**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**

**2015**

# 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**

**2015**

# AGRADECIMENTOS

# 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, 4 de Dezembro de 2015.

A constante busca pela qualidade de uma solução em forma de software, fez com que empresas do ramo de desenvolvimento aderissem a realização de testes automatizados em seus sistemas. A partir deste cenário, surgiram inúmeras ferramentas e *frameworks* para suprir esta demanda, que se propõem 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, torna-se muito custoso, sendo necessário um grande desprendimento de recursos humanos. Assim, este trabalho apresenta abordagens reutilizáveis de teste que objetivam trazer praticidade e facilidade na confecção de novos testes, utilizando para isso métodos de classes genéricas de testes confeccionados no mesmo, especificamente voltados para sistemas web desenvolvidos na linguagem de programação Java.

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

# SUMÁRIO

# LISTA DE FIGURAS

# LISTA DE TABELAS

# 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 tecnologia da informação e, em especial, as ferramentas e metodologias de teste de software ganham maior visibilidade. Além do aumento da produtividade e da diminuição do retrabalho, as empresas na área de tecnologia da informação buscam um melhor relacionamento com seus clientes por meio do melhor planejamento e gestão de suas atividades de desenvolvimento e da diminuição no número de defeitos nos produtos entregues (JOMORI; VOLPE; ZABEU, 2004).

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. 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.

Os testes automatizados angariam cada vez mais adeptos ao longo dos últimos anos. Tal fato se deve principalmente à grande redução de custo observada a médio e longo prazos com o uso desta prática (OLIVEIRA et al., 2007). Em contrapartida, os testes automatizados demandam um grande custo inicial em sua codificação e, com isso, aumentam 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 classes 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 para os

quais foram desenvolvidos.

### 1.1.2 Passos de metodologia

- Apresentar solução para testes unitários utilizando interface de anotação + Selenium WebDriver;
- Exibir solução para testes funcionais utilizando Cucumber + Selenium WebDriver;
- Apresentar solução para testes funcionais utilizando Selenium WebDriver;
- Expor qualidades e defeitos das soluções apresentadas.

## 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 necessária 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 necessária 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 aos 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 como 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, conseguir contemplar na totalidade os requisitos tratados com o cliente ao longo do processo. VASCONCELOS et al. (2006) afirmam 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. Afirmam 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 (1987) 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 GARVIN, atendendo a estes oito critérios, o sistema apresentará qualidade. PRESSMAN (2011) 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 (KOSCIANSKI; SOARES, 2007).

### 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 tem por objetivo definir as características de um software, a fim de permitir o exame de seu atendimento (KOSCIANSKI; SOARES, 2007).

## 2.2 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 (BOEHM; BROWN; LIPOW, 1976). 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 (MALDONADO et al., 2004). Sendo assim, faz-se essencial o mapeamento de um processo de testes para que se possa criar garantias e métricas que reduzam os erros, maximizando a qualidade, CRESPO et al. (2004) descrevem o processo de teste como sendo a composição de quatro macro etapas: Planejamento, projeto, execução e acompanhamento dos testes de unidade.

### 2.2.1 Plano de teste

Embora pareça ideal realizar testes sobre o projeto inteiro, muitas vezes esta tarefa pode torna-se inviável devido ao tamanho e complexidade do sistema em questão. Desta forma, torna-se necessário descrever as áreas e testes necessários em um sistema. Podemos retratar esta

situação com um plano de teste. O plano de teste estrutura e organiza as informações referentes ao processo de teste durante todo o projeto, facilitando assim o planejamento do Projeto de Teste (PIQUEIRO; ZADRA, 2015). Pode ser visto como uma etapa de planejamento do projeto e pode ser descrito como um conjunto de casos de teste.

### 2.2.2 Caso de teste

O propósito do caso de teste é definir uma unidade de teste que será executada pelo testador, seja manual ou automaticamente (RIOS, 2006). Um caso de teste é constituído visando descrever passos sequenciais necessários para validação de um artefato de teste. O caso de teste deve especificar a saída esperada e os resultados esperados do processamento (MYERS; SANDLER; BADGETT, 2011).

### 2.2.3 Tipos de Teste

Existem muitos tipos de teste, mas podemos destacar entre elas: teste de unidade, integração, sistema, aceitação e regressão. Serão descritos nas próximas subseções os tipos utilizados no desenvolvimento deste trabalho.

#### 2.2.3.1 Teste unitário

O teste de unidade focaliza o esforço de verificação na menor unidade de projeto do software (PRESSMAN, 2011). Em uma aplicação orientada a objeto, o teste, neste caso, seria voltado ao método.

#### 2.2.3.2 Teste funcional

Baseando-se numa descrição *black-box* da funcionalidade do software, o teste funcional é capaz de checar se o software está de acordo com sua especificação independentemente da sua implementação (FANTINATO et al., 2005).

### 2.2.4 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 (RIOS, 2006). Deve-se também

escolher na etapa de estratégia os tipos de testes que deverão ser realizados, assim como os estágios de teste escolhidos. Uma estratégia de teste de software deve acomodar testes de baixo nível, necessários para verificar se um pequeno segmento de código fonte foi implementado corretamente, bem com testes de alto nível, que validam as funções principais do sistema de acordo com os requisitos do cliente (PRESSMAN, 2011).

Tradicionalmente, entre desenvolvedores de teste, utiliza-se uma estratégia onde temos um equilíbrio entre testar o sistema após pronto e testar somente antes de desenvolver o sistema. Esta estratégia assume uma visão incremental do teste, começando com o teste de unidades individuais de programa, passando para os testes destinados a facilitar a integração de unidades e culminando com testes que usam o sistema concluído (PRESSMAN, 2011).

### 2.2.5 Testes de Sistemas Web

O teste de aplicações Web se reveste de características peculiares, uma vez que se tratam, em geral, de aplicações colocadas na Internet e cujo acesso, em teoria, é aberto a qualquer usuário (RIOS, 2006). Sendo assim os tais sistemas diferem-se das aplicações web pelos cuidados necessários com questões como a possibilidade de utilização de sistemas operacionais e navegadores distintos, segurança e conectividade. Além disso, a relevância econômica das aplicações Web aumenta a importância de controlar e de melhorar a sua qualidade (FIDELIS; MARTINS, 2004).

## 2.3 Ferramentas para teste de software

Existem muitas ferramentas desenvolvidas para realização de testes de software web. Neste trabalho serão descritas três(3) dentre elas, que serão as ferramentas utilizadas no desenvolvimento dos testes estudados.

### 2.3.1 Selenium HQ

É um *framework* open source utilizado para automatização de testes funcionais em aplicações web (CHIAVEGATTO[1] et al., 2013). Segundo PEREIRA (2012) 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, pois, alguns comandos não necessitam de parâmetros para serem executados (SIXPENCE; ADÃO; SMITH, 2011).

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 (PEREIRA, 2012).

### 2.3.2 Cucumber

Cucumber é uma ferramenta de desenvolvimento de testes, voltada para sistemas web, que adota uma linguagem de alto nível bem próxima a 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 (NUNES, 2009).

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. LOPES (2011) compara Cucumber com o software Capybara[1] e afirma que o primeiro apresenta 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` (LOPES, 2011).

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

[1] Capybara é uma biblioteca escrita na linguagem de programação Ruby que torna fácil simular como um usuário interage com sua aplicação. http://jnicklas.github.io/capybara/

cenário em arquivos auxiliares (SCHMITZ; BECKER; BERLATTO, 2013). Cucumber executa seus arquivos `.feature` e esses arquivos contêm especificações executáveis escritos em uma linguagem chamada Gherkin (REFERENCE CUCUMBER.IO, 2015), 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 2.1.

```
Cenario: Fazer login no sistema como um aluno
  Dado acesso o endereco /login.htm
  Quando preencho o campo login com o valor lamaral buscando pelo id
  E preencho o campo senha com o valor teste123 buscando pelo id
  E clico no elemento button.btn.btn-default buscando pelo css
  Entao verifico se atributo nome tag do elemento id buscando pelo nome não está nulo
```

Figura 2.1: Um exemplo de código cucumber.

