5
Estudo de caso

Neste capítulo serão apresentados e discutidos os resultados, dificuldades e soluções encontradas na aplicação da abordagem proposta na geração de testes para uma aplicação real. Também serão apresentados os resultados dos testes gerados e considerações sobre as ferramentas desenvolvidas.

5.1.
Visão geral sobre a aplicação alvo dos testes

A aplicação alvo dos testes é uma aplicação web que se encontra em produção e é utilizada por uma empresa que atua no ramo petrolífero.

Esta aplicação faz parte de um sistema que é responsável por coletar e processar dados sobre a integridade de *risers*, que são dutos flexíveis usados para o transporte de petróleo do solo marítimo até a plataforma petrolífera.

A coleta de dados se dá por meio de diferentes tecnologias aplicadas a sensores instalados nas plataformas. O processamento desses dados é feito através da aplicação web que interpreta, através de algoritmos cadastrados, as informações recolhidas pelos sensores e apresenta os resultados para o usuário na forma de gráficos e relatórios, entre outros.

Esta aplicação permite o gerenciamento de diversos locais de produção em tempo real e requer o cadastro de inúmeras informações tais como usuários e suas permissões, algoritmos, gráficos, locais de produção, entre outros. Todas as informações cadastrais, além dos dados brutos recebidos e dados processados, são armazenados em um banco de dados.

5.2.
Aplicação do processo

Dado o elevado número de elementos que podem ser cadastrados no sistema e o tempo disponível para a realização dos testes, apenas um subconjunto de elementos foi escolhido para passar pelo processo de geração de testes. Entretanto, tomou-se o cuidado de que todos os *widgets* de interface que podem ter sua interação com o usuário automatizada (seletores, campos de texto, checkbox, etc)

estivessem representados em pelo menos um dos conjuntos de testes gerados. Também foram escolhidas funcionalidades críticas para o objetivo do sistema (verificar a integridade de *risers*), como as funcionalidades “Cadastrar tipo de local”, “Cadastrar local”, “Cadastrar tipo de equipamento”, “Cadastrar sensor”.

As funcionalidades testadas foram:

- Efetuar autenticação
- Listar grupos
- Cadastrar grupo
- Remover grupo
- Listar tipo de local
- Cadastrar tipo de local
- Listar local
- Cadastrar local
- Listar tipo de equipamento
- Cadastrar tipo de equipamento
- Cadastrar sensor

Cada funcionalidade acima pode ser expressa em um caso de uso e poderão também ser referenciadas como tal a partir de agora. Alguns casos de uso podem ter como pré-condição a execução de outro caso de uso. Na lista acima, por exemplo, o caso de uso “Efetuar autenticação” é pré-condição do caso de uso “Listar grupos”, que, por sua vez, é pré-condição do caso de uso “Cadastrar grupo”. A figura abaixo apresenta um diagrama de estados para esses casos de uso assim como proposto em (Staa, 2010b).

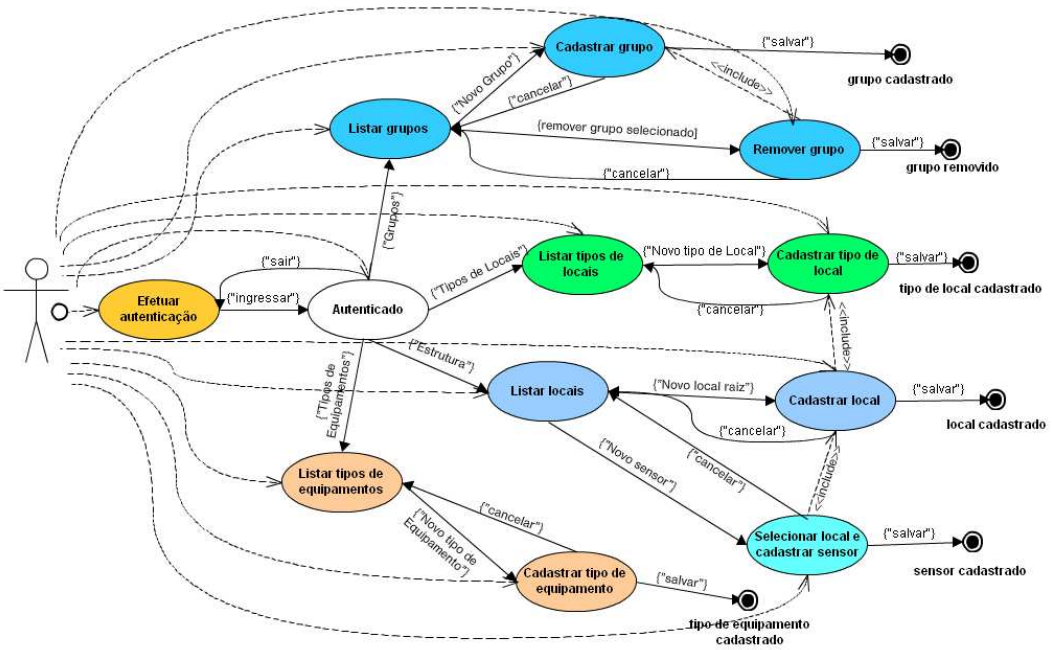


Figura 41 – Diagrama de estado dos casos de uso do sistema alvo.

Antes da elaboração da descrição dos casos de uso, havia a necessidade de adquirir-se certa experiência na criação da tabela de decisão para que se fosse capaz de identificar quais informações seriam essenciais no caso de uso e a melhor forma de apresentá-las, a fim de que pudessem servir de especificações para a elaboração da tabela de decisão. Assim, para as primeiras dez funcionalidades da lista acima, a etapa da criação da descrição do caso de uso foi suprimida e o processo começou pela criação da tabela de decisão.

Após os primeiros testes terem sido gerados com o processo partindo da tabela de decisão, e após a definição de como deveria ser a estrutura da descrição dos casos de uso, foi gerado um formulário de caso de uso para a funcionalidade “Efetuar autenticação”. A inversão na ordem deu-se a fim de que, com base na tabela de decisão já pronta, pudéssemos definir quais informações necessárias à tabela de decisão poderiam ser expressas pelo caso de uso, e consequentemente extraídas deste, e qual seria a melhor forma de representá-las.

