Rosana Silva Matos

Qualidade de software: ferramentas, padrões e boas práticas para testes baseados em interfaces gráficas

São Paulo

2014

Rosana Silva Matos

Qualidade de software: ferramentas, padrões e boas práticas para testes baseados em interfaces gráficas

Monografia apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do título de Especialista em Gestão e Engenharia da Qualidade

Orientador: Adherbal Caminada Netto

São Paulo

2014

Ao meu amado companheiro Rafael,
aos amigos e à minha querida mãe (in memoriam)

Agradecimentos

Agradeço à minha família, aos amigos e ao meu namorado Rafael pelo apoio, companheirismo e ajuda durante todo o curso. Foram essenciais para que eu conseguisse evoluir e dedicar meu tempo a este trabalho. Também agradeço a todos os professores das disciplinas que cursei nesses dois últimos anos de dedicação e estudo, e especialmente pela paciência, ajuda, apoio e conselhos do professor Adherbal. Deixo também meus eternos agradecimentos à professora Regina pelas dicas, técnicas, conselhos, leituras sugeridas e experiências compartilhadas nas disciplinas de comunicação que mudaram muitas coisas em minha vida pessoal e profissional. E ao professor José Aparecido por tudo que aprendi relacionado à liderança e inteligência. Agradeço também a empresa em que trabalho pelo apoio e incentivo à realização deste curso, em especial ao Fabio, um dos melhores líderes que já conheci. Pois foi junto a ele, dentro da empresa, que tive as oportunidades que me fizeram aprender a desaprender e reaprender, de errar, melhorar, aceitar e acertar. Sem essas oportunidades, não teria as experiências necessárias para tornar este curso tão proveitoso, útil e prático.

"A narrativa revela o sentido sem cometer o erro de defini-lo"

Hannah Arendt

Resumo

No desenvolvimento de software, equipes ágeis precisam testar muito e testar sempre. Entregar software funcionando aos clientes de forma contínua exige da equipe feedback constante sobre a qualidade do produto. Assim, automatizar os testes de software em equipes ágeis torna-se imprescindível devido às muitas entregas feitas em períodos muito curtos. Este trabalho mostra uma abordagem sobre questões relacionadas a testes de software e sua automatização, com base em um caso real. Com foco nos testes de aceitação e em métodos ágeis, são apresentadas as ferramentas, linguagens, padrões e boas práticas que a equipe de Quality Assurance usou para a implantar os testes automáticos na empresa, nominada neste trabalho como *Empresa de Serviços de Tecnologia (EST)*.

Palavras-chave: software. testes. ágil.

Abstract

In software development, agile teams need to test a lot and always test. Deliver working software to customers continuously requires team constant feedback on the quality of the product. Thus, automate software testing in agile teams become indispensable due to many deliveries made in very short periods. This work shows an approach on issues related to software testing and its automation, based on a real case. Focusing on acceptance testing and agile methods are shown the tools, languages, standards and best practices that the team of Quality Assurance used to implement automated testing in the company, nominated here as *Empresa de Serviços de Tecnologia (EST)*.

Key-words: software. testing. agile.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

API	Application Programming Interface.
CSS	Cheat Style.
GUI	Graphical User Interface.
HTML	HyperText Markup Language.
PO	Product Owner.
QA	Quality Assurance.
RUP	Rational Unified Process.
XML	Extensible Markup Language.
XP	Extreme Programming.

Sumário

1 Introdução

O desenvolvimento de *software* requer o uso de processos, ferramentas e técnicas adequadas que facilitem sua produção e manutenção. Cada vez mais as empresas buscam aumentar a qualidade do produto, tornando-se necessário melhorar os processos a fim de reduzir custos. Entregar *softwares* com mais funcionalidades a cada versão, em menor tempo e com custos reduzidos, força as empresas a revisarem cada etapa dentro do processo de desenvolvimento para eliminar gargalos e maximizar o desempenho (1). Além do desafio em atender todas as necessidades individuais e específicas de cada cliente. Após uma análise em relação aos testes realizados pela equipe de *Quality Assurance (QA)* da empresa denominada neste trabalho como *Empresa de Serviços de Tecnologia (EST)*, foi observado que a grande dependência de testes manuais se tornou um gargalo para a liberação das versões dos sistemas.

A melhoria da gestão de qualidade abrange uma série de atividades que levarão a melhorar o processo de desenvolvimento de *software*. Em conseqüência, versões do sistema são entregues com melhor qualidade e no prazo (2). Com um processo puramente baseado em testes exploratórios (3), foram dadas à equipe de QA metas para o estudo, definição e implantação dos testes automáticos. Com este desafio em mãos, os analistas de testes foram capacitados e hoje estão aptos a automatizar os testes de aceitação dos sistemas.

1.1 Objetivos gerais

Automatizar *scripts* de testes não é simples pois é preciso seguir padrões para evitar retrabalho e diminuir o custo de manutenção quando ocorrerem mudanças no sistema. Por isso, esse trabalho se concentra principalmente em quais padrões seguir e quais boas práticas manter ao criar *scripts* automáticos.

1.1.1 Objetivos específicos

Este trabalho teve como objetivo específico mostrar ferramentas, padrões e boas práticas para o desenvolvimento de testes automáticos em interfaces gráficas, ou *Graphical User Interface (GUI)*.

1.2 Escopo do trabalho

A equipe de *Quality Assurance (QA)* da empresa, aqui denominada como *Empresa de Serviços de Tecnologia (EST)*, implantou a automatização de testes baseados em

2 Fundamentação Teórica

2.1 Metodologias ágeis

Dezessete líderes de projetos de *software* que se opunham aos modelos tradicionais de desenvolvimento, como os modelos cascata, espiral e *Rational Unified Process (RUP)*, se uniram em 2001 com o objetivo de criar um novo método de desenvolvimento de *software*. No entanto observaram que não seria fácil definir um método perfeito que se adequasse a todas as necessidades e contextos. Assim chegaram ao consenso de listar princípios, 12 no total, que deram origem ao Manifesto Ágil (4). A partir daí, surgiram métodos baseados no Manifesto Ágil, como o *Lean*, *Scrum* e Programação eXtrema (XP) (5) (6). Dizer que um *software* é desenvolvido usando métodos ágeis significa afirmar que a equipe está preparada para mudanças e que se adapta a novos fatores durante o desenvolvimento do *software*, não ficando presa ao planejamento prévio de todo o projeto. Além da equipe trabalhar com *Feedback* constante, também são feitas entregas contínuas de partes operacionais do *software* (7) (2). Os quatro princípios básicos que os métodos ágeis devem valorizar são:

- Indivíduos e iterações, no lugar de processos e ferramentas;
- *Software* funcionando no lugar de documentação muito detalhada;
- Colaboração com os clientes é mais importante que contratos;
- Adaptação às mudanças no lugar de ficar preso ao plano inicial.