### 2.3.3 JUnit

JUnit é uma ferramenta de apoio ao teste unitário, a qual auxilia desenvolvedores na automação dos testes e verificação dos resultados (BIASI, 2006). A escolha desta ferramenta, como parte integrante da solução desenvolvida, deveu-se ao fato da mesma ser voltada para sistemas desenvolvidos na linguagem de programação Java, além de ser altamente versátil, possibilitando assim, integração em um único código da mesma com os demais *frameworks* que serão utilizados (Selenium e Cucumber).

Justifica-se a sua utilização neste trabalho por tratar-se de uma API (*Application Programming Interface*), que viabiliza a comparação de um valor obtido em algum teste com o valor esperado pelo mesmo. Desta forma, pode-se validar e verificar se a funcionalidade de um sistema está trabalhando adequadamente.

## 2.4 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 padronizados. 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 (CAGNIN et al., 2004).

GUIZZARDI diz que, por razões históricas, 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 (GUIZZARDI, 2000). 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.

Como solução em forma de reutilização de teste, (CAGNIN et al., 2004) apresentam uma abordagem composta por: a) uma estratégia, que define e associa requisitos de teste a padrões de linguagens de padrões de análise; b) diretrizes, que apoiam o engenheiro de software na decisão de quais requisitos de teste disponíveis devem ser reusados e instanciados para casos de teste concretos. A estratégia que define os requisitos de teste da abordagem proposta foi aplicada aos padrões de uma linguagem de padrões de análise, utilizados na reengenharia de um sistema legado de biblioteca. Já (KARINSALO; ABRAHAMSSON, 2004) sugerem um modelo de processo de desenvolvimento de teste que leva técnicas de reutilização de software e atividades em conta, revela ainda que, a fim de produzir material de teste reutilizável, as entidades de software têm de ser expressa em termos de recursos, em que os materiais de ensaio estão anexados.

Segundo (PATEL; KOLLANA, 2014), Tata Consultancy Services (TCS) é uma empresa provedora de serviços em TI (ITSP) que desenvolveu um repositório de casos de teste reutilizáveis com o objetivo de reduzir o custo dos testes em projetos de Implementação de software empresarial (ESI).(PATEL; KOLLANA, 2014) trás um estudo do programa de reutilização para analisar o seu impacto. Um conjunto de métricas foram definidos e os dados pertinentes foram obtidas a partir de usuários do repositório.

(CRESPO et al., 2004) apresentam uma metodologia para implantação ou melhoria do processo de teste em empresas desenvolvedoras de software, com o objetivo de viabilizar a utilização das práticas de teste pelas empresas. Apesar de não tratar-se de uma solução que contenha reutilização de software este trabalho tem relação direta com o trabalho desenvolvido, pois a reutilização nada mais visa que melhorias no processo de desenvolvimento dos testes de

software.

# 3 DESENVOLVIMENTO

A seguir serão descritas as atividades desenvolvidas a fim de alcançar os objetivos propostos por este trabalho. Será apresentado o escopo do projeto, assim como trechos dos códigos desenvolvidos e estrutura das soluções encontradas ao longo do desenvolvimento do projeto.

## 3.1 Delimitação de escopo

Como escopo, foram delimitados alguns pré-requisitos necessários para utilização da solução proposta neste trabalho, onde o sistema deve ser um software web desenvolvido na linguagem de programação Java, podendo apresentar também funções Ajax e Javascript.

## 3.2 Solução para testes unitários com anotações e Selenium

Em um primeiro momento, com objetivo de automatizar os testes de uma aplicação a fim de reduzir o esforço na confecção e codificação de novos testes para as equipes de desenvolvimento e testes de uma organização, escolheu-se desenvolver uma solução baseada em três etapas que são descritas conforme imagem 3.1.

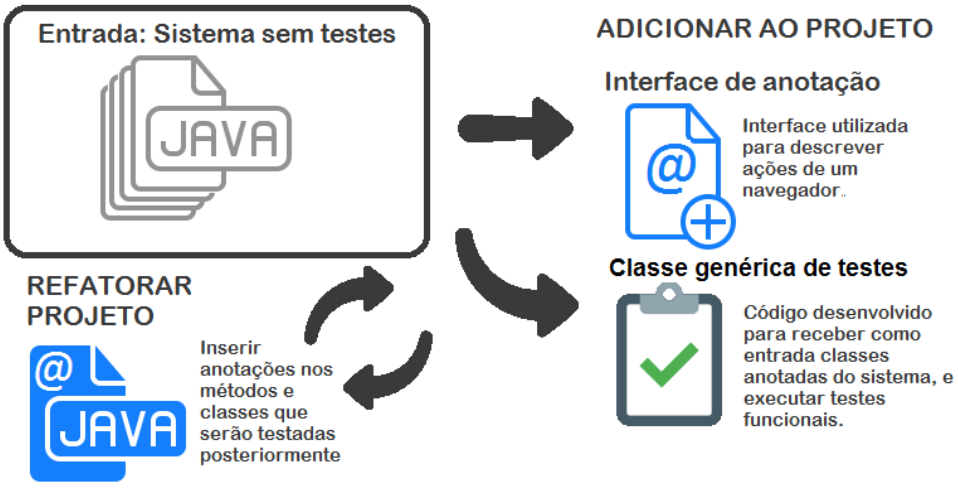


Figura 3.1: Estrutura da solução.

Temos como ponto de partida um sistema onde se possui pouco ou nenhum teste codificado. Para o mesmo descrevemos três fases no desenvolvimento de uma solução, em forma de testes para o sistema, que seriam: criar uma classe de testes genérica, responsável por realizar os

testes necessários na aplicação; descrever uma interface de anotação, que representaria ações de elementos web (*HTML*) que seriam testados posteriormente e, por fim, inserir as anotações nos métodos e classes do sistema que devem ser testados nesta aplicação.

### 3.2.1 Classe genérica de testes

Por tratar-se de uma abordagem para sistemas web, optou-se pela utilização de dois *frameworks* de testes, que possibilitam interações com elementos web, para confecção da classe responsável pela execução dos testes automatizados. O primeiro é o Selenium. Através da utilização da classe `WebDrive`, contida em seu pacote, podemos descrever uma sequência de passos que são executados por um navegador, como cliques e inserção de dados em campos de um formulário. Sendo assim, instanciando um objeto `WebDrive`, podemos iniciar um navegador e realizar uma série de instruções pré-determinadas para validar se a aplicação comporta-se de forma adequada.

Por fim, necessitamos validar se após realizar o preenchimento de formulários do software, o resultado obtido condiz com o resultado esperado. Neste momento é onde se faz necessária a adição de trechos de códigos Junit na classe de teste desenvolvida, pois o mesmo disponibiliza em seu pacote um conjunto de métodos *assert*, com os quais é possível comparar, por exemplo, se após salvar um formulário de cadastro, a mensagem de "cadastrado com sucesso"aparece na tela. A imagem 3.2 apresenta um trecho da classe confeccionada para realização dos testes.

```
@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!");
  }
}
```

Figura 3.2: Classe genérica de teste desenvolvida

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 que indica quais classes deverão ser testadas e para cada uma das mesmas cria-se elementos `webDrive` para executar as ações indicadas pela anotação.

### 3.2.2 Interface de anotação

Após finalizar o desenvolvimento da classe de testes genérica, passou a existir a necessidade de criar um mecanismo para informar, à classe de testes, quais seriam as validações necessárias. Para esta tarefa, por tratar-se de projetos Java, resolveu-se criar uma interface de anotação que seria vinculada aos métodos e classes, onde seriam necessários executar os testes posteriores. Desta forma, a classe `TestaFormulario.java`, responsável pela execução dos testes, varreria as classes do sistemas e, toda vez que encontrar uma classe anotada por essa interface, executaria um teste, onde, cada método que necessitaria de um teste possuiria, em sua anotação, o tipo do campo, as ações que deveriam ser executas e o valor que deveria ser preenchido.