Finalmente, após a definição das regras de escrita dos formulários de caso de uso, o processo inteiro foi aplicado na ordem proposta sobre o caso de uso “Cadastrar sensor”: primeiramente houve a criação da descrição do caso de uso, seguida de sua interpretação para a montagem da tabela de decisão; em seguida os casos de teste foram gerados a partir da tabela decisão e finalmente houve a criação automática do script de teste a partir dos casos de teste.

Os resultados da aplicação do processo e das execuções dos testes gerados, bem como a discussão das dificuldades encontradas e algumas soluções propostas, são apresentados na próxima subseção. Também são feitas algumas comparações do tempo gasto no esforço da geração da massa de testes perante outras abordagens de criação de testes. É importante ressaltar que, para essas comparações, não foi levado em conta o tempo gasto na execução da ferramenta desta dissertação que recebe casos de teste semânticos no formato XML e gera scripts de testes executáveis (seções 3.4 e 4.2), visto que esta ferramenta termina sua execução quase que instantaneamente e o esforço do usuário consiste apenas em passar o arquivo de entrada para a ferramenta. Entretanto, o tempo para refatoração do código gerado, quando foi necessário este passo, foi computado.

5.2.1. Exemplo “Efetuar autenticação”

Para um melhor entendimento da aplicação do processo e dos artefatos gerados, vamos analisar o caso de uso “Efetuar autenticação”, o primeiro da lista apresentada na seção anterior.

A tela apresentada para autenticação do usuário e a descrição do caso de uso encontram-se, respectivamente, nas figuras 42 e 43.

Figura 42 – Tela para “Efetuar autenticação”.

<table>
<tr><td>Caso de uso</td><td colspan="2">Efetuar autenticação</td></tr>
<tr><td>Resumo</td><td colspan="2">Usuário deseja autenticar-se no sistema para adquirir direitos de uso.</td></tr>
<tr><td>Escopo</td><td colspan="2">Usuário carrega um endereço no browser e efetua a autenticação fornecendo nome e senha.</td></tr>
<tr><td rowspan="2">Atores</td><td>Usuário</td><td>Obter autorização de acesso ao sistema com determinado direitos de uso.</td></tr>
<tr><td>Sistema</td><td>Permitir somente a usuários acesso ao sistema segundo um determinado conjunto de direito de uso.</td></tr>
<tr><td>Invariante</td><td colspan="2">O cadastro de usuários autorizados está atualizado, disponível e criptografado.</td></tr>
<tr><td>Pré-condições</td><td colspan="2">O usuário carregou o endereço ‘HTTP://localhost:8081/aut’.</td></tr>
<tr><td>Fluxo principal</td><td colspan="2">1. O usuário preenche o campo Nome.
2. O usuário preenche o campo Senha.
3. O usuário clica no botão Ingressar.
4. O sistema valida as informações fornecidas.
5. O sistema exibe a tela principal da aplicação.
6. Fim do caso de uso.</td></tr>
<tr><td>Fluxos alternativos</td><td colspan="2">EVENTO E1: O usuário preenche o campo Nome vazio
ALTERNATIVA AO PASSO: 1
E1.1. O sistema exibe a mensagem ‘Erro de login’.
E1.2. RETORNA AO PASSO 1
FIM EVENTO E1

EVENTO E2: O usuário preenche o campo Nome com valor não cadastrado.
ALTERNATIVA AO PASSO: 1.
E2.1. O sistema exibe a mensagem ‘Erro de login’.
E2.2. RETORNA AO PASSO 1.
FIM EVENTO E2.

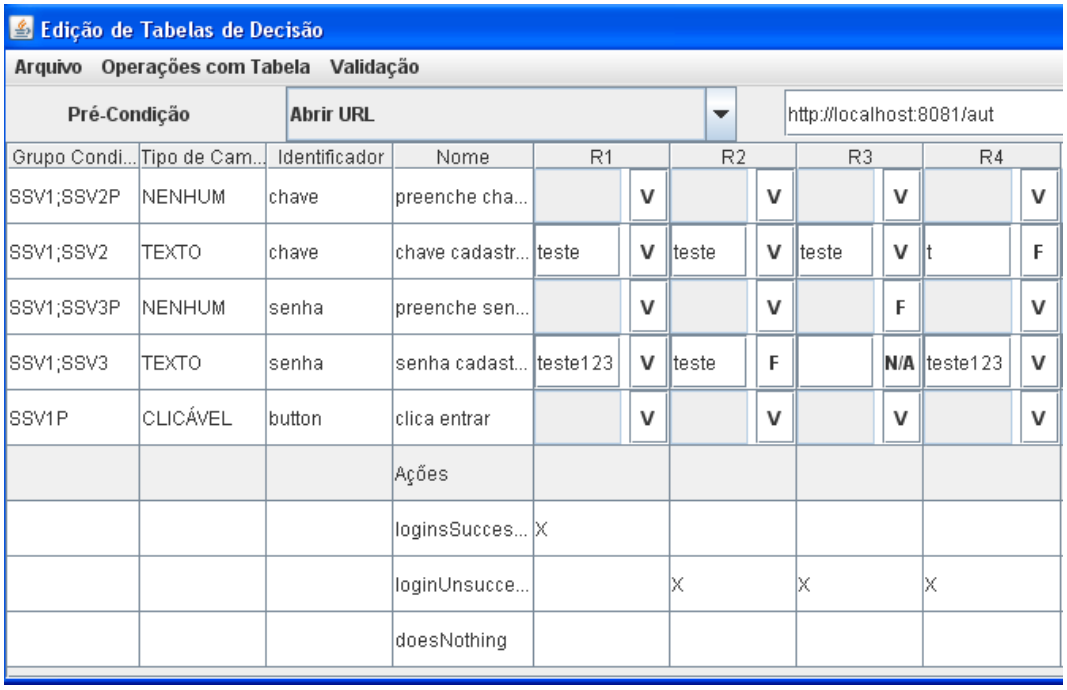
EVENTO E3: O usuário preenche o campo Senha com valor vazio.
ALTERNATIVA AO PASSO: 2.
E3.1. O sistema exibe a mensagem ‘Erro de login’.
E3.2. RETORNA AO PASSO 2.
FIM EVENTO E3.