Na *EST*, todas as equipes e projetos possuem seu modelo de ciclo de vida baseado em metodologias ágeis. O projeto *SLL* usa a metodologia *Scrum*, detalhada no próximo tópico.

2.2 *Scrum*

Como a metodologia predominante no desenvolvimento de software na *EST* é o *Scrum*, esta seção mostra mais detalhes sobre ele. *Scrum* é um *Framework* para desenvolver e manter produtos, dentro do qual as pessoas podem tratar e resolver problemas complexos de forma adaptativa, enquanto produzem e entregam produtos com o mais alto valor possível. O *Scrum* se baseia na afirmação de que o conhecimento vem da experiência e de tomadas de decisões baseadas no que é conhecido. Além disso, o *Scrum* usa uma abordagem iterativa e incremental para a melhoria contínua na previsão e controle de riscos (1). Os projetos que usam *Scrum* progridem através de várias iterações chamadas de

com mais frequência para detectar os defeitos o quanto antes e dar mais *feedbacks* para a equipe sobre a manutenção da qualidade do produto. Uma nova versão do sistema não pode diminuir a qualidade do que foi entregue em uma versão anterior. Por isso existe a necessidade de sempre executar testes de regressão nas novas versões do produto.

2.4 Testes de Regressão

Os testes de regressão são feitos com o objetivo de garantir que funcionalidades que já existiam no sistema continuam funcionando corretamente. Não se trata de uma fase de testes, mas de uma técnica que pode ser usada em qualquer momento do desenvolvimento do *software*. Se uma nova versão do sistema é lançada, então o analista de teste precisa não somente executar os novos testes como também selecionar uma série de testes, a fim de verificar se defeitos foram introduzidos no sistema durante a implementação da nova funcionalidade (14).

2.5 Testes automatizados

Toda a fase de testes no processo de desenvolvimento de *software* é custosa e requer muito tempo da equipe (2). Em geral, gerir e executar atividades de testes envolve uma grande quantidade de cenários e dados e, quando feitos manualmente, exigem ainda mais tempo, além de serem propensos a erros (15). Automatizar os testes, em oposição aos manuais, significa torná-los independentes de intervenção humana e isso requer o uso de ferramentas específicas e de linguagens de programação (5). Criar, manter, melhorar e executar testes automatizados exige não somente bons conhecimentos em ferramentas e linguagens, como também definir e implementar uma série de boas práticas para um melhor desempenho e diminuição de custos (5). Vários tipos de testes podem ser automatizados, como por exemplo: testes unitários, testes em interfaces gráficas, testes de integração, testes de desempenho e testes de carga.

2.6 Plataformas e ferramentas de desenvolvimento

Para criar e manter os *scripts* de testes automáticos, foram usadas ferramentas de desenvolvimento de *software*, listadas a seguir.

Java

A linguagem de programação escolhida foi Java (16), pois é a linguagem usada em todos os nossos projetos. Isso faz dessa linguagem a mais difundida dentro da empresa e a mais viável para uso, já que assim facilita-se tanto sua aprendizagem

2.7.1.1 WebDriver

Principal interface para os testes. Representa um navegador *web* e fornece métodos para controlá-lo. A tabela 1 apresenta os principais métodos desta interface.

Tabela 1 – Exemplos de métodos disponíveis na classe WebDriver do Selenium.

void close()	Fecha a janela atual, saindo do navegador.
WebElement findElement(By by)	Procura pelo elemento na página de acordo com o argumento dado (xpath, css-selector etc).
java.util.List<WebElement> findElements(By by)	Procura por todos os elementos da página de acordo com o argumento dado (xpath, css-selector etc).
void get(java.lang.String url)	Carrega a página passada como argumento no navegador.
java.lang.String getCurrentUrl()	Retorna uma string que contém a URL aberta pelo navegador.
java.lang.String getPageSource()	Retorna o código fonte da última página aberta pelo navegador.
java.lang.String getTitle()	Retorna o título da página atual do navegador.
WebDriver.Options manage()	Permite gerenciar cookies do navegador, logs, timeouts etc.
WebDriver.Navigation navigate()	Abstração que permite acessar o histórico e navegar para uma determinada URL.
void quit()	Fecha a instância do Selenium WebDriver e todas os navegadores associados.

2.7.1.2 WebElement

Representa um elemento em HTML. Essa interface apresenta as principais operações que podem ser feitas para interagir com um elemento de uma página *web*. A tabela 2 apresenta os principais métodos desta interface.

2.7.1.3 WebDriverWait

Testes baseado em GUI são pesados e lentos (5). Uma base de testes muito grande pode levar horas para ser executada, então é imprescindível usar estratégias para que essa base seja executada em um tempo razoável para a equipe (31). Além disso, um método de teste deve ser criado para ser o menor possível. Testes curtos são mais focados, são bem mais rápidos durante a execução, e são mais propensos a serem executados mais cedo e com mais frequência. Além disso são mais fáceis de manter (32).

A interface WebDriverWait permite controlar o tempo de espera entre as ações executadas na tela do sistema, através da classe ExpectedConditions (33). O Thread.sleep é uma alternativa ao uso da classe WebDriverWait, mas interrompe a execução do teste por um tempo exato e fixo. O que é uma desvantagem já que se a ação estiver pronta para

PageFactory. A classe PageFactory procura os elementos na página *web* e os inicializa para que a Page Object os utilize. Para indicar à PageFactory quais elementos ela deve procurar, é usada a anotação:

@FindBy Marca um campo da Page Object indicando o mecanismo para localizar o elemento na página *web*.

Ao usar a notação @FindBy deve-se especificar como encontrar o elemento desejado preenchendo os campos a seguir (ver exemplos no Apêndice C):

how Especifica o método para localizar o elemento. Uma forma de localizar os elementosm é usando seletores CSS, por exemplo.

using Especifica o *locator* do elemento.

Uma base de testes completa envolve o uso de várias ferramentas e diz respeito a vários tipos de testes: unitários, de integração, de negócios e por final, os testes de GUI. O Selenium não resolve tudo e a ferramenta sequer se propõe a isso. Então recomenda-se usar o Selenium apenas como uma das estratégias de automatização dos testes do projeto. A seguir é apresentada uma lista com algumas abordagens de testes que devem ser seguidas pela equipe de desenvolvimento. A figura 2 apresenta a Pirâmide de Automatização que mostra qual é a proporção ideal em relação à quantidade de teste para cada uma dessas abordagens (9).

Testes Manuais

Feitos pelo analista de testes.

Deve ter o foco em novas funcionalidades.

Deve se basear em testes exploratórios.

Para uma boa estratégia, a base de testes automáticos para os testes de regressão deve ser eficaz e confiável. Assim o analista de testes se preocupa mais com os novos testes.