Deu-se o nome de `Teste.java` para a interface, e sua estrutura está ilustrada em 3.3.

```
@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 3.3: Interface Teste

Tem-se, então, que cada método existente nesta classe representa uma ação específica da classe `WebDrive`, possibilitando, assim, manipular elementos *HTML* na execução do teste.

### 3.2.3 Inserção das anotações no projeto

Por fim, a última etapa da solução em questão, trata-se de uma etapa contínua no projeto, pois sempre que uma nova classe é mapeada no sistema ou um novo cadastro é criado, a mesma ocorrerá. Nesta etapa se faz necessária a adição das anotações nos códigos do projeto, onde se deve inserir a anotação em todas as classes e métodos que devem ser testados. O código 3.4 ilustra e exemplifica uma classe Java contendo anotações referentes à interface Teste.

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

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

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

    @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 3.4: Classe anotada com @Teste

Na anotação são inseridas informações referente às ações que devem ser executadas para a propriedade em questão. Os métodos possuem anotações referentes apenas ao campo que representam, enquanto as informações mais gerais sobre a página, que está sendo testada, são inseridas na anotação da classe propriamente dita.

### 3.2.4 Discussão sobre a solução

Após a finalização do desenvolvimento desta primeira solução encontrada, notou-se a clara necessidade de alteração na sistemática de como os testes 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 tem uma sequência bem mapeada que deve ser seguida.

Outro fator determinante para a alteração desta solução foi o fato de, ao se utilizar anotações, o mesmo não possibilitaria o teste de um mesmo campo com dois valores diferentes, pois só seria possível informar na anotação do método um único valor utilizado no teste. 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.

## 3.3 Solução para testes funcionais com Cucumber e Selenium

Visando a produção de testes funcionais, optou-se pela inclusão do *framework* Cucumber na composição desta nova solução, somando-se assim as demais ferramentas já utilizadas até o atual momento. A mesma foi escolhida por possibilitar a descrição de testes em forma de cenários. Um cenário pode ser vistos como uma composição de testes unitários que na primeira solução apresentada estariam dispersos. Sendo assim torna-se viável descrever os requisitos funcionais do sistema através de testes funcionais nos arquivos *.feature* do Cucumber, onde os mesmo poderão ser compostos um uma série de cenários. Outro ponto relevante para a sua escolha seria sua interface amigável, onde os cenários são escritos em uma linguagem bem próxima a linguagem natural, tornando-os de fácil entendimento por todos os envolvidos do projeto, garantindo assim uma maior qualidade também quando pensamos em documentação das funcionalidades desenvolvidas em um sistema.

A solução é composta por três estágios, onde em um primeiro será inserido no projeto a classe genérica de testes já descrita, porem dela só serão utilizados os métodos que descrevem ações do Selenium WebDriver. No segundo estágio ocorre a adição de uma classe, onde são apresentados métodos que representam *steps* da linguagem Cucumber o que possibilita descrever, no ultimo estágio, cenários no Cucumber em arquivos `.feature`, que descrevem testes para as funcionalidades codificadas no sistema. A imagem 3.5 apresenta esse fluxo.

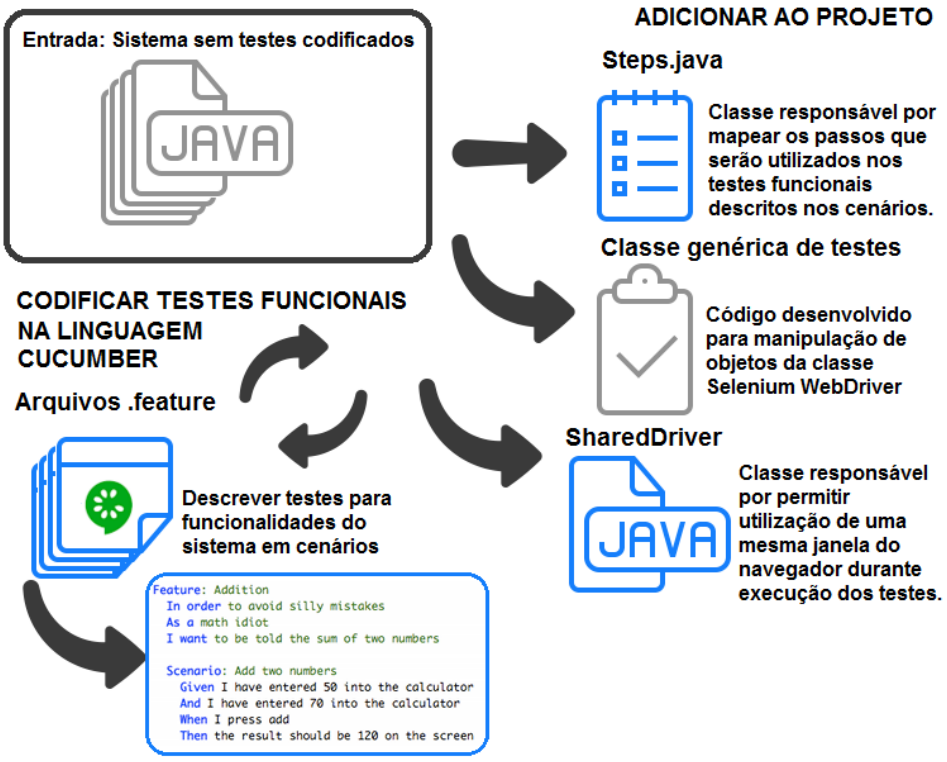


Figura 3.5: Estrutura da solução utilizando Selenium + Cucumber.

### 3.3.1 Classe genérica de testes

Na classe genérica de testes foram mantidos os métodos anteriormente utilizados que permitiam gerenciar ações do Selenium WebDriver, como apresenta a figura 3.6. Entretanto, foram removidos os métodos que realizavam os testes com base nas anotações inseridas nas classes Java do projeto.

### 3.3.2 Classe que descreve Steps

A classe `Steps.java` foi criada com intuito de realizar o elo de ligação entre as ações da classe WebDriver e frases utilizadas nos arquivos `.feature` da linguagem Cucumber. Nesta classe foram confeccionados métodos que mapeiam ações do Cucumber e, dentro dos mesmos, ocorrem as chamadas dos métodos da classe WebDriver como ilustra o código em 3.7.

A linguagem Cucumber quando trabalha conjuntamente com Java, disponibiliza anotações como: `@Dado, @Quando, @E e @Entao`, que corresponde de forma 1 para 1 frases descritas nos arquivos `.feature`. Como podemos ver na ilustração, cada método que corresponde a um *Step* do Cucumber apresenta uma anotação que identifica o mesmo, contendo uma frase fixa e parâmetros que são esperados. Dessa forma, podem-se descrever frases que