EVENTO E4: O usuário preenche o campo Senha com valor não cadastrado.
ALTERNATIVA AO PASSO: 2.
E4.1. O sistema exibe a mensagem ‘Erro de login’.
E4.2. RETORNA AO PASSO 2.
FIM EVENTO E4.</td></tr>
<tr><td>Pós-condições</td><td colspan="2">O usuário está autenticado e com os respectivos direitos de acesso.</td></tr>
<tr><td>Regras de negócio</td><td colspan="2">-Restrições de Campos:
Nome:
1. Não pode ser vazio.
2. Deve estar cadastrado.

Senha:
1. Não pode ser vazio.
2. Deve estar cadastrada, considerando o nome fornecido.</td></tr>
</table>

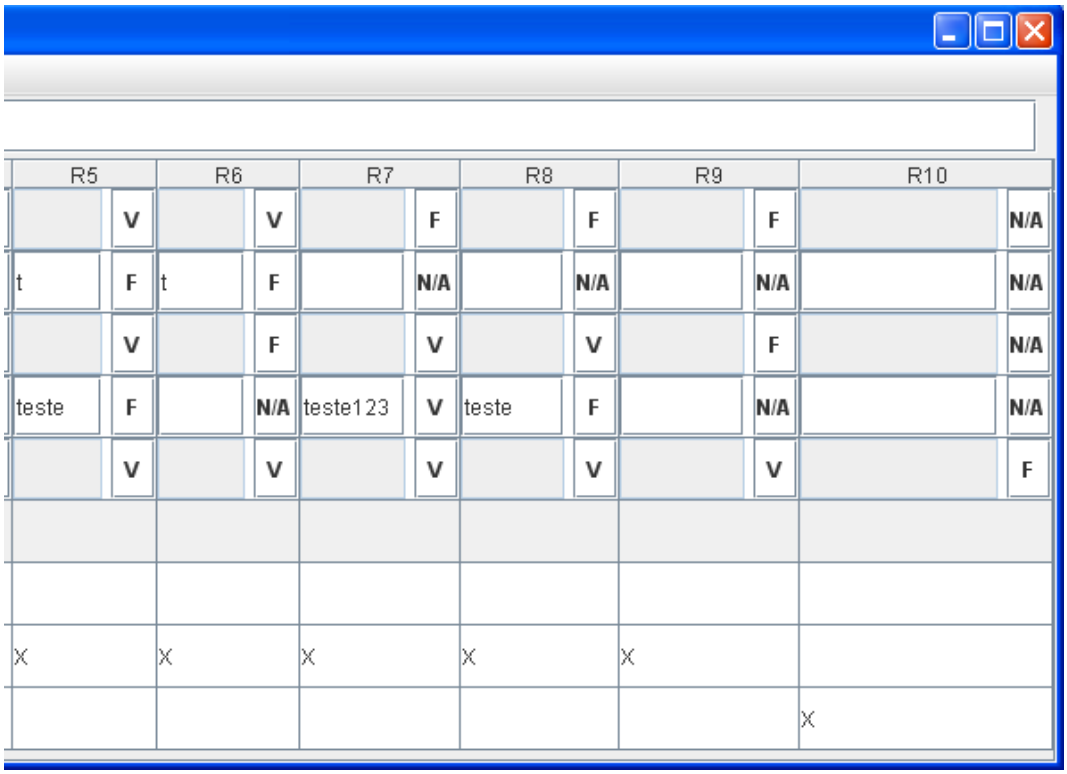
Figura 43 – Descrição de caso de uso para “Efetuar autenticação”.

A partir da descrição do caso de uso e com o auxílio da ferramenta desenvolvida montamos a tabela de decisão das duas figuras abaixo:



Grupo Condi...	Tipo de Cam...	Identificador	Nome	R1		R2		R3		R4	
SSV1;SSV2P	NENHUM	chave	preenche cha...		V		V		V		V
SSV1;SSV2	TEXTO	chave	chave cadastr...	teste	V	teste	V	teste	V	t	F
SSV1;SSV3P	NENHUM	senha	preenche sen...		V		V		F		V
SSV1;SSV3	TEXTO	senha	senha cadast...	teste123	V	teste	F		N/A	teste123	V
SSV1P	CLICÁVEL	button	clica entrar		V		V		V		V
			Ações								
			loginsSucces...	X							
			loginUnsucce...			X		X		X	
			doesNothing								

Figura 44 – Primeira parte da tabela de decisão para “Efetuar autenticação”.



R5		R6		R7		R8		R9		R10	
	V		V		F		F		F		N/A
t	F	t	F		N/A		N/A		N/A		N/A
	V		F		V		V		F		N/A
teste	F		N/A	teste123	V	teste	F		N/A		N/A
	V		V		V		V		V		F
X		X		X		X		X			
										X	

Figura 45 – Segunda parte da tabela de decisão para “Efetuar autenticação”.