Testes em GUI

Quebram mais facilmente.

Lentos e por isso mais custosos.

Devem representar a menor porcentagem da base de testes.

Testes de API

Tratam regras de negócio.

Muito úteis quando ocorrem mudanças no sistema.

Por serem mais leves, devem ser em maior quantidade do que os testes em GUI.

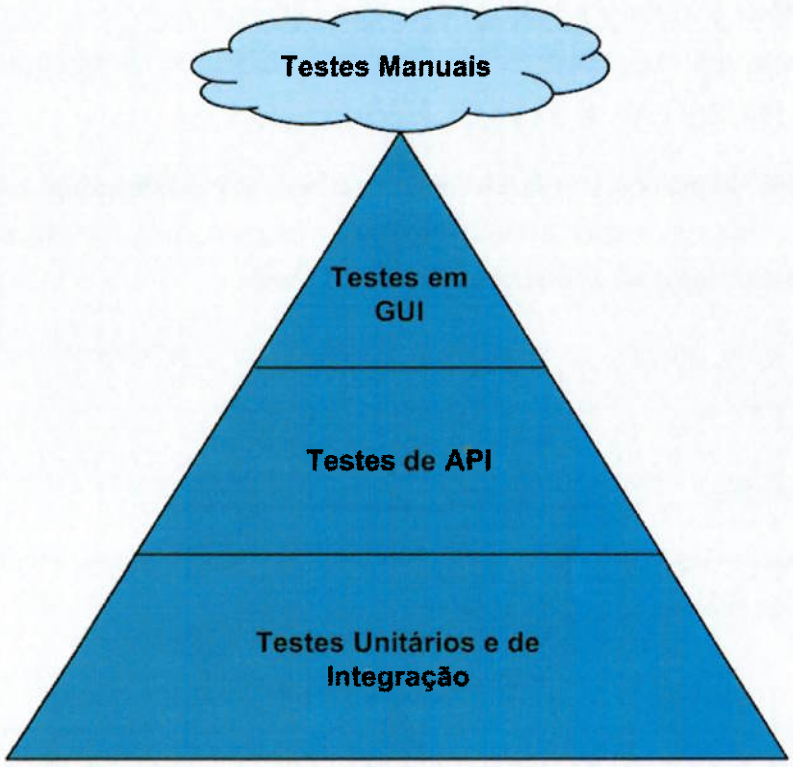

Figura 2 – Pirâmide de Automação - proporção ideal para cada abordagem de testes. Fonte: Crispin L. et al (9)

os testes e uma aplicação gráfica para executá-los (também pode ser feito em terminal, usando linha de comando). É uma ferramenta simples e fácil de usar, basta conhecer suas anotações e asserções. Além disso já está inclusa na instalação do Eclipse.

2.7.2.1 JUnitAnotations

Testes de unidade escritos com o apoio do JUnit fazem uso de algumas anotações de uso geral. Elas devem ser usadas logo acima da assinatura do método (ver Apêndice A) e indicam qual a função do método na classe: se é de pré-configuração para o teste, pós-configuração para o teste ou se é para ser executado como um caso de teste. As anotações são:

@Test Principal anotação. Indica que o método anotado é um teste.

@Before Identifica o método para que seja executado antes de cada método de teste. Efeito simétrico à anotação @After.

através de um outro método, ou através de outra classe, ou ainda através de um arquivo de extensão csv.

2.7.3 Model Citizen

Model Citizen (26) é um modelo baseado em anotações para a geração de dados para os testes. Com ele é possivel mapear as classes de entidades do projeto para os testes. Dessa forma consegue-se gerar os dados dos testes de forma dinâmica para que posteriormente sejam inseridos no banco de dados usando o DbSetup (ver Apêndice B).

2.7.4 DbSetup

DbSetup (27) é uma API Java *open-source* e gratuita que ajuda a criar cenários de banco de dados para executar testes unitários. Similar ao DBUnit (28), porém mais simples, sem a necessidade de usar arquivos em XML. Essa ferramenta permite inserir os dados necessários para os testes no banco de dados e após sua execução, limpar o banco de dados, voltando ao seu estado inicial. O DBSetup foi usado nesse projeto juntamente com o Model Citizen. Enquanto o Model Citizen gera os dados para os testes, o DBSetup é usado para inserí-los e removê-los do banco de dados (ver Apêndice B).

2.8 Outras ferramentas

2.8.1 Seletores CSS e jQuery

Os seletores CSS são utilizados para identificar elementos e definir os estilos que serão aplicados a eles. No caso dos testes automáticos, os seletores CSS são utilizados para identificar elementos na página *web*. O jQuery (29) é uma biblioteca para desenvolvimento rápido de *javascript* que interage com páginas HTML. Com ela é possível atribuir eventos, definir efeitos, criar e alterar elementos da página. Além disso, permite criar *locators* para indentificar elementos na página. A identificação do elemento, chamado de *locator*, é usado pelo Selenium Webdriver para que seja aplicada uma ação, seja clicar em um botão, preencher um campo ou até mesmo validar uma mensagem exibida na tela.

2.8.2 Inspeção de elementos

Firebug (30) é uma extensão para o Firefox que permite inspecionar HTML, CSS, *Document Object Model (DOM)* e JavaScript. Possui diversos recursos, mas para fins de automação com o Selenium, é usado apenas para inspecionar a estrutura e atributos dos elementos da tela. Existe também o Firebug Lite, uma versão simplificada do Firebug, que é compatível com os demais navegadores: Google Chrome, Opera, IE6+ e Safari. No

3 Sobre a Empresa de Serviços de Tecnologia

A *EST* é uma empresa de desenvolvimento de *software*, voltada para a área de saúde, atualmente com duas unidades, uma em São José dos Campos e outra em São Paulo. Desenvolve *softwares* para hospitais, laboratórios e consultórios médicos que auxiliam em diversos processos: desde o gerenciamento financeiro até a entrega dos resultados aos pacientes.

3.1 Missão, Visão e Valores

3.1.1 Missão

A missão da *EST* é combinar excelência técnica, conhecimento de causa, compromisso e colaboração de talentos, para criar *software* que permita ao cliente prestar serviços personalizados e ao mesmo tempo eficientes, operar de forma simples em ambientes complexos e transformar seu conhecimento em valor para seu negócio.

3.1.2 Visão

Tornar-nos referência em soluções de tecnologia e suporte ao cliente, fundamentando-nos em nosso *know-how*, nossa plataforma, equipe e na capacidade de inovar continuamente.

3.1.3 Valores

- Confiabilidade;
- Transparência;
- Colaboração;
- Domínio do negócio;
- Previsibilidade;
- Prontidão no suporte;
- Comprometimento com prazos;
- Disciplina nos processos;
- Ética.