```
public Object getValorPropriedadeCampo(WebElement webElement, String atributo, String
    valorAtributo) {
      if (atributo.equals(ATRIBUTO_COMPARACAO_ASSERT_TEXTO)) {
          return webElement.getText();
      } else if (atributo.equals(ATRIBUTO_COMPARACAO_ASSERT_NOME_TAG)) {
          return webElement.getTagName();
      } else if (atributo.equals(ATRIBUTO_COMPARACAO_ASSERT_ATRIBUTO)) {
          return webElement.getAttribute(valorAtributo);
      } else if (atributo.equals(ATRIBUTO_COMPARACAO_ASSERT_VALOR_CSS)) {
          return webElement.getCssValue(valorAtributo);
      } else if (atributo.equals(ATRIBUTO_COMPARACAO_ASSERT_CAMPO_EXIBIDO)) {
          return webElement.isDisplayed();
      } else if (atributo.equals(ATRIBUTO_COMPARACAO_ASSERT_CAMPO_SELECIONADO)) {
          return webElement.isSelected();
      } else if (atributo.equals(ATRIBUTO_COMPARACAO_ASSERT_CAMPO_HABILITADO)) {
          return webElement.isEnabled();
      } else if (atributo.equals(ATRIBUTO_COMPARACAO_URL)) {
          return webDriver.getCurrentUrl();
      } else if (atributo.equals(ATRIBUTO_COMPARACAO_TITULO_PAGINA)) {
          return webDriver.getTitle();
      }
      return null;
  }

  public WebElement getEncontraCampo(String identificador, String campo) {
      return webDriver.findElement(getEncontra(identificador, campo));
  }

  public By getEncontra(String identificador, String campo) {
      if (identificador.equals(IDENTIFICADOR_ID)) {
          return By.id(campo);
      } else if (identificador.equals(IDENTIFICADOR_NOME)) {
          return By.name(campo);
      } else if (identificador.equals(IDENTIFICADOR_NOME_CLASSE)) {
          return By.className(campo);
      } else if (identificador.equals(IDENTIFICADOR_CSS)) {
          return By.cssSelector(campo);
      } else if (identificador.equals(IDENTIFICADOR_XPATH)) {
          return By.xpath(campo);
      } else if (identificador.equals(IDENTIFICADOR_TEXTO_DO_LINK)) {
          return By.linkText(campo);
      } else if (identificador.equals(IDENTIFICADOR_NOME_TAG)) {
          return By.tagName(campo);
      } return null;
  }
```

Figura 3.6: Métodos para busca de elemento web utilizando Selenium WebDriver

correspondam a pequenas ações executadas em um navegador. Assim, cada método é responsável por executar uma ação específica, tornando possível reutilizar estes métodos para compor os cenários que posteriormente serão descritos.

Para tornar a solução mais completa, inseriu-se nessa classe, métodos que permitissem manipular campos que dependessem de chamadas Ajax e JavaScript, como demonstra a imagem 3.8

Essas ações são possíveis através da utilização dos métodos oferecidos pela classe `WebDriverWait` do Selenium.

```
    @Dado("^atribuir timeout navegador (\\d+)$")
    public void setTimeoutNavegador(int timeout) {
        webDriver.manage().timeouts().implicitlyWait(timeout, TimeUnit.SECONDS);
    }

    @Dado("^acesso o endereco (.*)$")
    public void acessarEndereco(String url) {
        webDriver.get(TestePropriedades.urlSistema + url);
    }

    @Quando("^seleciono a opcao (.*) no campo (.*) buscando pelo (.*)$")
    public void selecionarOpcaoCampo(String opcao, String campo, String identificador) {
        WebElement webElement = getEncontraCampo(identificador, campo);
        if (webElement != null) {
            Select select = new Select(webElement);
            select.selectByVisibleText(opcao);
        }
    }

    @E("^limpo o campo (.*) buscando pelo (.*)$")
    public void limparCampo(String campo, String identificador) {
        WebElement webElement = getEncontraCampo(identificador, campo);
        if (webElement != null) {
            webElement.clear();
        }
    }

    @E("^preencho o campo (.*) com o valor (.*) buscando pelo (.*)$")
    public void preencherCampo(String campo, String valor, String identificador) {
        WebElement webElement = getEncontraCampo(identificador, campo);
        if (webElement != null) {
            webElement.sendKeys(valor);
        }
    }

    @E("^clico no elemento (.*) buscando pelo (.*)$")
    public void clicarElemento(String campo, String identificador) {
        WebElement webElement = getEncontraCampo(identificador, campo);
        if (webElement != null) {
            webElement.click();
        }
    }

    @E("^submeto o elemento (.*) buscado pelo (.*)$")
    public WebElement submeterElemento(String campo, String identificador) {
        WebElement webElement = funcoes.getEncontraCampo(identificador, campo);
        if (webElement != null) {
            webElement.submit();
        }
        return webElement;
    }
```

Figura 3.7: Classe genérica contendo steps Cucumber que descrevem ações do Selenium WebDriver

### 3.3.3 Descrição de cenários

Após a inserção das duas classes Java apresentados anteriormente no projeto, torna-se possível escrever cenários que representem as funcionalidades que devem ser testadas. Desta forma, pode-se criar arquivos `.feature` em que os cenários são compostos por uma sequência definida utilizando os passos descritos na classe `Steps`, onde a mesma realiza as ações em um navegador. Podemos ver um exemplo desta abordagem no código apresentado em 3.9.

```
@E("^aguardo (\\d+) milisegundos para verificar se elemento (.*) buscando pelo (.*) esta
    presente$")
public void aguardarCampoExistente(Long tempo, String campo, String identificador) {
    WebDriverWait webDriverWait = new WebDriverWait(webDriver, tempo);
    webDriverWait.until(ExpectedConditions.presenceOfElementLocated(
        funcoes.getEncontra(identificador, campo)));
}

@E("^aguardo (\\d+) milisegundos para verificar se elemento (.*) buscando pelo (.*) esta
    visivel$")
public void aguardarCampoVisivel(Long tempo, String campo, String identificador) {
    WebDriverWait webDriverWait = new WebDriverWait(webDriver, tempo);
    webDriverWait.until(ExpectedConditions.visibilityOf(funcoes.getEncontraCampo(identificador,
        campo)));
}

@E("^aguardo (\\d+) milisegundos para verificar se elemento (.*) buscando pelo (.*)
    desapareceu$")
public void aguardarCampoDesaparecer(Long tempo, String campo, String identificador) {
    WebDriverWait webDriverWait = new WebDriverWait(webDriver, tempo);
    webDriverWait.until(ExpectedConditions.invisibilityOfElementLocated(
        funcoes.getEncontra(identificador, campo)));
}
```

Figura 3.8: Método que permite manipulações com espera de tempo ou ação

```
#language: pt

Funcionalidade: Ações executadas por um aluno do curso de Ciência da Computação no registro
    de seu
  plano individual de estudos complementares.

  Contexto: Fazer login no sistema como um aluno
    Dado acesso o endereco /login.htm
    Quando preencho o campo login com o valor lamaral buscando pelo id
    E preencho o campo senha com o valor teste123 buscando pelo id
    E clico no elemento button.btn.btn-default buscando pelo css
    Entao verifico se atributo nome tag do elemento id buscando pelo nome não está nulo

  @aceitacao
  Cenario: Adicionar umas disciplina ao plano com sucesso.
    Dado seleciono a opcao ELC1051 - Computação Gráfica Avançada (60h) no campo
        idDisciplinaAdicionar buscando pelo id
    Quando preencho o campo piecDisciplinaAdicionar.cursoOfertante com o valor Ciência da
        Computação buscando pelo id
    E preencho o campo piecDisciplinaAdicionar.semestreAnoRealizacao com o valor II/2011
        buscando pelo id
    E clico no elemento adicionarPiecDisciplina buscando pelo id
    Entao comparo a igualdade entre o valor esperado Sucesso! com atributo texto do elemento
        h4 buscando pelo css