A partir desta tabela de decisão é gerado de forma automática um arquivo XML descrevendo os casos de testes e, a partir deste XML, outra ferramenta gera automaticamente o código de testes automatizado da figura 46, que executa os passos descritos pelas condições da tabela de decisão. Este código utiliza como oráculos as ações da tabela de decisão. Estes oráculos são representados no teste por chamadas a funções que encontram-se em outra classe (Results), representada na figura 47. Estas funções, os oráculos, devem ter o mesmo nome das ações da tabela de decisão e devem ser escritas manualmente.

```
package testes;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import com.thoughtworks.selenium.DefaultSelenium;

public class TestLoginSuite
{

	public DefaultSelenium getSelenium(){
		return this.selenium;
	}

	@Test
	public void r1(){
		final DefaultSelenium selenium = getSelenium();
		waitForElement( selenium, "chave" );
		selenium.type( "chave", "teste" );
		waitForElement( selenium, "senha" );
		selenium.type( "senha", "teste123" );
		waitForElement( selenium, "button" );
		selenium.click( "button" );
		Results.loginsSuccessful( selenium );
	}

	@Test
	public void r10(){
		final DefaultSelenium selenium = getSelenium();
		Results.doesNothing( selenium );
	}

	@Test
	public void r2(){
		final DefaultSelenium selenium = getSelenium();
		waitForElement( selenium, "chave" );
		selenium.type( "chave", "teste" );
		waitForElement( selenium, "senha" );
		selenium.type( "senha", "teste" );
		waitForElement( selenium, "button" );
		selenium.click( "button" );
		Results.loginUnsuccessful( selenium );
	}

	@Test
	public void r3(){
		final DefaultSelenium selenium = getSelenium();
		waitForElement( selenium, "chave" );
		selenium.type( "chave", "teste" );
```

```
		waitForElement( selenium, "button" );
		selenium.click( "button" );
		Results.loginUnsuccessful( selenium );
	}

	@Test
	public void r4(){
		final DefaultSelenium selenium = getSelenium();
		waitForElement( selenium, "chave" );
		selenium.type( "chave", "t" );
		waitForElement( selenium, "senha" );
		selenium.type( "senha", "teste123" );
		waitForElement( selenium, "button" );
		selenium.click( "button" );
		Results.loginUnsuccessful( selenium );
	}

	@Test
	public void r5(){
		final DefaultSelenium selenium = getSelenium();
		waitForElement( selenium, "chave" );
		selenium.type( "chave", "t" );
		waitForElement( selenium, "senha" );
		selenium.type( "senha", "teste" );
		waitForElement( selenium, "button" );
		selenium.click( "button" );
		Results.loginUnsuccessful( selenium );
	}

	@Test
	public void r6(){
		final DefaultSelenium selenium = getSelenium();
		waitForElement( selenium, "chave" );
		selenium.type( "chave", "t" );
		waitForElement( selenium, "button" );
		selenium.click( "button" );
		Results.loginUnsuccessful( selenium );
	}

	@Test
	public void r7(){
		final DefaultSelenium selenium = getSelenium();
		waitForElement( selenium, "senha" );
		selenium.type( "senha", "teste123" );
		waitForElement( selenium, "button" );
		selenium.click( "button" );
		Results.loginUnsuccessful( selenium );
	}

	@Test
	public void r8(){
		final DefaultSelenium selenium = getSelenium();
		waitForElement( selenium, "senha" );
		selenium.type( "senha", "teste" );
		waitForElement( selenium, "button" );
		selenium.click( "button" );
		Results.loginUnsuccessful( selenium );
	}

	@Test
	public void r9(){
		final DefaultSelenium selenium = getSelenium();
		waitForElement( selenium, "button" );
		selenium.click( "button" );
		Results.loginUnsuccessful( selenium );
	}

	@Before
```

```
	public void setUp(){
		final Pattern pattern = Pattern.
				compile("(http://[^\\s]+)(/[^\\s]+)" );
		String openArg = "/";
		String address = "http://localhost:8081/aut";
		final Matcher matcher = pattern.matcher( address );
		if ( matcher.matches() )
		{
			address = matcher.group( 1 );
			openArg = matcher.group( 2 );
		}
		this.selenium = new DefaultSelenium(
				"localhost", 4444,"*firefox", address );
		this.selenium.start();
		this.selenium.open( openArg );
	}

	@After
	public void tearDown(){
		this.selenium.stop();
	}

	public void waitForElement( final DefaultSelenium selenium,
				final String locator ){
		int count = 0;
		while ( !selenium.isElementPresent( locator ) )
		{
			try{
				if ( count > 200 ){
				final String msg = String.format(
				"Elemento com localizador '%s' não encontrado",
				locator );
				throw new RuntimeException( msg );
				}
				Thread.sleep( 100 );
				count++;
			}
			catch ( final InterruptedException e ){}
		}
	}
	DefaultSelenium selenium;
}
```

Figura 46 – Teste gerado automaticamente para “Efetuar autenticação”.

```
package testes;
import junit.framework.Assert;
import com.thoughtworks.selenium.Selenium;

public class Results
{

	public static void doesNothing( final Selenium selenium ){
		assert true;
	}

	public static void loginsSuccessful( final Selenium selenium ){
		try{
			Thread.sleep( 3000 );
		}
		catch ( final InterruptedException e ){}
		waitForElementAndAssert( selenium, "cadastros" );
	}

	public static void loginUnsuccessful( final Selenium selenium ){
		waitForElementAndAssert( selenium, "loginMensagemImagem" );
	}
```

```
	public static void waitForElementAndAssert(
			final Selenium selenium, final String locator ){
		int count = 0;
		while ( !selenium.isElementPresent( locator ) )
		{
			try{
				if ( count > 200 ){
					break;
				}
				Thread.sleep( 100 );
				count++;
			}
			catch ( final InterruptedException e ){
				// DOES NOTHING
			}
		}
		 Assert.assertTrue( "Elemento: " + locator +
		 " não encontrado.", selenium.isElementPresent(locator));
	}
}
```

Figura 47 – Oráculos para teste “Efetuar autenticação”.

5.2.2. Resultados e considerações

O tempo gasto para a montagem da tabela de decisão para cada funcionalidade é mostrado na tabela abaixo, juntamente com o tempo gasto para a escrita dos métodos responsáveis por implementar as ações (oráculos). As linhas na tabela estão em ordem cronológica de implementação dos testes.

Caso de uso	Tempo para montar tabela (min.)	Tempo para refatoração do código gerado (min.)	Tempo para programar oráculos (min.)	Tempo total (min.)
Efetuar autenticação	8	0	5	13
Listar grupos	2	0	3	5
Cadastrar grupo	18	0	27	45
Remover grupo	8	0	3	11
Listar tipos de local	1	0	1	2
Cadastrar tipo de local	16	0	6	22
Listar locais	1	0	1	2
Cadastrar local	33	0	8	41
Listar tipos de equipamento	1	0	1	2
Cadastrar tipo de equipamento	3	0	2	5
Cadastrar sensor tipo A	89	8	8	105
Cadastrar sensor tipo B	28	0	4	32
Cadastrar sensor tipo C	3	0	2	5

Tabela 4 – Tempo gasto para a geração dos testes a partir da tabela de decisão.

A funcionalidade “Cadastrar sensor” apresentava muitos caminhos alternativos e englobava a criação de três tipos diferentes de sensores, através da seleção do tipo do mesmo a partir de um seletor presente no formulário. Dependendo do tipo escolhido, campos diferentes podiam ser carregados dinamicamente no formulário. Dada a sua complexidade, a tabela de decisão para esta funcionalidade ficou com 20 condições distintas, o que torna muito difícil sua edição e a visualização das dependências entre as condições (seção 4.1.1.1). Assim, quebrou-se o teste desta funcionalidade em três tabelas, sendo uma para cada tipo de sensor, obtendo-se três tabelas de decisão menos complexas e de maior facilidade de edição. Esta partição dos testes da funcionalidade “Cadastrar sensor” está representada nas últimas três linhas da tabela acima.