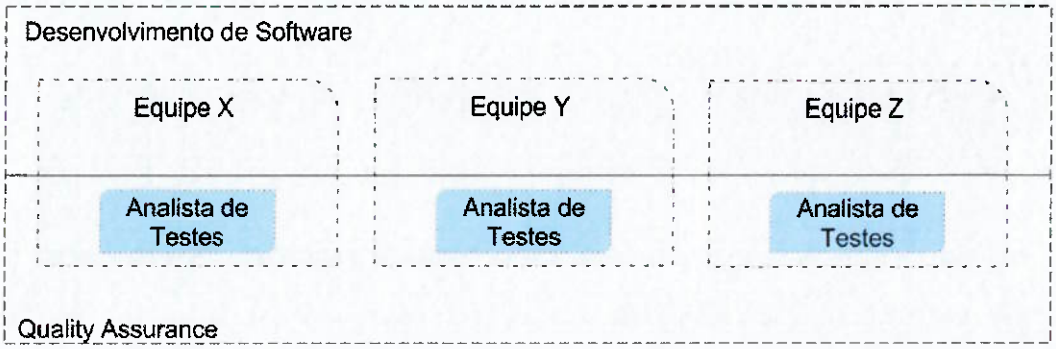

Figura 4 – Visão de relação entre equipe de QA e desenvolvimento: apesar de fisicamente estarem separados, ainda assim os analistas de testes formam uma equipe

entrada de novos clientes e novos pedidos. Os projetos desenvolvidos não possuem data de término, pois são aplicações usadas em processos complexos de gestão médico, laboratorial e hospitalar e, a cada dia, novos conjuntos de melhorias e funcionalidades são adicionados aos sistemas. Como os testes são manuais, a dificuldade em refazer os testes de regressão nos sistemas aumenta. Um exemplo prático dessa demanda pode ser visualizado a partir dos dados do projeto do *SLL*. A cada versão do sistema, além dos testes manuais das novas funcionalidades, todos os testes manuais de regressão precisam ser refeitos. A tabela 4 mostra dados reais da quantidade de alterações no sistema em um dado período e a relação entre programadores e analistas de testes da equipe.

Tabela 4 – Alterações no sistema, total de programadores e total de analistas de testes para o projeto do *Sistema de Liberação de Laudos*.

Período 08/02/2014 a 08/03/2014			
Alterações no código	Erros e melhorias	Programadores	Analistas de Testes
106	71	10	1

Além disso, a figura 5 mostra dados para um módulo do projeto *SLL*, no mesmo período. É perceptível a tendência que a complexidade do sistema tem em aumentar (linha azul).

Gerenciar, validar e garantir que todas essas alterações dêem origem a entregas confiáveis e livres de problemas críticos se torna um grande desafio, uma vez que as equipes de testes da *EST* o fazem de forma manual. O uso de processos ágeis ajuda a codificar e programar entregas mais curtas, mas não é fator determinante quanto a manutenção da qualidade dos produtos. Processos de desenvolvimento ágeis requerem processos de testes ágeis, que acompanhem a evolução dos sistemas e que possam garantir que os testes já feitos manualmente em versões anteriores do sistema sejam feitos automaticamente em versões futuras, formando assim uma base de testes de regressão automáticos para os produtos da *EST*. Essa base, no entanto, precisa ser confiável e fácil de manter, para que não seja facilmente descartada e evitar retrabalho o máximo possível.

4 Ferramentas, padrões e boas práticas para a manutenção dos testes

Este capítulo apresenta as ferramentas de desenvolvimento, os padrões e boas práticas que são usados para a criação e manutenção dos testes automáticos em GUI do projeto *SLL*.

4.1 Plataformas e ferramentas de desenvolvimento

A seleção das ferramentas de desenvolvimento é crucial para o sucesso do projeto de automatização. Tanto para que os analistas de testes sejam treinados e capacitados para trabalhar no projeto, quanto para que a base de testes automáticos seja sustentável.

A figura 6 mostra uma visão geral de todas as ferramentas definidas para o desenvolvimento dos testes do projeto *EST*.

4.2 Selenium WebDriver

Para facilitar a manutenção dos testes, foi usado o padrão Page Object com o auxílio da classe PageFactory do Selenium WebDriver (ver Apêndice C). Para considerar que os testes estão dentro do padrão de Page Object, foi definido que:

- Cada tela do sistema deve ter pelo menos uma Page Object específica, que contém somente os elementos e serviços específicos da tela;
- Deve existir pelo menos uma Page Object geral que contém todos os elementos e serviços comuns a todas (ou várias) telas do sistema, a fim de evitar código duplicado. Todas as Page Objects específicas devem herdar da Page Object geral;
- Todos os métodos de testes devem usar somente os serviços fornecidos por suas respectivas Page Objects. Um método de teste não pode apresentar o uso de API do Selenium WebDriver. Isso garante que o *script* de teste está isolado, facilitando sua manutenção.

A figura 7, no final deste capítulo, mostra uma visão geral da estrutura dos testes usando o padrão Page Object.

que se trata de uma classe de Page Object que contém os serviços que serão usados nas classes de testes.
Exemplo de nome para classe de Page Object: LoginPageObject.

Classes de *Factory*

Cada classe que utiliza Model Citizen e DbSetup para criação dos cenários de testes deve ter em seu nome a palavra *"Blueprint"*, indicando que se trata de uma classe do tipo *Factory*.
Exemplo de nome para classe do tipo Factory: LoginBlueprint.

Métodos de Testes

Dentro da classe de testes, cada método de teste deve ser nomeado de forma que:
a) Evite o uso de gerundismo;
b) Indique o objetivo do teste;
c) Indique o tipo da validação do teste: se está validando um caso de sucesso ou de falha.
Exemplos: criarUsuarioValidoDeveRetornarMensagemDeSucesso, logarNoSistemaComUsuarioInvalidoDeveMostrarMensagemDeErro.

Asserções

Todo método de teste deve acabar com uma ou mais asserções (ver JUnit no capítulo 2). O principal objetivo de um método de teste é validar ou verificar um evento ou registro. Pode ser uma mensagem de sucesso ou de erro, se um texto está correto, ou então se um registro está no banco de dados do sistema, entre outros.

4.4 Independência entre os casos de testes

Cada caso de teste deve ser independente dos resultados dos demais testes. Isso evita com que o sucesso na execução de um caso de teste dependa do sucesso na execução de outros (um ou mais) casos de testes. Se isso acontece, então sempre que der erro na execução de um caso de teste, não significará que de fato esse caso de teste está com erro. Poderá acontecer de ter dado erro em outros casos dos quais a execução deste dependia. Para o projeto *SLL* foi mantida a independência entre os testes adicionando no banco de dados cenários específicos para o teste. Assim, a edição de um registro, por exemplo, não dependerá da execução do teste que cria esse mesmo registro no sistema. O Apêndice B apresenta um exemplo de como o DbSetup e Model Citizen foram usados para manter a independência entre os testes.

erros/validações), funcionalidades isoladas, telas mais usadas e as menos usadas também. Ao contrário, é sempre recomendado que a equipe escolha uma das estratégias para automatizar os testes baseados em GUI.