  @rejeicao
  Cenario: Não permitir adicionar disciplina ao plano quando a mesma já está incluida.
    Dado seleciono a opcao ELC1051 - Computação Gráfica Avançada (60h) no campo
        idDisciplinaAdicionar buscando pelo id
    Quando preencho o campo piecDisciplinaAdicionar.cursoOfertante com o valor Ciência da
        Computação buscando pelo id
    E preencho o campo piecDisciplinaAdicionar.semestreAnoRealizacao com o valor II/2011
        buscando pelo id
    E clico no elemento adicionarPiecDisciplina buscando pelo id
    Entao comparo a igualdade entre o valor esperado Disciplina já inserida no plano. com
        atributo texto do elemento piec.errors buscando pelo id
```

Figura 3.9: Testes de uma funcionalidade descrita por seus cenários em um arquivo .feature

Como visto em 3.9, basta utilizar os passos previamente definidos nos métodos da classe `Steps.java` complementando apenas com os parâmetros necessários para identificação do

elemento que se deseja manipular em uma tela do sistema web.

### 3.3.4 Utilizando a mesma janela do navegador para execução dos testes

Ao iniciar os testes na solução, após finalizar a codificação da mesma, notou-se que existia uma perda de tempo quando os testes eram executados, pois, a cada cenário executado o Selenium WebDriver fechava a janela do navegador ativa e, quando um novo cenário fosse executado, uma nova janela era aberta. Para evitar que este retrabalho fosse realizado e, assim, fosse possível reutilizar a janela do navegador ativa, encontrou-se na documentação existente na página da linguagem Cucumber[2] a solução para esta questão, utilizando a classe `SharedDriver` desenvolvida por Aslak Hellesoy e disponibilizada no projeto `cucumber-jvm`[3], onde deve-se instanciar um objeto desta classe no lugar da classe WebDriver, utilizada anteriormente.

### 3.3.5 Validando solução em sistemas web

Após a finalização da codificação da solução, iniciou-se a fase de validação e aplicação da solução utilizando Cucumber + Selenium em softwares web. Nesta etapa foram escolhidos dois softwares web, que respeitassem os requisitos existentes no escopo delimitado pelo projeto em questão. O primeiro sistema escolhido foi um sistema desenvolvido para efetuar a solicitação de disciplinas complementares de graduação, do curso de Ciência da Computação, da Universidade Federal de Santa Maria. O Segundo sistema, em que os testes funcionais seriam aplicados, seria um software de uma empresa privada, da região de Santa Maria, chamada Megatecnologia. Este sistema comercializado como SILAS BPMs, apresenta um sistema voltado para modelagem de processo BPMs, no qual é possível descrever etapas, tramitações, telas e campos de um processo.

#### 3.3.5.1 Aplicando solução no sistema de registro de disciplinas complementares

Para realização desta tarefa, foram utilizadas as funcionalidades descritas no documento de especificação deste software para validar se o mesmo apresenta todas as funcionalidades que se compromete a atender. Como o processo apresenta dois atores principais, onde o primeiro seria o aluno e, o segundo, um membro do colegiado do curso, foram separados os testes em dois arquivos `.feature`. O primeiro contendo ações realizadas por alunos e, o segundo, para

[2] https://cucumber.io/

[3] https://github.com/cucumber/cucumber-jvm/blob/master/examples/java-webbit-websockets-selenium/src/test/java/cucumber/examples/java/websockets/SharedDriver.java

ações que um membro do colegiado executa.

Como forma de reduzir o trabalho de um desenvolvedor de teste, utilizando os recursos disponibilizados pela linguagem Cucumber, pode-se utilizar o recurso `Contexto` para descrever uma sequência de passos que deverá ser executada para cada um dos cenários de uma funcionalidade. Assim, evita-se de ter que reescrever a mesma sequência de passos para todos os cenários. A imagem 3.10 exemplifica essa utilização.

```
Funcionalidade: Ações executadas por um membro do colegiado do curso de Ciência da Computação
    na aprovação/regeição de planos individuais de estudos complementares.

  Contexto: Fazer login no sistema como um membro do colegiado
    Dado acesso o endereco /login.htm
    Quando preencho o campo login com o valor colegiado buscando pelo id
    E preencho o campo senha com o valor colegiado123 buscando pelo id
    E clico no elemento button.btn.btn-default buscando pelo css
    Entao verifico se atributo nome tag do elemento id buscando pelo nome não está nulo
```

Figura 3.10: Exemplo de utilização do Contexto em um arquivo Cucumber

O `Contexto` deve ser descrito no inicio do código de uma funcionalidade, antes dos cenários do mesmo.

Outro recurso da linguagem Cucumber útil para redução na escrita de novos testes, é a `Delineacao do Cenario`, que permite reutilizar o mesmo cenário, aplicando os valores contidos em uma tabela descrita pelo recurso `Exemplos` como mostra o código 3.11

Em `Delineacao do Cenario` ao invés de ser inseridos valores para testes, são informados os rótulos existentes no cabeçalho da tabela do recurso `Exemplos`. Desta forma, quando o teste for ser executado, esse valor será substituído, em tempo de execução, pelo valor existente nessa coluna para cada uma das linhas da tabela.

Utilizando-se destes recursos concluiu-se, de forma satisfatória, a codificação de cenários de testes para todos os requisitos existentes neste sistema.

#### 3.3.5.2 Aplicando solução no sistema SILAS BPMs

Para o software em questão, por tratar-se de um sistema que possui mais de oito anos de desenvolvimento e por não possuir nenhum tipo de teste automatizado codificado, optou-se por retirar, do *task bord* onde a equipe de desenvolvimento, da empresa em questão, gerenciava tarefas a desenvolver e desenvolvidas, as ultimas funcionalidades finalizadas para descrevê-las como cenários e, assim realizar os primeiros testes. Logo no início do mapeamento dos testes, notaram-se necessidades que o sistema de registros de disciplinas complementares não

```
Delineacao do Cenario: 1 - Não permitir inserir no PIEC disciplina que não faça parte do
    curso, sem o preenchimento do campo relevância da integralização
     2 - Não permitir cadastrar nova instituição com sigla já cadastrada
     3 - Não permitir cadastrar nova disciplina sem sua respectiva sigla
     4 - Não permitir cadastrar nova disciplina sem sua respectiva nome
     5 - Não permitir inserir disciplina com uma sigla já cadastrada
    Dado seleciono a opcao Adicionar outra disciplina no campo idDisciplinaAdicionar buscando
        pelo id
    Quando preencho o campo novaDisciplina.codigo com o valor <codigoDis> buscando pelo id
    E preencho o campo novaDisciplina.nome com o valor <nomeNovaDisciplina> buscando pelo id
    E preencho o campo piecDisciplinaAdicionar.relevanciaIntegralizacao com o valor
        <relevancia> buscando pelo id
    E seleciono a opcao Adicionar outra instituição no campo novaDisciplina.idInstituicao
        buscando pelo id
    E seleciono a opcao 78 horas no campo novaDisciplina.cargaHoraria buscando pelo id
    E preencho o campo novaInstituicao.nome com o valor Universidade Teste buscando pelo id
    E preencho o campo novaInstituicao.sigla com o valor <siglaInst> buscando pelo id
    E preencho o campo piecDisciplinaAdicionar.arquivoPlanoEnsino com o valor
        C:\\Users\\Lucas\\Desktop\\putty.exe buscando pelo id
    E clico no elemento adicionarPiecDisciplina buscando pelo id
    Entao comparo a igualdade entre o valor esperado <msgErro> com atributo texto do elemento
        piec.errors buscando pelo id

  Exemplos:
    | codigoDis | nomeNovaDisciplina | relevancia | siglaInst | msgErro                     |
    | TESTE95   | Teste informática  |            | USC       | Preencha o campo relevância.|
    | TESTE95   | Teste informática  | Teste 1    | UFSM      | Sigla já cadastrada         |
    |           | Teste informática  | Teste 2    | UFT       | Preencha o campo código.    |
    | TESTE98   |                    | Teste 3    | UFS       | Preencha o campo nome.      |
    | ELC1051   | Computação Gráfica | Teste 4    | UFAR      | Código já cadastrado.       |
```

Figura 3.11: Cenário executado para cada linha existente na tabela Exemplos

possuía. Como, por exemplo, a necessidade de guardar valores de um campo, para após submeter alguma função, comparar com o novo valor do campo em questão. Também percebeu-se a indispensabilidade de permitir ações condicionais nos testes. Essa necessidade se deu pelo fato de, em alguns casos, não se conhecer o estado atual de algum elemento na tela em questão e, desta forma, o teste poderia falhar mesmo quando não existisse problema. Após algumas tentativas de utilizar a solução proposta para esses novos cenários, notou-se que seria inviável trabalhar sem que essas questões fossem viabilizadas.

### 3.3.6 Considerações sobre a solução Cucumber + Selenium

Com o avanço na confecção de testes aplicados em sistemas reais, percebeu-se uma série de vantagens e desvantagens na utilização desta solução no processo de desenvolvimento e testes de funcionalidades. Como vantagens, podemos elencar o fato de, ao escrever cenários utilizando o IDE que tenha suporte para a linguagem Cucumber, pouco se escreve, reutilizando sempre os *steps* descritos na classe `Steps.java`. Sendo, assim, necessário apenas informar os parâmetros esperados pelos mesmos. Ainda destaca-se o fato de os cenários descreverem, de forma simples, o que deve ocorrer na funcionalidade testada e, desta forma, serve também

como documentação. Quando pensamos nos pontos fracos da solução, podemos destacar os empecilhos detectados no desenvolvimento de testes para a aplicação SILAS BPMs. Somam-se a estes, algumas limitações que a linguagem Cucumber possui, como o fato de não ser possível reutilizar um cenário dentro de outro cenário[4] ou até mesmo utilizar como parâmetro valores que diferem dos tipos comuns da linguagem Java (int, float, String). Outro fator problemático nessa solução é o fato de todos os dados utilizados nos testes serem persistidos e, com isso, ser necessário a restauração manual do ultimo estado válido da base de dados onde os testes se realizam.

## 3.4 Solução para testes funcionais com Selenium WebDriver

Como forma de possibilitar a utilização de testes condicionais, armazenamento de valores e manipulação de objetos Java dentro do testes funcionais, optou-se pela confecção de uma nova abordagem neste projeto, mesmo que para isso se tivesse que abdicar da praticidade e interface amigável do Cucumber. Assim, os testes funcionais seriam descritos diretamente em classes de teste Java. A imagem 3.12 apresenta essa pequena alteração na abordagem dos testes proposta.

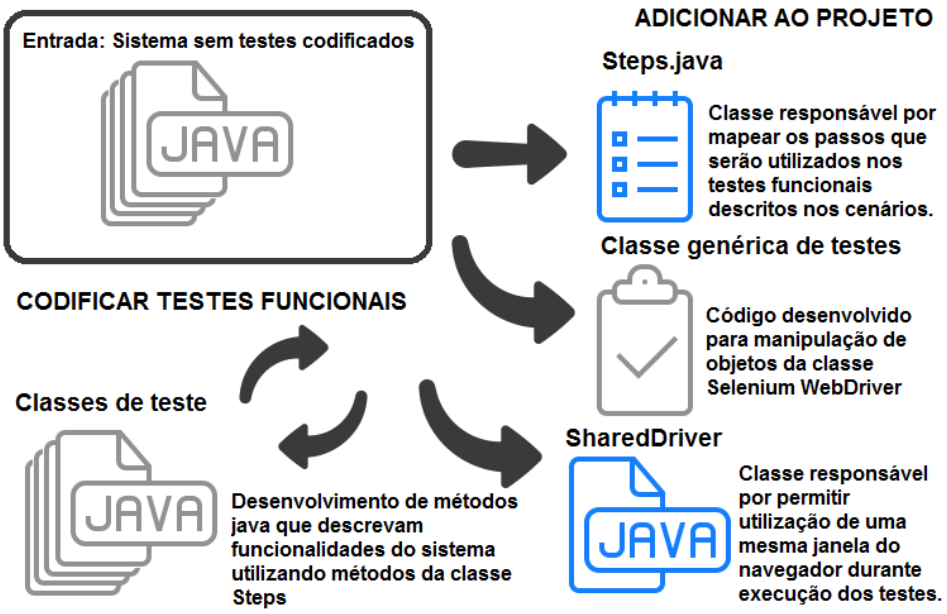


Figura 3.12: Estrutura da solução utilizando somente Selenium.

[4] Essa funcionalidade estava disponível na primeira versão da linguagem, mas a mesma foi removida. Embora o exemplo pode fazer sentido quando se trabalha de forma sequencial através dos cenários a longo prazo produz-se cenários que são difíceis de manter e entender (CUCUMBER WAVES GOODBYE TO GIVENSCENARIO, 2009).

### 3.4.1 Classes Java contendo testes funcionais

Como descrito na imagem 3.12, esta abordagem reutiliza praticamente toda a estrutura existente na abordagem Cucumber + Selenium, com isso, somente os arquivos onde são escritos os cenários mudam. Ao invés de criarmos arquivos `.feature`, contento as funcionalidades e cenários que serão testados, tem-se, agora, classes Java de teste, que são responsáveis por descrever os testes. Na figura 3.13 temos um exemplo de um cenário descrito utilizando a abordagem atual. Pode-se notar que os métodos da classe `Steps.java` ainda são utilizados, porém de forma direta.

```
//Quando um agrupamento é fechado, o sistema deve alterar a data de fechamento do mesmo
    para a data atual. No caso da caixa atual estar aberta
private void abreFechaAgrupamento() {
  navegador.clicarElemento("img[alt=\"Visualizar\"]", Steps.IDENTIFICADOR_CSS);
  navegador.aguardarCampoVisivel(100l, "img[alt=\"F\"]", Steps.IDENTIFICADOR_CSS);
  WebElement dataAbertura = navegador.getEncontraCampo("data_abertura_agrupamento_1");
  WebElement dataFechamento = navegador.getEncontraCampo("data_fechamento_agrupamento_1");
  boolean aberto = true;
  if (dataAbertura.getText() != null) {
    aberto = false;
  }
  navegador.clicarElemento("img[alt=\"F\"]");
  if (aberto) {
    assertEquals(SilasFuncoesDatas.sdf.format(new Date()), dataFechamento.getText());
  } else {
    assertEquals(SilasFuncoesDatas.sdf.format(new Date()), dataAbertura.getText());
  }
}
```

Figura 3.13: Teste funcional utilizando abordagem Selenium WebDriver

Por tratar-se de código Java, além de tornar possível utilizar testes condicionais e armazenamento de valores, novas funções tornaram-se plausíveis, como apresentado no código 3.14, onde realiza-se comparações com uma data utilizando um formato específico.

```
//Quando navagamos pelos agrupamentos, verificamos que as datas de abertura das caixas não
    são necessariamente as mesmas.
private void navegarEntreAgrupamentos() {
  navegador.clicarElemento("imagem_anterior_agrupamento_1");
  assertNotEquals(SilasFuncoesDatas.sdf.format(new Date()),
      navegador.getEncontraCampo("data_abertura_agrupamento_1").getText());
}
```

Figura 3.14: Realizando comparações utilizando o tipo de dados data.

Pode-se, também, localizar valores que não são estáticos na busca de elementos HTML existentes na tela atual de manipulação, como demonstra o trecho de código 3.15.

```
  @Test
  //Colocar datas ao lado da indentificação do agrupamento (bateria), para assim contemplar
      as necessidades da embrapa acre
  public void datasInicioFimBateria() {
    navegador.acessarEndereco(url + "/gerenciar-agrupamentos.htm");
    String idAgrupamento = navegador.getEncontraCampo(Steps.IDENTIFICADOR_ID, "id").getText();
    Agrupamento agrupamento = service.getAgrupamento(Long.valueOf(idAgrupamento));
    navegador.getEncontraCampo("data_abertura_agrupamento_" + agrupamento.getId());
    navegarEntreAgrupamentos();
  }
```

Figura 3.15: Realizando comparações utilizando o tipo de dados data.

### 3.4.2 Validando solução em sistemas web

Para validar esta solução, foram utilizados os mesmos dois projetos já usados na solução anterior.

#### 3.4.2.1 Aplicando solução no sistema de registro de disciplinas complementares

Como ocorrido com a solução anterior, essa solução viabiliza a codificação de testes para todos os requisitos funcionais deste software. Ilustração 3.16 trás a codificação do mesmo teste executado na imagem 3.10.