Como explicado na seção 4.1.1.2, os oráculos encontram-se em uma classe separada da dos testes. Estes são métodos estáticos que devem ser codificados manualmente, mas que podem ser referenciados e aproveitados por diferentes

testes. Esse reaproveitamento de código explica a redução no tempo de escrita dos oráculos de testes que contêm semelhanças ente si: a escrita de oráculos para testes do tipo “Cadastrar *” consumiu, respectivamente à ordem na tabela, 27, seis, oito e dois minutos. Da mesma forma, enquanto a escrita dos oráculos para os testes de “Listar grupos” consumiu três minutos, a escrita dos oráculos para os outros “Listar *” consumiu apenas um minuto, devido à refatoração e aproveitamento de código já existente.

Caso de uso	Número de regras da tabela de decisão	Número de casos de teste gerados
Efetuar autenticação	10	10
Listar grupos	2	2
Cadastrar grupo	17	17
Remover grupo	3	3
Listar tipos de local	2	2
Cadastrar tipo de local	17	17
Listar locais	2	2
Cadastrar local	33	33
Listar tipos de equipamento	2	2
Cadastrar tipo de equipamento	17	17
Cadastrar sensor tipo A	111	209
Cadastrar sensor tipo B	33	33
Cadastrar sensor tipo C	33	33

Tabela 5 – Número de casos de teste por funcionalidade.

Ao todo foram observadas apenas duas falhas em oito casos de teste. A primeira falha ocorreu em quatro casos de teste de “Cadastrar grupo”: se o formulário não passasse na validação e uma determinada *checkbox* tivesse sido marcada, a tela ficava em branco e era preciso recarregar o endereço do sistema no navegador e recomeçar o processo de efetuar o cadastro de um novo grupo (o comportamento correto seria retornar à tela de cadastro acusando os erros). A segunda falha observada ocorreu em quatro casos de teste de “Cadastrar local”: quando o formulário de cadastro não passava na validação, um seletor, anteriormente preenchido com uma das opções, voltava sem a opção escolhida anteriormente estar selecionada.

5.2.2.1. Comparação com testes existentes produzidos com Squish for Web

O sistema alvo já possuía testes automatizados através da ferramenta Squish for Web (Squish, 2010). Nessa ferramenta os testes podem ser produzidos através de "*capture and replay*" (Araújo e Staa, 2009) ou codificados diretamente. A técnica utilizada para gerar os testes com Squish foi a de codificação manual. Assim, anteriormente à aplicação deste trabalho, ao longo da codificação e execução dos testes com Squish for Web, algumas falhas no sistema foram identificadas e seus defeitos associados corrigidos, o que pode explicar o baixo número de falhas encontradas pelos testes resultantes da tabela de decisão.

Funcionalidade	Casos de teste de interesse utilizando tabelas de decisão	Casos de teste utilizando Squish for Web
Cadastrar grupo	16	2
Cadastrar tipo de local	16	2
Cadastrar local	32	4
Cadastrar tipo de equipamento	16	2
Cadastrar sensor tipo A	208	3
Cadastrar sensor tipo B	32	0
Cadastrar sensor tipo C	32	0

Tabela 6 – Comparação de número de casos de teste gerados.

A tabela 6 fornece, para as funcionalidades do tipo "Cadastrar <elemento>", o número de casos de teste gerados através do processo discutido nesta dissertação, e o número de casos de teste anteriormente gerados com a ferramenta Squish for Web. É importante explicar que cada tabela de decisão do tipo "Cadastrar <elemento>" possuía uma coluna que gerava um caso de teste que, na prática, não fazia nada, pois era o caso onde o usuário não clicava no *link* necessário para abrir a tela de cadastro. Como este teste poderia ser omitido para efeitos de comparação, na tabela 6 a contagem dos números de testes gerados com a tabela de decisão foi subtraída de uma unidade para a comparação com a quantidade de testes gerados com Squish. Analisando os dados presentes nessa tabela é possível constatar que houve mais casos de teste gerados com a utilização de tabelas de decisão em comparação com os testes escritos sem o uso delas, com

a ferramenta Squish. Uma possível explicação para isto é de que o emprego de tabelas de decisão torna mais fácil visualizar as condições a serem testadas.

Outro ponto importante a ser destacado é que metade destes casos de teste gerados com a tabela de decisão (segunda coluna da tabela anterior) contempla os casos onde o usuário clica no botão “cancelar” após preencher o formulário. Estes casos não constam nos testes produzidos com Squish for Web (segunda coluna da tabela anterior), porém, devido à sua menor importância, podem ter sido descartados intencionalmente. Entretanto, mesmo que este seja o motivo, e, tirando-se da contagem estes casos de teste, ainda teríamos, para cada funcionalidade, no mínimo o quádruplo de casos de teste gerados com o auxílio da tabela de decisão em relação ao número de testes produzidos anteriormente por codificação manual para Squish for Web. Em particular, para os casos “Cadastrar sensor tipo B” e “Cadastrar sensor tipo C” não havia sido criado nenhum caso de teste. Provavelmente isso se deu por questões de custo/benefício, visto que já havia testes para sensores do tipo A. Porém com a abordagem presente neste trabalho, os testes para os outros dois tipos de sensores são criados em pouco tempo, como pode ser visto nas duas últimas linhas da tabela 4.

O emprego de qualquer uma das duas técnicas oferecidas pela ferramenta Squish for Web, codificação manual ou código gerado por “*capture and replay*”, não exige a aplicação de nenhum método para a elicitação dos casos de teste. Assim, a existência das duas falhas encontradas e os dados da tabela 6, reforçam o argumento a favor do uso de métodos sistemáticos, como tabelas de decisão, para a elicitação e geração de casos de teste, pois demonstra que, sem tais métodos para escolher as condições a serem testadas, pode-se esquecer algum subconjunto de valores de condições ou de dados de entrada e, consequentemente, deixar de detectar alguma falha (Myers, 2004).