Confiança da base de testes

A base de testes precisa ser confiável, principalmente se for muito grande. Isso porque a equipe pode perder tempo com casos de testes que dão falso negativo - quando não existe falha no sistema, mas o *script* de teste não foi feito da forma correta e apresenta erro de execução. Ou com falso positivos - quando um teste deveria falhar, mas apresentou sucesso na execução.

Com o objetivo de manter sua base confiável, sempre se deve perguntar:

Essa base de testes garante o que? Dá segurança? Devem ser consideradas as estratégias e objetivos para os quais foi criada.

4.7 Outras boas práticas

Para o projeto *SLL*, as seguintes resoluções foram tomadas:

- Os testes serão executados a cada versão liberada para o ambiente de testes e a cada versão liberada para o ambiente de produção. Será usada a ferramenta Jenkins e sua execução é responsabilidade do analista de testes.
- Também foi combinado de que só serão automatizadas as telas e funcionalidades mais estáveis. O objetivo será diminuir os custos com manutenção dos *scripts*.
- Não serão automatizados os fluxos alternativos, em que os resultados são negativos. Ou seja, serão automatizados somente os fluxos principais em que o resultado do teste é positivo.
- A prioridade é sempre automatizar as telas e funcionalidades que estão mais próximas aos usuários finais do sistema, mas sempre considerando a estabilidade dos mesmos.

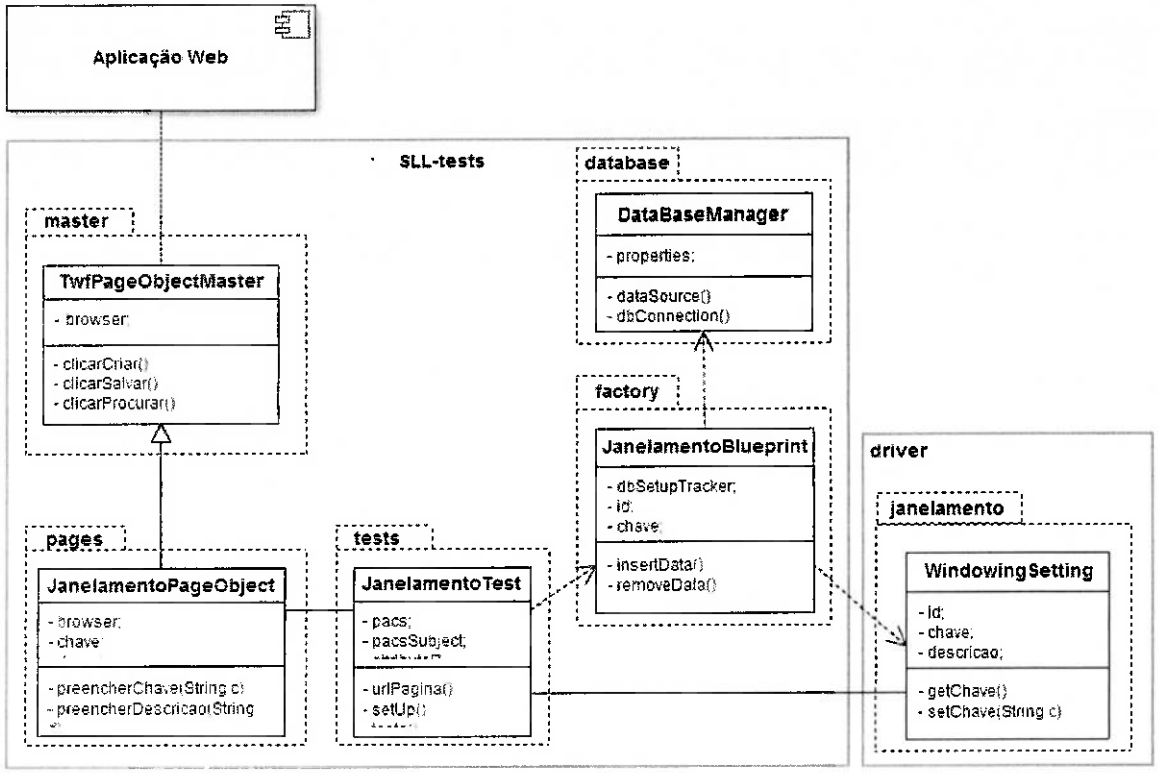

Figura 7 – Visão geral da estrutura dos testes do projeto *SLL*

5 Considerações Finais

O projeto de automatização de testes em interfaces gráficas da *EST* definiu métodos, ferramentas, padrões e boas práticas para criação e manutenção da base de testes.

Ao definir métodos, foi necessário adicionar uma nova etapa no processo dos testes feitos pelos Analistas de Testes, uma vez que todo o tempo deles era gasto em testes manuais. Dentro dessa etapa foi acrescentada uma atividade em que o Analista de Teste deve, para as novas funcionalidades do sistema, identificar quais cenários ou fluxos são automatizáveis para que sejam validados por toda a equipe. Também foi estipulado que cada equipe será responsável por decidir quais estratégias e quais testes são os mais adequados ao seu sistema. No entanto, sempre considerando os padrões e boas práticas para testes em GUI.

Em relação às ferramentas e linguagens, um dos maiores desafios foi escolhê-las diante de tantas opções no mercado, o que foi facilitado logo após ser percebido que o mais importante era o *know-how* da empresa em cada uma delas. Assim, a ideia principal foi usar, o máximo possível, as tecnologias já difundidas dentro da empresa.

Foi preciso montar um plano de treinamento para os novos e antigos integrantes da equipe. Atualmente, os integrantes mais antigos já ajudam a treinar os novos. Foram criados materiais para auxiliar na aprendizagem e um grupo sobre o assunto na rede social interna da empresa. O novo processo, os padrões e boas práticas também foram documentados.

O processo de recrutamento dos novos Analistas de Testes também sofreu mudanças. Isso porque sabe-se que muitos Analistas de Testes não trabalham com programação e ferramentas técnicas da área, pois preferem se especializar na parte de négocios, vendas e implantação dos sistemas. A fim de melhorar o desempenho da equipe na automatização, foi alterado o perfil de contratação para este tipo de vaga. As provas também foram modificadas de acordo com o novo perfil e, durante as entrevistas, são considerados sempre os candidatos que possuem vontade para aprender novas linguagens e ferramentas de programação.