```
@Test
/*Fazer login no sistema como um membro do colegiado*/
public void getLoginMembroColegiado() {
    steps.acessarEndereco("/login.htm");
    steps.preencherCampo("login", "colegiado", TesteFormulario.IDENTIFICADOR_ID);
    steps.preencherCampo("senha", "colegiado123", TesteFormulario.IDENTIFICADOR_ID);
    steps.clicarElemento("button.btn-default", TesteFormulario.IDENTIFICADOR_CSS);
    steps.compararSeNaoNulo(TesteFormulario.ATRIBUTO_COMPARACAO_ASSERT_NOME_TAG, "id",
        TesteFormulario.IDENTIFICADOR_NOME);
}
```

Figura 3.16: Cenário descrevendo mesma funcionalidade contida em 3.10

#### 3.4.2.2 Aplicando solução no sistema SILAS BPMs

Ao contrário do ocorrido com a aplicação de registro de disciplinas complementares do curso de Ciência da Computação da Universidade Federal de Santa Maria (UFSM), a solução que mescla a utilização da linguagem Cucumber juntamente com o *framework* Selenium WebDriver não contemplava, na totalidade, os testes necessários. Agora utilizando essa nova abordagem, tornou-se viável a codificação dos testes necessários e, com isso, optou-se, na empresa Megatecnologia, pela continuidade da realização de testes através desta solução.

### 3.4.3 Considerações sobre a solução Selenium WebDriver

A abordagem em questão, assim como as anteriores, apresenta vantagens e desvantagens. Como vantagem, pode-se elencar o fato de, por trabalhar somente com a linguagem Java, possibilitar uma série de ações, que não seriam possíveis, utilizando Cucumber na composição da solução. Como por exemplo, a utilização de testes condicionais, laços de repetição, busca de valores de um objeto Java, assim como manipulação de tipos que não façam parte dos tipos nativos da linguagem Java. Apesar disso, a solução apresenta um cenário mais complexo de ser entendido por qualquer interessado do projeto, comparado a solução utilizando Cucumber. Apresenta também, descrições mais poluídas visualmente e mais distantes de uma linguagem natural, além de possuir o mesmo problema referente à restauração da base de dados, após execução dos testes.

# 4 RESULTADOS

Como forma de validação das abordagens criadas neste trabalho, onde as mesmas visam a redução do trabalho para uma equipe de teste e desenvolvimento, realizou-se comparações que serviriam para levantar o ganho em utilizar uma abordagem ou outra.

## 4.1 Comparando abordagens funcionais

Mesmo conhecendo as principais virtudes das abordagens criadas, e também em quais escopos as mesmas funcionavam de forma adequada, ainda era necessário expor o ganho em utilizar umas dessas abordagens ao invés de criar um teste funcional somente utilizando métodos que as ferramentas de teste web disponibilizam.

Como forma de medir este ganho, resolveu-se descrever uma funcionalidade do sistema de registro de disciplina complementares, utilizando tanto a abordagem Cucumber + WebDriver, como a abordagem WebDriver e, por fim, criando um teste utilizando apenas os métodos disponíveis da ferramenta Selenium. Essas diferenças são apresentadas nas imagens 4.1, 4.2 e 4.3.

```
@Test
public void cadastrarDisciplinaSucesso() {
    WebDriver webDriver = new FirefoxDriver();
    webDriver.get(url + "/cadastro-disciplina.htm");
    webDriver.findElement(By.id("codigo")).sendKeys("ELC9898");
    webDriver.findElement(By.id("nome")).sendKeys("Disciplina nova");
    Select cargaHoraria = new Select(webDriver.findElement(By.id("cargaHoraria")));
    cargaHoraria.selectByVisibleText("60");
    webDriver.findElement(By.id("ativa1")).click();
    Select instituicao = new Select(webDriver.findElement(By.id("idInstituicao")));
    instituicao.selectByVisibleText("UFSM - Universidade Federal de Santa Maria");
    webDriver.findElement(By.id("preAprovada1")).click();
    webDriver.findElement(By.id("salvar")).click();
    assertEquals("Sucesso!", webDriver.findElement(By.cssSelector("h4")).getText());
}
```

Figura 4.1: Teste utilizando apenas o Selenium

Como pode-se notar pelo código 4.1, 4.2 e 4.3, existe uma diminuição no número de linhas escritas nos testes que exploram as duas abordagens deste trabalho em relação a codificação utilizando apenas o framework Selenium. Durante a realização dessas comparações percebeu-se também uma possível redução de caracteres digitados utilizando as abordagens apresentadas, isso quando trabalha-se com um IDE que possua suporte para as linguagens em questão. Por fim pode-se também levantar suspeita que a abordagem Cucumber + WebDriver trás ganhos na

```
Cenario: Cadastrar nova disciplina.
    Dado acesso o endereco /cadastro-disciplina.htm
    Quando preencho o campo codigo com o valor ELC9898 buscando pelo id
    E preencho o campo nome com o valor Disciplina nova buscando pelo id
    E seleciono a opcao 60 no campo cargaHoraria buscando pelo id
    E clico no elemento ativa1 buscando pelo id
    E seleciono a opcao UFSM - Universidade Federal de Santa Maria no campo idInstituicao
        buscando pelo id
    E clico no elemento preAprovada1 buscando pelo id
    E clico no elemento salvar buscando pelo id
    Entao comparo a igualdade entre o valor esperado Sucesso! com atributo texto do elemento
        h4 buscando pelo css
```

Figura 4.2: Teste utilizando solução Cucumber + WebDriver

```
@Test
//Cadastrar nova disciplina
public void cadastrarDisciplina() {
    steps.acessarEndereco("/cadastro-disciplina.htm");
    steps.preencherCampo("codigo", "ELC9898", TesteFormulario.IDENTIFICADOR_ID);
    steps.preencherCampo("nome", "Disciplina nova", TesteFormulario.IDENTIFICADOR_ID);
    steps.selecionarOpcaoCampo("60", "cargaHoraria", TesteFormulario.IDENTIFICADOR_ID);
    steps.clicarElemento("ativa1");
    steps.selecionarOpcaoCampo("UFSM - Universidade Federal de Santa Maria", "idInstituicao",
        TesteFormulario.IDENTIFICADOR_ID);
    steps.clicarElemento("preAprovada1");
    steps.clicarElemento("salvar");
    steps.compararIgualdade("Sucesso!", TesteFormulario.ATRIBUTO_COMPARACAO_ASSERT_TEXTO,
        "h4", TesteFormulario.IDENTIFICADOR_CSS);
}
```

Figura 4.3: Teste utilizando solução WebDriver

leitura dos testes, que se tornam mais compreensíveis, mesmo que não se entenda de linguagens de programação.

## 4.2 Comparando as três soluções propostas

Visando mapear de forma mais externa as abordagens desenvolvidas, confrontou-se as soluções propostas elencando suas respectivas virtudes e limitações e, com isso torna-se possível realizar escolhas sobre qual abordagem é mais apropriada para um cenário específico que se apresente. Por exemplo, quando a equipe decide que serão necessários apenas a realização de testes unitários, a abordagem para testes unitários que utiliza anotações apresenta-se como uma boa alternativa, pois a mesma possui recursos que permitem realizar este tipo de teste e ao mesmo tempo não necessita de codificação em classes de teste, pois a simples inserção de anotações nas classes do modelo é suficiente.

Quando deseja-se realizar testes funcionais e não se faz necessário utilizar verificações condicionais, adotar a segunda solução torna-se uma opção válida. A utilização do Cucumber, neste casos, torna-se prática e ainda possibilita que qualquer *stakeholder* do projeto entenda

as funcionalidades descritas pelos testes. A ultima abordagem pode ser apropriada quando o sistema em questão é mais complexo e necessita de maiores validações.

A tabela 4.1 apresenta os principais pontos fortes e as maiores fraquezas dessas três abordagens.

| | Unitário | Cucumber e Selenium | Funcional Selenium |
|---|---|---|---|
| Testes unitários | ✓ | ✓ | ✓ |
| Testes funcionais | | ✓ | ✓ |
| Elementos que dependam de funções Ajax e Javascript | | ✓ | ✓ |
| Cenários com escrita de fácil compreensão | | ✓ | |
| Realização de testes dependentes de condições | | | ✓ |
| Armazenar valor de um campo para utiliza-lo posteriormente | | | ✓ |
| Possibilita utilizar parâmetros que diferem dos tipos básicos (int, float, String) | ✓ | | ✓ |
| Retroceder o banco de dados ao último estado válido após realizar os testes | | | |
| Utilização de cenários dentro de outro cenário | | | ✓ |

Tabela 4.1: Vantagens e desvantagens das abordagens de teste confeccionadas

# 5 CONCLUSÃO

A proposta inicial deste trabalho consistia em apresentar uma solução em forma de códigos de teste que permitisse o reaproveitamento e reutilização dos mesmo, acarretando em um menor esforço na codificação de novos testes para sistemas web desenvolvidos em Java. Entretanto o resultado final obtido pelo trabalho em questão trata-se não de uma solução, mas sim de três abordagens utilizando ferramentes para teste de software que diferem entre si, cada uma possuindo ponto fortes e pontos fracos e sendo mais relevante para uma determinada situação. A condução e criação dessas abordagens foram tomadas ao longo do desenvolvimento a partir dos percalços e necessidades que apareciam em meio a validação das abordagens em questão.

Como contribuição este trabalho apresenta uma abordagem que mescla a utilização da linguagem Cucumber com o framework Selenium Webdriver, apresentando passos que podem tornar o processo de criação de testes em uma tarefa mais prática e rápida, sendo assim necessário apenas descrever cenários que correspondem a funcionalidades do sistema.

A lista a seguir expõem, em ordem de relevância, algumas melhorias que poderiam ser incorporadas em uma nova versão deste projeto:

- Apresentar no auto completar do IDE, os campos pertencentes ao contexto atual do .feature: Ao utilizar um IDE que possibilite manipulação de plugins, desenvolver mecanismo que pesquise nos arquivos JSP/JSF todos os campos referentes à um contexto atual, e assim apresente sugestão de alto completar para os cenários que estão sendo descritos pelo Cucumber.

- Não persistir no BD as alterações realizadas pelos testes automatizados: Possibilitar que ao finalizar execução de um teste funcional, o bando de dados utilizado retorne ao estado inicial, não persistindo assim as alterações realizadas pelos testes executados.

- Criar mecanismo que busque na árvore do projeto classes de validação dos formulários e transcreva essas informações em cenários de teste Cucumber: Criar mecanismo que permita gerar testes a partir das validações que são inseridas nas classes de validação referentes a uma classe de controle.

# REFERÊNCIAS

BIASI, L. B. Geração automatizada de drivers e stubs de teste para JUnit a partir de especificações U2TP. , [S.l.], 2006.

BOEHM, B. W.; BROWN, J. R.; LIPOW, M. Quantitative evaluation of software quality. In: SOFTWARE ENGINEERING, 2. **Proceedings...** [S.l.: s.n.], 1976. p.592–605.

CAGNIN, M. I. et al. Reuso na Atividade de Teste para Reduzir Custo e Esforço de VV&T no Desenvolvimento e na Reengenharia de Software. **XVIII Simpósio Brasileiro de Engenharia de Software (SBES)**, [S.l.], p.71–84, 2004.

CHIAVEGATTO[1], R. B. et al. Desenvolvimento Orientado a Comportamento com Testes Automatizados utilizando JBehave e Selenium. , [S.l.], 2013.

CRESPO, A. N. et al. Uma metodologia para teste de Software no Contexto da Melhoria de Processo. **Simpósio Brasileiro de Qualidade de Software**, [S.l.], p.271–285, 2004.

CUCUMBER Waves Goodbye to GivenScenario. Accessed: 2015-11-24, http://blog.josephwilk.net/ruby/cucumber-waves-goodbye-to-givenscenario.html.

FANTINATO, M. et al. AutoTest–Um framework reutilizável para a automação de teste funcional de software. **Cad. CPqD Tecnologia**, [S.l.], v.1, n.1, p.119–131, 2005.

FIDELIS, W. I. O.; MARTINS, E. FireWeb: uma ferramenta de suporte aos testes de regressão de aplicações web. **18th SBES–11th Tools Session**, [S.l.], p.1–6, 2004.

GARVIN, D. A. Competing on the 8 dimensions of quality. **Harvard business review**, [S.l.], v.65, n.6, p.101–109, 1987.

GUIZZARDI, G. **Desenvolvimento para e com reuso**: um estudo de caso no domínio de vídeo sob demanda. 2000. Tese (Doutorado em Ciência da Computação) — Universidade Federal do Espírito Santo.

JOMORI, S. M.; VOLPE, R.; ZABEU, A. C. P. Qualidade de software. **Revista TECHOJE,(IETEC-Instituto de Educação Tecnológica), ano XIII. Dispo-**

**nível em:< http://www. ietec. com. br/ietec/techoje/techoje/tecnologiadainformacao/2004/07/01/2004_07_01_0001. 2xt/-template_ interna**, [S.l.], 2004.

KARINSALO, M.; ABRAHAMSSON, P. Software reuse and the test development process: a combined approach. In: **Software Reuse**: methods, techniques, and tools. [S.l.]: Springer, 2004. p.59–68.

KOSCIANSKI, A.; SOARES, M. d. S. **Qualidade de software**. [S.l.]: São Paulo: Novatec Editora, 2007.

LOPES, D. **O Cucumber ainda tem o seu valor**. Accessed: 2015-08-10, http://www.infoq.com/br/articles/valor-cucumber.

MALDONADO, J. C. et al. Introduçao ao teste de software. **São Carlos**, [S.l.], 2004.

MYERS, G. J.; SANDLER, C.; BADGETT, T. **The art of software testing**. [S.l.]: John Wiley & Sons, 2011.

NUNES, D. Automação de Testes de Aceitação com Cucumber e JRuby. , [S.l.], 2009.

OLIVEIRA, R. B. de et al. Utilização de Padrões para Otimizar a Automação de Testes Funcionais de Software. , [S.l.], 2007.

PATEL, S.; KOLLANA, R. K. Test Case Reuse in Enterprise Software Implementation–An Experience Report. In: SOFTWARE TESTING, VERIFICATION AND VALIDATION (ICST), 2014 IEEE SEVENTH INTERNATIONAL CONFERENCE ON. **Anais...** [S.l.: s.n.], 2014. p.99–102.

PEREIRA, D. V. Estudo da ferramenta selenium ide para testes automatizados de aplicações web. , [S.l.], 2012.

PIQUEIRO, T. J. C.; ZADRA, A. N. Processo de Testes: abordagem de projeto de testes. , [S.l.], January 2015.

PRESSMAN, R. S. **Engenharia de software**. [S.l.]: McGraw Hill Brasil, 2011.

REFERENCE Cucumber.io. Accessed: 2015-08-05, https://cucumber.io/docs/reference.

RIOS, E. **Teste de software**. [S.l.]: Alta Books Editora, 2006.

SCHMITZ, F. H.; BECKER, H.; BERLATTO, L. d. F. **Cucumber - Um breve review**. Accessed: 2015-08-06, http://pt.slideshare.net/LaisBerlatto2/cucumber-um-breve-review.

SIXPENCE, E.; ADÃO, P.; SMITH, C. AUTOMATIZAÇÃO DE CASOS DE TESTE COMO PROCESSO DE MELHORIA DA QUALIDADE DO SOFTWARE: o caso da aplicação e-learning isupac3 no isutc. In: CONGRESSO LUSO MOÇAMBICANO DE ENGENHARIA - CLME. **Anais...** [S.l.: s.n.], 2011.

VASCONCELOS, A. M. L. de et al. Introdução à Engenharia de Software e à Qualidade de Software. , [S.l.], 2006.