O uso das ferramentas desenvolvidas para auxiliar na geração semi-automática dos testes também parece ser eficaz para reduzir o tempo gasto durante a criação dos scripts de teste. Os testes escritos com Squish não tiveram o tempo necessário para a sua codificação documentado, o que impossibilitou de compará-los com os tempos obtidos pelo método deste trabalho. Entretanto, foi perceptível para o autor, que também havia desenvolvido os testes com Squish for Web, que o uso das ferramentas propostas neste trabalho foi capaz de gerar os testes mais

rapidamente, proporcionalmente ao número de casos de teste gerados, que a codificação manual com a ferramenta Squish.

5.2.2.2. Comparação com testes produzidos manualmente

Os testes já existentes, produzidos com Squish for Web a partir de codificação totalmente manual, não tiveram seus tempos de escrita documentados, de tal forma que não pôde ser feita uma comparação formal entre o tempo gasto na produção destes testes com o tempo consumido na geração dos testes produzidos pelas ferramentas presentes nesta dissertação.

Assim, a avaliação da diferença de tempo gasto entre a geração semi-automática de scripts de teste apoiada pelas ferramentas desta dissertação e a geração puramente manual, foi feita com base na funcionalidade “Cadastrar grupo”, da seguinte forma: o script de teste gerado semi-automaticamente pelo processo foi replicado fidedignamente, em Java e com a utilização de JUnit e Selenium, através de codificação manual, sem o auxílio de funcionalidades de criação de código presentes em algumas IDEs e sem uso de “copiar” e “colar”. O tempo gasto na codificação manual foi comparado com o tempo gasto para montar a tabela de decisão apoiada pela ferramenta. O tempo para a escrita dos oráculos não foi levado em conta em nenhum dos dois casos, pois, em ambos os testes, teriam que ser escritos os mesmos oráculos. Assim, enquanto para a montagem da tabela da funcionalidade “Cadastrar grupo” foram gastos apenas 18 minutos (conforme tabela 4), a codificação manual consumiu 74 minutos. Assim, a geração do script de teste com o uso das ferramentas discutidas neste trabalho exigiu, aproximadamente, apenas 24% do tempo gasto na geração manual do script.

Também foi pedido a uma aluna do curso de Engenharia de Computação, que gerasse dois testes automatizados para a funcionalidade “Cadastrar Grupo”. A aluna não conhecia o sistema e, portanto, as funcionalidades a serem testadas foram-lhe explicadas. O primeiro teste deveria ser produzido através de codificação manual e o segundo com o uso das ferramentas presentes neste trabalho. O tempo gasto na geração dos dois testes e o número de casos de teste em ambos foram comparados. As funções que implementavam os oráculos dos testes foram fornecidas.

Para a criação do teste com codificação manual foi utilizada a IDE Eclipse (Eclipse, 2010). O script de teste consistiu em um arquivo Java, de execução automatizada através de JUnit e interação com o navegador automatizada a partir da API de Selenium para Java (Selenium, 2009). Para a elicitação dos casos de teste, a aluna escreveu em uma folha de papel todas as combinações possíveis de execução do formulário.

Geração manual	Funcionalidade Cadastrar grupo
Elicitação dos casos de teste	25min
Criação da classe contendo os testes	40min
Tempo total	65min
Geração a partir das ferramentas	
Geração da tabela de decisão apoiada por ferramenta (casos de teste semânticos)	22min
Refatoração do código gerado	3min
Tempo total	25min

Tabela 7 - Comparação com tempo gasto para geração manual.

Mesmo sem um método sistemático, como tabelas ou árvores de decisão, a aluna foi capaz de encontrar o mesmo número de casos de teste que foram produzidos com o uso da tabela de decisão apoiada por ferramenta: 16 casos de teste. Entretanto, como pode ser visto na tabela acima, o tempo total para a geração manual do teste foi de 65 minutos, enquanto que a geração com as ferramentas presentes neste trabalho consumiu apenas 25 minutos, obtendo um ganho de, aproximadamente, 61,5% com relação ao tempo de geração manual do teste. A aluna também destacou que a utilização da tabela de decisão possibilitou uma melhor visualização dos casos de teste e que a geração automática dos testes a partir da tabela de decisão fez com que o código a ser produzido manualmente exigisse um esforço mínimo (3 minutos segundo a tabela acima).

5.2.2.3. Comparação com a abordagem *"capture and replay"*

A abordagem para a geração semi-automática de testes apresentada neste trabalho e a abordagem "*capture and replay*" (seção 2.2) foram comparadas quanto ao tempo gasto para a geração de testes. Essa comparação deu-se através

dos testes gerados para duas funcionalidades do sistema: “Cadastrar grupo” e “Cadastrar local”. A ferramenta escolhida para gerar os casos de teste a partir de “*capture and replay*” foi o Selenium IDE. Para assegurar que o número de casos de teste gerados com o Selenium IDE fosse o mesmo que a quantidade de casos de teste gerados com as ferramentas deste trabalho, já que, nesta comparação, estamos apenas interessados no tempo consumido, utilizamos uma tabela de decisão escrita manualmente para visualizar todos os casos de teste necessários. Assim, o tempo de escrita dessa tabela de decisão foi contabilizado no tempo total gasto para a geração dos testes com “*capture and replay*” para determinada funcionalidade.

A geração dos scripts de teste com Selenium IDE, então, engloba os quatro passos seguinte:

1. **Escrita da tabela de decisão**: para identificação dos casos de teste;
2. **Gravação**: gravação das ações do usuário realizando manualmente todos os casos de teste;
3. **Refatoração do código**: todos os passos foram gravados em uma única execução do Selenium IDE, o que necessitou que o código gerado fosse refatorado para atribuir uma função a cada caso de teste e uma função inicial que prepara o ambiente para cada caso de teste (Todos estes passos encontravam-se em apenas uma única função no código oriundo da gravação). A gravação de cada caso de teste em diferentes execuções do Selenium IDE teria diminuído o tempo necessário neste passo, mas teria aumentado significativamente o tempo gasto no passo anterior, de forma que esta, na opinião do autor, é a forma mais rápida para geração dos testes, justificando, assim, sua escolha;
4. **Inserção de oráculos**: mesmo com “*capture and replay*” ainda é necessário inserir as chamadas para as funções que exercem o papel de oráculos (ou assertivas).

Com as ferramentas propostas neste trabalho, para a geração dos casos de teste para as duas funcionalidades citadas, foi necessário apenas:

1. Montagem da tabela de decisão apoiada por ferramenta, que gerará os casos de teste semânticos;

2. Execução de outra ferramenta para a geração dos scripts de testes executáveis a partir dos casos de teste semânticos;
3. Inserção do pacote Java da classe gerada para os testes;
4. Escrita do código necessário para implementar os métodos utilizados como oráculos.

Como, nas duas abordagens, foi necessário inserir o pacote Java da classe e codificar os métodos usados como oráculos, o tempo para estas tarefas não foi contabilizado para fins de comparação. A geração dos scripts de testes executáveis a partir dos casos de teste semânticos consiste apenas em rodar outra ferramenta passando como argumento o XML com a descrição dos casos de teste. Essa tarefa é executada em questões de segundos (para escolher o arquivo) e praticamente instantânea na execução do algoritmo, assim o seu tempo não foi considerado para fins de comparação. Dessa forma, para contabilizar o tempo gasto para a geração dos testes com a abordagem presente neste trabalho foi contabilizado apenas o tempo necessário para a geração da tabela de decisão apoiada por ferramenta, visto que não foi preciso nenhuma refatoração posterior do código produzido.

A tabela abaixo demonstra os resultados dos tempos consumidos para a geração dos testes para as funcionalidades “Cadastrar grupo” e “Cadastrar local” com as duas diferentes abordagens.

Selenium IDE (*capture and replay*)	Funcionalidade Cadastrar grupo	Funcionalidade Cadastrar local
Escrita da tabela de decisão	6min	17min
Gravação	5,5min	7min
Refatoração do código	12min	17,5min
Inserção de chamadas a oráculos	9min	15min
Tempo total	32,5min	56,5min
Tabela de decisão		
Geração da tabela de decisão apoiada por ferramenta (casos de teste semânticos)	18min	33min
Tempo total	18min	33min

Tabela 8 – Comparação com tempo gasto para geração com *capture and replay*.

Como podemos ver a partir dos dados da tabela acima, o ganho de tempo na geração de testes com a metodologia presente neste trabalho em relação ao tempo gasto para a geração de testes utilizando "*capture and replay*" foi de, aproximadamente, 44,6% para a primeira funcionalidade e 41,6% para a segunda funcionalidade. Poder-se-ia questionar a necessidade do uso de uma tabela de decisão para identificar os testes necessários na abordagem com "*capture and replay*" visto que seu uso não é obrigatório para a geração dos testes. Descontando-se o tempo gasto na escrita da tabela de decisão para os testes com o Selenium IDE, teríamos ainda, um ganho aproximado de 32,1% na primeira funcionalidade e de 16,5% na segunda. Porém, também teríamos mais chance de não ter contemplado alguns casos de teste (Myers, 2004).

5.2.2.4. Comparação com testes produzidos por desenvolvimento dirigido por comportamento

Para fins de comparação, foram feitos testes utilizando-se a abordagem DDC por meio da ferramenta JBehave (JBehave, 2008). Nessa ferramenta, como explicado na seção 2.1.1.1, a especificação de testes consiste em um arquivo texto contendo um ou mais cenários, escritos em linguagem natural semi-estruturada, que servem de roteiro para os testes. Os passos destes cenários devem ser mapeados em funções presentes em uma classe Java. Estas funções devem ser codificadas manualmente. A ferramenta JBehave, quando da execução dos testes, interpreta os passos dos cenários invocando as funções mapeadas para cada passo.

Os testes implementados para essa comparação foram os das funcionalidades "Cadastrar grupo" e "Cadastrar local". Para elicitar todos os casos de teste necessários, foi utilizada a mesma tabela de decisão escrita manualmente para os testes da seção anterior. Assim, o tempo para sua escrita também foi contabilizado para obtermos o tempo total. Também foi medido o tempo necessário para a escrita do arquivo de texto contendo os cenários e do código necessário para implementar os passos descritos nos cenários. O tempo total foi comparado com o tempo de escrita da tabela de decisão apoiada por ferramenta, já que após, a geração automática do script, não foi preciso nenhuma refatoração de código. O tempo necessário para a escrita dos oráculos não foi contabilizado em nenhuma das abordagens, pois as funções utilizadas como oráculos foram as mesmas, e o tempo necessário para escrevê-las pôde ser deduzido da equação.

JBehave (desenvolvimento dirigido por comportamento)	Funcionalidade Cadastrar grupo	Funcionalidade Cadastrar local
Escrita da tabela de decisão	6min	17min
Escrita de código de teste e arquivo texto de cenários	40min	46min
Tempo total	46min	63min
Tabela de decisão		
Geração da tabela de decisão apoiada por ferramenta (casos de teste semânticos)	18min	33min
Tempo total	18min	33min

Tabela 9 - Comparação com tempo gasto para geração com *JBehave*.

Na tabela acima podemos ver que a geração dos testes com JBehave para as funcionalidades “Cadastrar grupo” e “Cadastrar local” consumiram, respectivamente, 46 minutos e 63 minutos. Cada funcionalidade deu origem a uma classe Java que continha os métodos necessários para implementar os passos descritos no arquivo de cenários. Para a escrita dos testes da funcionalidade “Cadastrar local” houve uma refatoração e reaproveitamento de boa parte do código escrito para os testes de “Cadastrar grupo”, o que explica o tempo quase igual para a escrita do código de testes e cenários entre as duas funcionalidades, embora “Cadastrar local” tenha o dobro de casos de teste em comparação com “Cadastrar grupo”.

Com base nos resultados apresentados na tabela acima, podemos ver que a geração de testes a partir das ferramentas presentes neste trabalho obteve um ganho, com relação ao tempo para a geração dos testes de, aproximadamente, 60,9% para “Cadastrar grupo” e 47,7% para “Cadastrar local”. Se descontarmos o tempo para a escrita da tabela de decisão para os testes com JBehave, visto que não são obrigatórias, ainda obteríamos um ganho de, aproximadamente, 55% para “Cadastrar grupo” e 28,3% para “Cadastrar local”. A justificativa para o uso de uma tabela de decisão para a elicitação dos casos de teste para JBehave é mesma apresentada na seção anterior.

Com JBehave, além do código de teste, também temos como produto uma especificação do comportamento do sistema na forma de cenários. Poderíamos ter

especificação semelhante com uma descrição de caso de uso. Para comparação do esforço entre escrever os cenários e os casos de uso como proposto nesta dissertação, foram contabilizados os tempos de escrita das seguintes seções de caso de uso para as duas funcionalidades testadas: pré-condição, fluxo principal, fluxos alternativos e regras de negócio. Apenas essas seções foram escolhidas, pois elas contêm as informações que estavam representadas nos cenários gerados. A tabela abaixo contém os tempos da tabela anterior atualizados com o esforço para essas seções dos casos de uso.

JBehave (desenvolvimento dirigido por comportamento)	Funcionalidade Cadastrar grupo	Funcionalidade Cadastrar local
Escrita da tabela de decisão	6min	17min
Escrita de código de teste e arquivo texto de cenários	40min	46min
Tempo total	46min	63min
Tabela de decisão		
Redação do formulário de caso de uso	7min	7min
Geração da tabela de decisão apoiada por ferramenta (casos de teste semânticos)	18min	33min
Tempo total	25min	40min

Tabela 10 – Segunda comparação com tempo gasto para geração com *JBehave*.

Conforme os dados da tabela acima, ainda teríamos um ganho de, aproximadamente, 55,7% para “Cadastrar grupo” e 36,5% para “Cadastrar local” com relação ao tempo gasto. Se descontarmos o tempo para a escrita da tabela de decisão para os testes com JBehave, ainda ficaríamos com um ganho de, aproximadamente, 37,5% para “Cadastrar grupo” e 13% para “Cadastrar local”.

5.2.2.5. Dificuldades encontradas

Um problema encontrado diz respeito ao número de casos de uso que podem ser pré-condições para outro caso de uso. Como se pode ver na figura 41, o caso de uso “Cadastrar local” tem dois casos de uso como pré-requisito:

"Cadastrar tipo de local" e "Listar local". Isto ocorre porque um **Local** precisa de um **Tipo de Local** associado, cadastrado previamente. Além disso, para cadastrar um **Local**, precisamos antes listar os Locais disponíveis. Entretanto, a ferramenta desenvolvida para a edição de tabelas só prevê um caso de uso como pré-condição para os testes que estão sendo gerados. Assim, para contornar esta limitação, ao gerar os testes para o caso de uso "Cadastrar local", consideramos que já temos cadastrado previamente um **Tipo de Local** e atribuímos como pré-condição ao cadastro de **Local** apenas o caso de uso "Listar local". Para garantir que, ao tentar cadastrar um **Local**, já temos um **Tipo de Local** cadastrado, basta que os testes para **Tipo de Local**, que irão cadastrar com sucesso pelo menos um **Tipo de Local**, executem antes dos testes para cadastros de **Locais** e que nem todos os **Tipos de Locais** tenham sido deletados num eventual teste de remoção. Outra solução seria incluir manualmente, nos testes gerados para o caso de uso "Cadastro de local", o código necessário para o cadastro de **Tipo de Local**. Como neste estudo de caso os testes foram gerados no padrão JUnit (JUnit, 2009), esse código poderia ser incluído no método com anotação **@BeforeClass**, que executa antes de todos os testes presentes numa classe, e apenas uma única vez.

Outra dificuldade é o fato do número de regras em uma tabela de decisão crescer exponencialmente em função do número de condições existentes. Uma solução para muitos casos é o uso de Grupos Condicionais (seção 4.1.1.1), que visa descartar combinações de valores de condições impossíveis de serem satisfeitas ao mesmo tempo. Entretanto, nem sempre o seu emprego é possível, pois algumas condições podem não ter restrições entre si. Por exemplo, um formulário consistindo de cinco campos que devem ser preenchidos pelo usuário, conterá pelos menos cinco condições do tipo "usuário preenche valor para campo <campo>". Essas cinco condições, não apresentando nenhuma restrição entre si, resultarão em uma tabela de 32 regras. Adicionando-se mais um campo independente ao formulário, temos 64 regras. Admitindo-se que o formulário possui um botão "OK" para ser submetido e um botão "Cancelar" para cancelar o preenchimento, e que estes dois são mutuamente exclusivos obrigatórios (seção 4.1.1.1) entre si, tem-se então 128 regras (64 regras onde o usuário clica em "OK" e 64 regras onde o usuário clica em "Cancelar"). Uma tabela deste tamanho torna-se difícil de preencher. Uma solução seria particioná-la, por exemplo, em duas tabelas de 64 colunas, cada uma com um dos botões mencionados sendo clicados

no final. Ainda assim, mesmo que divididas em duas tabelas, o usuário ainda teria que montar todas as 128 regras. Uma solução para diminuir o esforço exigido seria, dadas as condições e seus Grupos Condicionais, gerar automaticamente todas as combinações de valores para as regras em termos de valores “V” (verdadeiro), “F” (Falso) e “---” (Indiferente). Entretanto, na implementação atual, o usuário ainda teria que marcar as ações associadas a cada regra e preencher as informações necessárias para campos de entrada de dados alfanuméricos. Uma solução para evitar que o usuário precisasse preencher a informação de entrada para campos alfanuméricos seria existir a opção de carregar esses dados a partir de um arquivo que conteria, para cada condição de interesse, o dado para que a condição fosse verdadeira e o dado cujo valor tornaria a condição falsa. Todavia, os dados de regras preenchidas a partir deste arquivo ainda poderiam ser editados via interface do editor de tabelas de decisão, assim como é feito atualmente, caso deseja-se mudar algum valor para determinada regra.

Outro ponto importante a ser considerado é que o sistema alvo dos testes deve ser desenvolvido de forma que possibilite que a ferramenta responsável pela automação da interação com o navegador seja capaz de identificar os elementos a serem exercitados. Para essa identificação podem ser usadas diversas estratégias como a ordem de ocorrência do elemento no HTML e seu tipo ou formas mais fáceis, como a atribuição de um identificador único (atributo “id” no caso de elementos HTML). No caso de entidades cadastradas dinamicamente e que precisam ser identificadas pela ferramenta de automação, pode-se utilizar estratégias como acrescentar ao atributo “id” do elemento alguma informação preenchida no formulário anterior usada para identificar esta entidade. Essas alterações simples podem facilitar bastante a geração dos testes, e, portanto, o desenvolvedor do sistema deve ter esta questão em mente ao implementar o sistema.