Para o futuro, serão feitas as seguintes atividades:

- Definir metas para acompanhar a produtividade da equipe em relação aos testes automáticos;
- Analisar esses resultados para aplicar mudanças e melhorias;
- Estudar e aplicar estratégias para melhorar o desempenho dos testes, como por

6 Conclusão

O acompanhamento dos resultados do projeto apresentado neste trabalho está em andamento e os benefícios da sua implantação na empresa ainda não foram totalmente avaliadas. As vantagens já são percebidas pela equipe de QA, não somente por já ter *scripts* de testes prontos, como também por todo o aprendizado obtido ao longo da implantação deste projeto.

O tempo que a equipe está ganhando a médio e longo prazo já é perceptível neste momento. A equipe começou a trabalhar nas metas para a implantação dos testes em Janeiro de 2014 e já são observadas as vantagens antes mesmo do ano acabar. O principal benefício até o momento tem sido o fato de o analista de teste poder eliminar tarefas repetitivas e muito mecânicas e, assim, usar esse tempo para outras atividades de testes.

Os testes de regressão, apesar de toda a sua importância e valor reconhecidos, em geral são um tipo de teste que a equipe não consegue executar em todos os casos e cenários a cada versão do sistema, se feito de forma manual. Agora é completamente possível executar todos esses testes dentro dos prazos.

É claro que esta base de testes crescerá muito. E quando for alcançada esta etapa, haverá novos desafios técnicos referentes a desempenho. Mas a equipe já começa a se preparar nesse sentido, estudando enquanto essa base cresce, para futuramente começar a aplicar o que se tem aprendido com os problemas encontrados.

Referências

1 VICENTE, A. A. *Definição e gerenciamento de métricas de teste no contexto de métodos ágeis.* Dissertação (Dissertação de Mestrado) — Universidade de São Paulo, São Carlos, 2010. Disponível em: <http://www.teses.usp.br/teses/disponiveis/55/55134/tde-23062010-083439/>. Acesso em: 2014-03-08. Citado 2 vezes nas páginas 21 e 23.

2 PRESSMAN, R. S. *Software Engineering: a Practioner's Approach.* 7th. ed. McGraw-Hill: Addison-Wesley, 2011. Citado 4 vezes nas páginas 21, 23, 24 e 25.

3 BACH, J. *Exploratory Testing Explained.* Disponível em: <http://www.satisfice.com/articles/et-article.pdf>. Acesso em: 2014-10-05. Citado 2 vezes nas páginas 21 e 24.

4 BEEDLE ARIE VAN BENNEKUM, A. C. W. C. M. F. J. H. A. H. R. J. J. K. B. M. R. C. M. K. S. J. S. D. T. M. *Principles behind the Agile Manifesto.* Página Web. Disponível em: <http://agilemanifesto.org/principles.html>. Acesso em: 2014-10-20. Citado na página 23.

5 BERNARDO, P. C. *Padrões de testes automatizados.* Dissertação (Dissertação de Mestrado) — Universidade de São Paulo, São Paulo, 2011. Disponível em: <http://www.teses.usp.br/teses/disponiveis/45/45134/tde-02042012-120707/>. Acesso em: 2014-03-08. Citado 4 vezes nas páginas 23, 24, 25 e 27.

6 FADEL, H. S. A. *Metodologias ágeis no contexto de desenvolvimento de software: XP, Scrum e Lean.* Dissertação (Dissertação de Mestrado) — Universidade Estadual de Campinas, Limeira, 2010. Disponível em: <http://www.ft.unicamp.br/liag/Gerenciamento/monografias/Lean%20Agil_v8.pdf>. Acesso em: 2014-04-25. Citado na página 23.

7 LIBARDI, V. B. P. *Métodos Ágeis.* Dissertação (Dissertação de Mestrado) — Universidade Estadual de Campinas, Limeira, 2010. Disponível em: <http://www.ft.unicamp.br/liag/Gerenciamento/monografias/monogafia_metodos_ageis.pdf>. Acesso em: 2014-04-25. Citado na página 23.

8 SCHWABER, J. S. K. *Um guia definitivo para o Scrum*: As regras do jogo. [S.l.], 2013. Disponível em: <https://www.scrum.org/Portals/0/Documents/Scrum%20Guides/2013/Scrum-Guide-Portuguese-BR.pdf>. Acesso em: 2014-04-18. Citado na página 24.

9 CRISPIN, J. G. L. *Agile Testing: A Practical Guide for Testers and Agile Teams.* Upper Saddle River: Addison-Wesley, 2009. Citado 3 vezes nas páginas 24, 29 e 31.

10 SIMÃO, A. da S. *Contribuições para o Teste de Software.* Dissertação (Livre Docência em Engenharia de Software) — Universidade de São Paulo, São Carlos, 2011. Disponível em: <http://www.teses.usp.br/teses/disponiveis/livredocencia/55/tde-17082011-153853/>. Acesso em: 2014-03-08. Citado na página 24.

11 FOWLER, M. *Continuous Integration.* Disponível em: <http://www.martinfowler.com/articles/continuousIntegration.html>. Acesso em: 2014-10-05. Citado na página 24.

28 DBUNIT. *About DbUnit.* Disponível em: <http://dbunit.sourceforge.net/>. Acesso em: 2014-09-27. Citado na página 33.

29 FOUNDATION, T. jQuery. *What is jQuery?* Disponível em: <http://jquery.com/>. Acesso em: 2014-09-27. Citado na página 33.

30 FIREBUG. *What is Firebug?* Disponível em: <http://getfirebug.com/whatisfirebug>. Acesso em: 2014-10-05. Citado na página 33.

31 HARTY, J. *Web app acceptance test survival techniques, Part 3: Musings.* Blog Article. Disponível em: <http://googletesting.blogspot.com.br/2009/05/web-app-acceptance-test-survival.html>. Acesso em: 2014-10-20. Citado na página 27.

32 SEMTURS, C. *GTAC 2008: The Value of Small Tests.* Palestra gravada em vídeo. Disponível em: <https://www.youtube.com/watch?v=MpG2i_6nkUg>. Acesso em: 2014-10-20. Citado na página 27.

33 K, M. K. *WebDriver Wait Commands.* Blog Article. Disponível em: <http://assertselenium.com/2013/01/29/webdriver-wait-commands/>. Acesso em: 2014-10-20. Citado na página 27.

34 STEWART, S. *My Selenium Tests Aren't Stable!* Blog Article. Disponível em: <http://googletesting.blogspot.com.br/2009/06/my-selenium-tests-arent-stable.html>. Acesso em: 2014-10-20. Citado na página 28.

35 BARISON, D. *Boas práticas ao criar testes com Selenium.* Blog Article. Disponível em: <http://www.dextra.com.br/boas-praticas-ao-criar-testes-com-selenium/>. Acesso em: 2014-10-20. Citado na página 28.

36 MAZZI, C. E. D. *Convenções de Código Java.* Blog Article. Disponível em: <http://www.devmedia.com.br/convencoes-de-codigo-java/23871>. Acesso em: 2014-10-20. Citado na página 75.

Glossário

Backlog Atividades/tarefas acumuladas, ou Requisitos acumulados.

Blueprint Um plano de projeto, modelo ou desenho técnico.

Factory Fábrica. Referente à produção de dados para os cenários de testes.

Feedback Comentários, parecer, resposta.

Framework Plataforma, model.

Interface Um ponto de comunicação entre dois sistemas. Dispositivo que permite comunicação entre usuário e computador.

Locator Localizador.

Planning Planejamento. Refere-se a uma etapa do Scrum.

Review Revisão. Usado em uma etapa do Scrum: Sprint Review.

Script Sequência de comandos.

Sprint Referente à corrida de velocidade. Refere-se a uma etapa do Scrum.

Timeout Tempo limite.

Apêndices

APÊNDICE A – JUnit - Anotações, Asserções e Parâmetros

A classe abaixo tem o único propósito de ilustrar didaticamente como anotações podem ser empregadas em favor dos testes usando o JUnit. A execução desta classe irá apenas facilitar a compreensão das funções de cada anotação.

Listing A.1 – Exemplo de Anotações do JUnit

```
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import org.junit.Ignore;

public class Anotacoes {

/* A execução irá produzir na saída padrão (o que não é boa prática de teste) a mensagem
    "Uma única vez antes de qualquer teste", decorrente da execução do método
    setUpBeforeClass(). Observe que o método pode ter outro identificador. Sabe-se que
    esta mensagem será exibida antes de qualquer outra ação porque esta é a única senteng
    a do método que será executado antes de qualquer teste da classe, o que é indicado
    pela anotação @BeforeClass.*/
    @BeforeClass
    public static void setUpBeforeClass() throws Exception {
        System.out.println("BeforeClass:" + "Esse método executará uma unica vez antes de
            todos os testes");
    }

/* Em seguida é exibida a mensagem "Antes de todo teste". Esta mensagem é impressa pelo m
    étodo setUp(). Este método, que poderia possuir outro identificador, está anotado por
     @Before, o que significa que o método em questão será executado uma vez antes da
    execução de cada teste da classe. Imediatamente após esta classe é executado um mé
    todo anotado por @Test. Neste caso, o método testSimples().*/
    @Before
    public void setUp() throws Exception {
        System.out.println("Before:" + "Esse método executará sempre antes de cada método
            de teste");
    }

/* A execução dos métodos anotados por @Before e @After ocorre, respectivamente, antes e
    após a execução de todo método de teste, anotado por @Test. Quando não mais existirem
     métodos de teste a serem executados, será executado o método tearDownAfterClass().
    Este é o método executado não por este identificador, mas porque está anotado por
    @AfterClass.*/
    @AfterClass
    public static void tearDownAfterClass() throws Exception {
        System.out.println("AfterClass:" + "Esse método executará uma única vez após
            todos os testes");
    }
```

A classe abaixo tem o único propósito de ilustrar didaticamente o uso de alguns asserts do JUnit.

Listing A.2 – Exemplo de Asserções do JUnit

```
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertFalse;
import static org.junit.Assert.assertTrue;
import org.junit.Test;

public class Assercoes{

    @Test
    public void testeUsandoAssertEquals(){
        // ... Linhas com o código do teste ....

        // Usando assertEquals(java.lang.Object expected, java.lang.Object actual)
        // Este assert validará se o texto atual (da tela) é igual ao da mensagem esperada
            .

        assertEquals("mensagem que eu espero que apareça na tela", tipoDocumento.mensagem
            ().getText());

        // Mensagem Esperada = "mensagem que eu espero que apareça na tela"
        // Mensagem Atual = tipoDocumento.mensagem().getText()
    }

    @Test
    public void testeUsandoAssertTrue(){
        // ... Linhas com o código do teste ....

        // Usando assertTrue(boolean condition)
        // A condição se refere ao elemento da tela que contém a mensagem.
        // O resultado da condição será true (se o elemento aparecer de fato na tela)
        // ou false (caso o elemento não apareça na tela).

        assertTrue(tipoDocumento.mensagem().isDisplayed());

        // A função isDisplayed() do Selenium verifica se o elemento aparece na tela e
            retorna true ou false.
    }

    @Test
    public void testeUsandoAssertFalse(){
        // ... Linhas com o código do teste ....

        // Usando assertFalse(boolean condition)
        // Neste caso o teste passa somente se o elemento que contém a mensagem não
            aparecer na tela.

        assertFalse(tipoDocumento.mensagem().isDisplayed());
    }
}
```

APÊNDICE B – Criação de Cenários - DbSetup e Model Citizen

Para criar os cenários dos testes primeiro é necessário criar uma classe para a conexão com o banco de dados. No projeto SLL usamos o banco PostgreSQL.

Listing B.1 – Manager: essa classe é usada para conectar ao banco de dados

```
package database;
import java.io.IOException;
import java.io.InputStream;
import java.sql.Connection;
import java.sql.DriverManager;
import java.util.Properties;
import javax.sql.DataSource;
import org.springframework.jdbc.datasource.SimpleDriverDataSource;
import com.ninja_squad.dbsetup.DbSetup;
import com.ninja_squad.dbsetup.DbSetupTracker;

public abstract class DatabaseManager {
    //para leitura do arquivo que contém as informações do banco de dados
    private static Properties properties;
    //caminho para o arquivo que contém as informações do banco de dados
    private static final String DATABASE_PROPS = "/META-INF/database.properties";
    //classes do DbSetup usadas para inserir os dados no banco
    protected static DbSetupTracker dbSetupTracker = new DbSetupTracker();
    protected static DbSetup dbSetup;

    /* Esse método lê as informações do banco e retorna um datasource que será usado pelo
        DbSetup para inserir os dados no banco*/
    public static DataSource getDataSource() throws Exception {
        // carrega os valores do arquivo properties
        Properties properties = getProperties();
        String url = properties.getProperty("url");
        String username = properties.getProperty("username");
        String password = properties.getProperty("password");
        return new SimpleDriverDataSource(new org.postgresql.Driver(), url, username,
            password);
    }

    /**
    * retorna o properties
    *
     * @return
    */
    public static Properties getProperties() throws IOException {
        properties = new Properties();
        InputStream is = DatabaseManager.class.getResourceAsStream(DATABASE_PROPS);
        properties.load(is);
        return properties;
    }
}
```

```
    @Default
    private FieldCallback<Float> windowCenter = new FieldCallback<Float>(){  //atributo
        que representa o centro do Janelamento
        @Override
        public Float get(Object arg0) {
            Random rand = new Random();
             return (float) rand.nextInt(1000);
        }
    };

    @Default
    private FieldCallback<Float> windowWidth = new FieldCallback<Float>(){  //atributo
        que representa a largura do Janelamento
        @Override
        public Float get(Object arg0) {
            Random rand = new Random();
             return (float) rand.nextInt(1000);
        }
    };
    // aqui termina o código que utiliza o Model Citizen

    //A partir daqui começa o código que utiliza o DbSetup para a inserção dos dados no
        banco

    /* Esse método insere os dados no banco*/
    public static void insertData(WindowingSetting janelamento) throws Exception{
        Insert insert = Insert.into("DCMD_WINDOWING_SETTING")
                .columns("id", "name", "description", "windowcenter", "windowwidth")
                .values(janelamento.getId(), janelamento.getName(), janelamento.
                    getDescription(), janelamento.getWindowCenter(), janelamento.
                    getWindowWidth()).build();

        dbSetup = new DbSetup(new DataSourceDestination(DatabaseManager.getDataSource()),
             insert);
        dbSetupTracker.launchIfNecessary(dbSetup);
    }

    /* Esse método remove os dados do banco*/
    public static void removeData() throws Exception {
        Operation delete = Operations.sequenceOf(
                DeleteAll.from("DCMD_MODALITY_SETTINGS"),
                DeleteAll.from("DCMD_WINDOWING_SETTING"));

        dbSetup = new DbSetup(new DataSourceDestination(DatabaseManager.getDataSource()),
             delete);
        dbSetupTracker.launchIfNecessary(dbSetup);
    }
}
```

APÊNDICE C – Selenium WebDriver - PageObject e PageFactory

Aqui será apresentada uma classe que usa o padrão PageObject para encapsular informações da página web, e dentro dela será mostrado o uso da classe PageFactory que ajuda a localizar e inicializar os elementos da página.

Listing C.1 – Classe que representa uma PageObject no projeto SLL

```
package pages.crud;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.FindBy;
import org.openqa.selenium.support.How;
import org.openqa.selenium.support.PageFactory;

// Essa é a PageObject. Dentro dela usamos a anotação @FindBy da classe PageFactory.
public class JanelamentoPageObject{

    private WebDriver driver;
    private final String URL = "http://meuDominio/meuSistema";

   // Os atributos da classe são os elementos da página web - aqui é feito o mapeamento
   // Anotação @FindBy faz parte da biblioteca PageFactory
   // campos
   @FindBy(how = How.NAME, using = "j_name")  // exemplo usando "how" e "using"
    public WebElement nome;
   @FindBy(how = How.NAME, using = "j_descricao")
    public WebElement descricao;
   @FindBy(how = How.NAME, using = "j_centro")
    public WebElement centro;
   @FindBy(how = How.NAME, using = "j_largura")
    public WebElement largura;
    // botões
    @FindBy(how = How.NAME, using = "btn_create")
    public WebElement botaoCriar;
    @FindBy(how = How.NAME, using = "btn_update")
    public WebElement botaoEditar;
    @FindBy(how = How.NAME, using = "btn_saveNew")
    public WebElement botaoSalvarNovo;
    @FindBy(how = How.NAME, using = "btn_saveUpdate")
    public WebElement botaoSalvarEdicao;
    @FindBy(how = How.NAME, using = "btn_remove")
    public WebElement botaoRemover;

   /*Construtor da classe - aqui os elementos precisam ser inicializados*/
   public JanelamentoPageObject(WebDriver driver) {
        this.driver = driver;
        PageFactory.initElements(driver, this); // essa linha é obrigatória pois
            inicializa os atributos que usam a anotação @FindBy
   }
```

```
    //A partir daqui começam os métodos de serviços: cadastrar, editar, procurar, remover
        etc


    public JanelamentoPageObject cadastrarJanelamento(String nome, String descricao,
        String centro, String largura) {
        clicarCriar();
        preencherNome(nome);
        preencherDescricao(descricao);
        preencherCentro(centro);
        preencherLargura(largura);
        clicarSalvarNovo();
    }

    public JanelamentoPageObject editarJanelamento(String nome, String descricao, String
        centro, String largura){
        clicarEditar();
        preencherNome(nome);
        preencherDescricao(descricao);
        clicarSalvarEdicao();
    }

   //... a classe continua com os serviços
}
```

```
    @Parameters({"JANELAMENTO_TEST_SEL, TESTE SELENIUM, 132, 100"})
    @Test
    public void editarJanelamentoValidoMostraMensagemDeSucesso(String nome, String
        descricao, String centro, String largura){
        janelamento.buscarPorNome(subject.getName());
        janelamento.editarJanelamento(nome, descricao, centro, largura);
        Assert.assertTrue(janelamento.verificarMensagem("sucesso"));
    }

    @Test
    public void removerJanelamentoRetornaListaVaziaNaBusca(){
        janelamento.buscarPorNome(subject.getName());
        janelamento.clicarRemover();
        WebElement registro = janelamento.buscarPorNome(subject.getName());
        Assert.assertNull(registro);
    }

    //... a classe continua com outros casos de testes para o Janelamento
}
```

Anexos

ANEXO A – Convenções de Código Java

Este anexo apresenta um conjunto de padrões que os desenvolvedores devem seguir ao programar em Java. Esse conteúdo foi criado por Carlos Eduardo Domingues Mazzi e seu texto completo pode ser lido no site do DevMedia no artigo *Convenções de Código Java* (36). Nele são mostradas várias regras, desde os imports e comentários, até nomenclaturas e declarações feitas em Java.
A lista a seguir mostra os padrões para nomes.

Packages

O prefixo do nome do pacote deve ser único, deve sempre ser escrito em letras minúsculas todo-ASCII e deve ser um dos nomes de domínio de nível superior, atualmente com, edu, gov, mil, net, org, códigos de duas letras identificando os países, tal como especificado na norma ISO 3166, 1981. Componentes subseqüentes do nome do pacote varia de acordo com uma organização próprias convenções de nomenclatura internos. Tais convenções podem especificar que certos componentes do nome do diretório haver divisão, departamento, projeto, máquina, ou nomes de login.
Exemplos:
com.sun.eng
com.apple.quicktime.v2
edu.cmu.cs.bovik.cheese

Classes

Os nomes de classe devem ser substantivos, em maiúsculas e minúsculas com a primeira letra de cada palavra interna em maiúscula. Tente manter seus nomes de classe simples e descritivo. Sempre evite palavras-ligadas , evite todas siglas e abreviaturas, seja semântico.
Exemplos:
class Raster;
class ImageSprite;

Interfaces

Nomes de interfaces devem ser usadas com as primeiras letras em maiúsculas como nome de classes.
Exemplos:
interface RasterDelegate; interface Storing;