(TestePropriedades.ATRIBUTO_COMPARACAO_ASSERT_TEXTO)) {
        return webElement.getText();
    } else if (teste.getAtributoCampoComparacaoAssert()
        .equals(TestePropriedades.ATRIBUTO_COMPARACAO_ASSERT_NOME_TAG)) {
        return webElement.getTagName();
    } else if (teste.getAtributoCampoComparacaoAssert()
        .equals(TestePropriedades.ATRIBUTO_COMPARACAO_ASSERT_ATRIBUTO)) {
        return null;
    } else if (teste.getAtributoCampoComparacaoAssert()
        .equals(TestePropriedades.ATRIBUTO_COMPARACAO_ASSERT_VALOR_CSS)) {
        return null;
    } else if (teste.getAtributoCampoComparacaoAssert()
        .equals(TestePropriedades.ATRIBUTO_COMPARACAO_ASSERT_CAMPO_EXIBIDO)) {
        return webElement.isDisplayed();
    } else if (teste.getAtributoCampoComparacaoAssert()
        .equals(TestePropriedades.ATRIBUTO_COMPARACAO_ASSERT_CAMPO_SELECIONADO)) {
        return webElement.isSelected();
    } else if (teste.getAtributoCampoComparacaoAssert()
        .equals(TestePropriedades.ATRIBUTO_COMPARACAO_ASSERT_CAMPO_HABILITADO)) {
        return webElement.isEnabled();
    }
    return null;
}
```

```
private WebElement getEncontraCampo(String identificador, String campo) {
  if (identificador.equals(TestePropriedades.IDENTIFICADOR_ID)) {
    return webDriver.findElement(By.id(campo));
    } else if (identificador.equals(TestePropriedades.IDENTIFICADOR_NOME)) {
        return webDriver.findElement(By.name(campo));
    } else if (identificador.equals(TestePropriedades.IDENTIFICADOR_NOME_CLASSE)) {
        return webDriver.findElement(By.className(campo));
    } else if (identificador.equals(TestePropriedades.IDENTIFICADOR_CSS)) {
        return webDriver.findElement(By.cssSelector(campo));
    } else if (identificador.equals(TestePropriedades.IDENTIFICADOR_XPATH)) {
        return webDriver.findElement(By.xpath(campo));
    }
    return null;
}
```

Após a finalização do desenvolvimento desta primeira solução encontrada, notou-se a clara necessidade de alteração na sistemática de como os teste ocorreriam, necessidade esta oriunda de algumas limitações que a solução trás, como por exemplo, a não possibilidade de realizar testes que envolvam mais de um método, tornando praticamente inviável a tarefa de realizar testes funcionais, pois não seria possível mapear um cenário mais completo, que depende de mais de uma variável e que tem uma sequência bem mapeada que deve ser seguida.

Outro fator determinante para o descontinua mento desta solução, foi o fato de por permitir apenas testes unitários, o mesmo não possibilitaria o teste de um mesmo campo, com dois valores diferentes e além disto, também não seria viável a execução de testes unitários para um sistema muito grande, pois é sabido que os testes devem ser executados sobre os pontos críticos do software e não na totalidade do projeto.

4 PRÓXIMAS ETAPAS

Como continuação do trabalho realizado até o momento, são planejadas as seguintes atividades: