UNIVERSIDADE DE SÃO PAULO

Instituto de Ciências Matemáticas e de Computação

Estudo e Definição de Teste de Software para o contexto de Sistemas Multi-tenants

Victor Hugo Santiago Costa Pinto

Tese de Doutorado do Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional (PPG-CCMC)

SERVIÇO DE PÓS-GRADUAÇÃO DO ICMC-USP

Data de Depósito:

Assinatura: _____

Victor Hugo Santiago Costa Pinto

Estudo e Definição de Teste de Software para o contexto de Sistemas Multi-tenants

Tese apresentada ao Instituto de Ciências Matemáticas e de Computação – ICMC-USP, como parte dos requisitos para obtenção do título de Doutor em Ciências – Ciências de Computação e Matemática Computacional. *EXEMPLAR DE DEFESA*

Área de Concentração: Ciências de Computação e Matemática Computacional

Orientadora: Profa. Dra. Simone do Rocio Senger de Souza

USP – São Carlos
Julho de 2019

Ficha catalográfica elaborada pela Biblioteca Prof. Achille Bassi
e Seção Técnica de Informática, ICMC/USP,
com os dados inseridos pelo(a) autor(a)

S659e Santiago Costa Pinto, Victor Hugo
 Estudo e Definição de Teste de Software para o
 contexto de Sistemas Multi-tenants / Victor Hugo
 Santiago Costa Pinto; orientador Simone do Rocio
 Senger de Souza. -- São Carlos, 2019.
 226 p.

 Tese (Doutorado - Programa de Pós-Graduação em
 Ciências de Computação e Matemática Computacional) --
 Instituto de Ciências Matemáticas e de Computação,
 Universidade de São Paulo, 2019.

 1. Teste de Software. 2. Computação em Nuvem. 3.
 Multi-tenancy. 4. Critérios de teste. I. do Rocio
 Senger de Souza, Simone, orient. II. Título.

Victor Hugo Santiago Costa Pinto

**Study and Definition of Software Testing for the context of
Multi-tenant Systems**

Thesis submitted to the Institute of Mathematics
and Computer Sciences – ICMC-USP – in
accordance with the requirements of the Computer
and Mathematical Sciences Graduate Program, for
the degree of Doctor in Science. *EXAMINATION
BOARD PRESENTATION COPY*

Concentration Area: Computer Science and
Computational Mathematics

Advisor: Profa. Dra. Simone do Rocio Senger
de Souza

USP – São Carlos
July 2019

Dedico este trabalho à minha filha Nicole, à minha esposa Paula, minha mãe Cineide e às minhas irmãs Yasmim e Emanuella que contribuíram de modo especial para o meu crescimento como ser humano e como profissional. Amo todas vocês!

AGRADECIMENTOS

Cinco anos se passaram nessa jornada e muitas coisas alegres aconteceram; casei, me tornei pai de uma filha linda, mudei para Lavras etc. Muitos acontecimentos marcantes, diversas histórias para contar e muitas amizades conquistadas que me deram muita força para chegar nessa etapa. Um período de crescimento profissional no qual não vou encarar com tristeza ou saudade, mas como uma nova vida que pretendo conduzir com o mesmo empenho, teimosia, alegria e vontade com que despendi durante o doutorado.

Gostaria de agradecer a Deus por me amparar nos momentos difíceis que passei durante o doutorado e também por me conceder tantos momentos felizes com minha família e amigos. Até aqui Ele nunca me abandonou, todas as vezes que eu acreditava que as portas não iriam abrir, as oportunidades surgiam.

Um agradecimento especial a minha mãe. O simples fato de você existir me fortalece o suficiente para continuar lutando, você é o meu maior exemplo. O término de um trabalho como este sem dúvida não seria possível sem o apoio emocional que ela me concede. Apesar da distância, suas palavras de ânimo foram muito importantes. O sonho desse doutorado nasceu primeiro com ela, sozinha mas tinha que escolher um nome para um filho doutor e passado alguns anos, acabei indo para a cidade nº 1 em doutores por habitante. Mesmo com poucas condições, se empenhou em me oferecer a maior riqueza que se pode dar a um filho: o incentivo aos estudos.

Minha esposa também merece grande parte desta conquista. Conviver com um doutorando em Ciência da Computação não é fácil, ainda mais quando ele fica preocupado demais com o futuro. Foram incontáveis finais de semana perdidos e trancafiados em casa. Paula, parte deste título de doutorado também é seu.

Outra pessoa que merece um agradecimento especial é a Professora Simone, uma pessoa sensacional. As reuniões que tivemos foram sempre produtivas e me motivaram ao longo dessa caminhada. Aprendi com você que as pesquisas são fundamentais na carreira acadêmica, mas existem muitas conquistas ao longo do doutorado que não se resumem somente as publicações. As amizades e as discussões com o grupo permitem uma visão mais ampla, crítica e construtiva, importantes quando eu estiver exercendo o papel de professor e pesquisador, orientando os alunos e ajudando-os em seus sonhos.

Aos grandes amigos do LabES: Ricardo Ramos e Ricardo Vilela pelas discussões que tivemos no decorrer do doutorado. Agradeço também aos pais do Ricardo Ramos: Timóteo e

Dona Luzia pelo carinho em me receber tantas vezes em São Carlos. Agradeço imensamente o incentivo do meu amigo Rinaldo Michelão, em que nos momentos mais difíceis e solitários me convidava para almoçar junto com sua família nos finais de semana. Isso fez muita diferença, espero um dia retribuir. Agradeço a Márcia por brincar e cuidar da minha filha Nicole quando mais precisei me ausentar. Agradeço aos amigos de Lavras, pelos churrascos e pelas conversas que sempre me ajudaram a esfriar a cabeça: André Pimenta, Evelise, Jadson, Valéria, Mayron e Andreza.

Agradeço a todos os funcionários de ICMC que de forma direta ou indireta contribuíram para a realização desta tese de doutorado. Vocês são exemplos para qualquer universidade, sempre empenhados em atender da melhor forma os alunos.

Agradeço à Capes pelo apoio financeiro.

*“(...) Até aqui nos ajudou o Senhor.”
(1 Samuel 7:12)*

RESUMO

PINTO, V. H. S. C. **Estudo e Definição de Teste de Software para o contexto de Sistemas Multi-tenants**. 2019. 226 p. Tese (Doutorado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2019.

Multi-tenancy é um padrão arquitetural para SaaS (Software as a Service) pois permite que múltiplas organizações, ou tenants, tenham sua própria versão da aplicação executando e coexistindo em espaços virtualizados e isolados no mesmo hardware, reduzindo custos operacionais. Uma única instância com meta-dados configuráveis pode oferecer aos usuários finais, interfaces e funcionalidades alinhadas às regras de negócio diferentes para cada tenant. Entretanto, isso representa um desafio para a atividade de teste, uma vez que a execução das versões incluem fluxos de controle e dados em funcionalidades compartilhadas que podem ser afetadas pelo comportamento não determinístico, uma dificuldade conhecida na área de testes de programas concorrentes. Apesar da importância de desenvolver esses sistemas com alta qualidade, os modelos de implementação são fragmentados e não seguem uma metodologia padronizada, falta uma arquitetura de referência e abordagens de teste direcionadas às particularidades desse modelo de aplicações. Embora o modelo SaaS seja provavelmente o mais adotado no contexto de Computação em Nuvem e que multi-tenancy seja o mecanismo chave para isso, existe muita desconfiança em relação à segurança dos dados e a capacidade de uma única instância em execução não ser afetada por defeitos em versões isoladas. Este projeto de doutorado contribui nessa direção, propondo uma abordagem de teste que permite explorar defeitos relacionados ao atendimento às requisições e redirecionamentos que ocorrem nessas versões, uma vez que tais operações implicam em definições e usos subsequentes de uma variável de identificação dos tenants e seus recursos. Uma taxonomia preliminar de defeitos foi proposta, um sistema de e-commerce chamado *MtShop* foi desenvolvido seguindo práticas recentes, estudos experimentais foram conduzidos envolvendo testes de aceitação automatizados com execução paralela, um modelo de teste foi definido para apoiar a definição de um conjunto de critérios de cobertura e a extração de elementos requeridos pelos testes e estudos foram realizados para avaliar a aplicabilidade dos critérios propostos. Os resultados dos estudos exploratórios demonstram que a abordagem proposta é aplicável e o modelo de teste é capaz de representar aplicações multi-tenants. Os critérios podem contribuir para a qualidade desse tipo de sistemas em termos de cobertura e identificação de defeitos que possuem natureza concorrente.

Palavras-chave: Teste de software, Multi-tenancy, Computação em nuvem, Critérios de teste.

ABSTRACT

PINTO, V. H. S. C. **Study and Definition of Software Testing for the context of Multi-tenant Systems**. 2019. 226 p. Tese (Doutorado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2019.

Multi-tenancy is an architectural pattern for Software as a Service (SaaS) because it allows multiple organizations, so-called tenants, to have their own version of the application running and coexisting in virtual and isolated spaces on the same hardware, reducing operating costs. A single instance with configurable metadata can provide to end-users, different interfaces and features targeted to the business rules for each tenant. However, this represents a challenge for testing activity, once the execution of the versions includes control and data flow in shared functionality that may be affected by non-deterministic behavior, a known difficulty in the testing of concurrent programs. Despite the importance of developing high-quality cloud applications, implementation models are fragmented and do not follow a standardized methodology, a reference architecture and testing approaches to address the particularities of this application model are lacking. Although the SaaS model is probably the most widely adopted in the context of cloud computing and multi-tenancy is the key mechanism for this, there is mistrust about data security and the fact of a single instance running not being affected by faults from isolated versions. This project contributes in this direction and it proposes a testing approach that allows identifying faults related to the fulfillment of requests and redirects that occur in these versions, since such operations involve the definition and subsequent uses of a variable for tenant identification, including its resources. A preliminary fault taxonomy for multi-tenant SaaS systems was proposed, an e-commerce system called *MtShop* was developed following recent practices, experimental studies were conducted involving parallel and automated acceptance tests, a testing model was defined to support the definition of a set of coverage criteria and extraction of required elements by the tests and studies were performed to evaluate the applicability of the proposed testing criteria. The exploratory case studies indicate that the proposed approach is applicable and the testing model can represent multi-tenant applications. The proposed criteria can contribute to the quality of such systems in terms of coverage and reveal faults with concurrent nature.

Keywords: Software testing, Multi-tenancy, Cloud computing, Testing criteria.

LISTA DE ILUSTRAÇÕES

Figura 1 – Cenário típico da atividade de teste.	37
Figura 2 – Papéis e modelos de serviços na computação em nuvem.	51
Figura 3 – Níveis de maturidade para um SaaS.	53
Figura 4 – Diagrama entidade relacionamento para o <i>MtShop</i>	62
Figura 5 – As principais camadas do <i>MtShop</i>	63
Figura 6 – Configurações de conexão para o tenant 1 no <i>MtShop</i>	64
Figura 7 – Páginas iniciais das três lojas (versões) configuradas no <i>MtShop</i>	64
Figura 8 – Página de lista de produtos disponíveis em uma das versões do <i>MtShop</i> . . .	65
Figura 9 – Página de visualização de dados de produtos em uma das versões do <i>MtShop</i> . .	65
Figura 10 – Página do carrinho de compras em uma das versões do <i>MtShop</i>	66
Figura 11 – Página de confirmação do Pedido em uma das versões do <i>MtShop</i>	66
Figura 12 – Página de confirmação da compra em uma das versões do <i>MtShop</i>	67
Figura 13 – Página de registro de usuários em uma das versões do <i>MtShop</i>	67
Figura 14 – Página de login em uma das versões do <i>MtShop</i>	68
Figura 15 – Página de gerenciamento de produtos (Administrador) em uma das versões do <i>MtShop</i>	68
Figura 16 – String de busca.	82
Figura 17 – Distribuição de artigos (fase de execução).	84
Figura 18 – Distribuição dos estudos primários de acordo com as categorias.	85
Figura 19 – Quantidade de artigos selecionados a partir de workshops, journals e confe- rências.	85
Figura 20 – Diagrama de entidade relacionamento do <i>iCardapio</i>	107
Figura 21 – <i>Box plot</i> para os defeitos identificados com um crescente número de versões. .	113
Figura 22 – Um teste de normalidade para os dados coletados quando as versões do <i>iCardapio</i> foram submetidas ao teste automatizado.	113
Figura 23 – Teste de normalidade quando 2, 5 e 10 versões foram submetidas ao teste automatizado.	114
Figura 24 – Grafo de efeitos para o cenário de teste multi-tenant.	115
Figura 25 – Resultados da execução do teste automatizado e paralelo que verifica o campo <i>name</i> dos produtos em duas versões do <i>iCardapio</i>	116
Figura 26 – Distribuição das maiores ocorrências de defeitos por execução e seus tipos para <i>iCardapio</i> e <i>MtShop</i>	124

Figura 27 – <i>Box plots</i> para o número de defeitos em média para o <i>iCardapio</i> (A) e o <i>MtShop</i> (B) nas três execuções.	126
Figura 28 – $PCFG_{MT}$ para a aplicação <i>App</i>	136
Figura 29 – Uma visão arquitetural da estratégia de teste.	141
Figura 30 – $PCFG_{MT}$ para a aplicação <i>MtShopFails</i>	149
Figura 31 – $PCFG_{MT}$ para a aplicação <i>MtShopSafe</i>	154
Figura 32 – $PCFG_{MT}$ para a aplicação <i>iCardapio</i>	158

LISTA DE QUADROS

Quadro 1 – Diferenças entre sistemas SaaS e aplicações web tradicionais sob o ponto de vista técnico.	59
Quadro 2 – Diferenças entre sistemas SaaS e aplicações web tradicionais sob o ponto de vista dos negócios.	60
Quadro 3 – Testes utilizados em aplicações na nuvem.	77
Quadro 4 – Visão comparativa dos principais tipos de teste para aplicações na nuvem. .	79

LISTA DE CÓDIGOS-FONTE

Código-fonte 1 – Trecho de código da classe <i>MultiTenancyInterceptor</i>	69
Código-fonte 2 – Trecho de código da classe <i>TenantContextHolder</i>	70
Código-fonte 3 – Trecho de código da classe <i>DataSourceBasedMultiTenantConection-ProviderImpl</i>	71
Código-fonte 4 – Trecho de código da classe <i>CartController</i>	72
Código-fonte 5 – Trecho de código da classe <i>CustomAuthenticationToken</i>	73
Código-fonte 6 – Trecho de código da classe <i>MultiTenancyInterceptor.java</i>	181
Código-fonte 7 – Trecho de código da classe <i>AdminProduct.java</i>	181
Código-fonte 8 – Trecho de código da classe <i>RegisterController.java</i>	184
Código-fonte 9 – Trecho de código da classe <i>OrderController.java</i>	185
Código-fonte 10 – Trecho de código da classe <i>AdminHome.java</i>	186
Código-fonte 11 – Trecho de código da classe <i>ProductController.java</i>	187
Código-fonte 12 – Trecho de código da classe <i>HomeController.java</i>	187
Código-fonte 13 – Trecho de código da classe <i>CartController.java</i>	188
Código-fonte 14 – Trecho de código da página <i>cart.jsp</i>	189
Código-fonte 15 – Trecho de código da página <i>viewproduct.jsp</i>	189
Código-fonte 16 – Trecho de código da página <i>header.jsp</i>	190
Código-fonte 17 – Trecho de código da classe <i>AdminHome.java</i>	191
Código-fonte 18 – Trecho de código da classe <i>AdminProduct.java</i>	192
Código-fonte 19 – Trecho de código da classe <i>CartController.java</i>	194
Código-fonte 20 – Trecho de código da classe <i>CustomAuthenticationFilter.java</i>	195
Código-fonte 21 – Trecho de código da classe <i>CustomAuthenticationToken.java</i>	196
Código-fonte 22 – Trecho de código da classe <i>CustomUserDetails.java</i>	196
Código-fonte 23 – Trecho de código da classe <i>HomeController.java</i>	197
Código-fonte 24 – Trecho de código da classe <i>MultiTenancyInterceptor.java</i>	197
Código-fonte 25 – Trecho de código da classe <i>OrderController.java</i>	198
Código-fonte 26 – Trecho de código da classe <i>ProductController.java</i>	199
Código-fonte 27 – Trecho de código da classe <i>RegisterController.java</i>	200
Código-fonte 28 – Trecho de código da classe <i>ICardapioTenantResolver.java</i>	203
Código-fonte 29 – Trecho de código da classe <i>RestaurantController.java</i>	204
Código-fonte 30 – Trecho de código da interface <i>CurrentTenantResolver.java</i>	205
Código-fonte 31 – Trecho de código da classe <i>Product.java</i>	206
Código-fonte 32 – Trecho de código da classe <i>Restaurant.java</i>	207

Código-fonte 33 – Trecho de código da classe <i>RestaurantFacade.java</i>	209
Código-fonte 34 – Trecho de código da classe <i>MultiTenantSimpleJpaRepository.java</i> . .	211

LISTA DE TABELAS

Tabela 1	– Informações sobre o <i>MtShop</i> coletadas por meio da SonarQube.	62
Tabela 2	– Lista de artigos chave usados para calibrar a string de busca.	82
Tabela 3	– Elementos do formulário de extração de dados.	83
Tabela 4	– Principais questões relacionadas com o desenvolvimento de sistemas SaaS multi-tenants.	86
Tabela 5	– Principais questões identificadas em relação à garantia de qualidade.	88
Tabela 6	– Classificação de defeitos para programas concorrentes.	95
Tabela 7	– Taxonomia preliminar de defeitos para sistemas SaaS multi-tenants.	97
Tabela 8	– Subconjunto de erros e defeitos identificados a partir da taxonomia.	101
Tabela 9	– Informações sobre o <i>iCardapio</i> coletadas pela SonarQube.	107
Tabela 10	– Dados coletados com os testes automatizados.	112
Tabela 11	– Exemplo do formulário utilizado para a coleta de dados.	120
Tabela 12	– Dados coletados com a execução dos testes para as versões do <i>iCardapio</i>	122
Tabela 13	– Dados coletados com a execução dos testes para as versões do <i>MtShop</i>	123
Tabela 14	– Informações obtidas pelo modelo de teste para a aplicação <i>App</i>	138
Tabela 15	– Elementos requeridos pelos critérios de teste em relação ao sistema <i>App</i>	140
Tabela 16	– Cenários de teste em relação as versões <i>MtShopFails</i> e <i>MtShopSafe</i>	148
Tabela 17	– Conjuntos obtidos pela aplicação do modelo de teste para a versão <i>MtShopFails</i>	150
Tabela 18	– Elementos requeridos pelos critérios de teste em relação a versão <i>MtShopFails</i>	151
Tabela 19	– Cobertura alcançada com os critérios de teste em relação a versão <i>MtShopFails</i>	152
Tabela 20	– Defeitos identificados na versão <i>MtShopFails</i>	153
Tabela 21	– Conjuntos obtidos pela aplicação do modelo de teste para a versão <i>MtShopSafe</i>	155
Tabela 22	– Elementos requeridos pelos critérios de teste em relação a versão <i>MtShopSafe</i>	156
Tabela 23	– Cobertura alcançada com os critérios de teste em relação a versão <i>MtShopSafe</i>	156
Tabela 24	– Defeitos identificados na versão <i>MtShopSafe</i>	157
Tabela 25	– Cenários de teste em relação ao <i>iCardapio</i>	157
Tabela 26	– Conjuntos obtidos pela aplicação do modelo de teste para a versão <i>iCardapio</i>	159
Tabela 27	– Elementos requeridos pelos critérios de teste em relação a versão <i>iCardapio</i>	159
Tabela 28	– Defeitos identificados na versão <i>iCardapio</i>	159
Tabela 29	– Cobertura alcançada com os critérios de teste em relação a versão <i>iCardapio</i>	160
Tabela 30	– Casos de teste elaborados com os critérios EPC e BVA para o <i>MtShopFails</i>	215
Tabela 31	– Defeitos identificados no <i>MtShopFails</i>	218
Tabela 32	– Casos de teste elaborados com os critérios EPC e BVA para o <i>iCardapio</i>	223

Tabela 33 – Defeitos identificados no *iCardapio*. 225

SUMÁRIO

1	INTRODUÇÃO	27
1.1	Contexto	27
1.2	Motivação	30
1.3	Objetivos	30
1.3.1	<i>Objetivo geral</i>	30
1.3.2	<i>Objetivos específicos</i>	30
1.4	Metodologia de pesquisa	31
1.5	Contribuições	31
1.6	Organização da tese	32
2	FUNDAMENTOS DE TESTE DE SOFTWARE	35
2.1	Considerações iniciais	35
2.2	Teste de Software	36
2.2.1	<i>Técnica funcional</i>	38
2.2.2	<i>Técnica baseada em defeitos</i>	39
2.2.3	<i>Técnica estrutural</i>	40
2.2.4	<i>Automatização de testes de aceitação do usuário</i>	43
2.3	Considerações finais	45
3	COMPUTAÇÃO EM NUVEM E SISTEMAS SAAS MULTI-TENANTS	47
3.1	Considerações iniciais	47
3.2	Computação em Nuvem	47
3.2.1	<i>Modelos de entrega de serviços</i>	49
3.3	SaaS (Software as a Service) e Multi-tenancy	51
3.3.1	<i>Elementos fundamentais no desenvolvimento</i>	55
3.3.1.1	<i>Aplicações web e SOA (Service-Oriented Architecture) versus SaaS</i>	58
3.4	Desenvolvimento de um sistema SaaS multi-tenant	61
3.4.0.1	<i>Interfaces e funcionalidades</i>	64
3.4.0.2	<i>Trechos de código representativos</i>	68
3.5	Considerações finais	74
4	TESTE DE SOFTWARE APLICADO A SISTEMAS SAAS MULTI-TENANTS E UMA CLASSIFICAÇÃO PRELIMINAR DE DEFEITOS	75
4.1	Considerações iniciais	75

4.2	Teste de Software aplicado a sistemas SaaS multi-tenants	76
4.3	Mapeamento Sistemático da Literatura	81
4.3.1	<i>Planejamento</i>	81
4.3.2	<i>Execução</i>	83
4.3.3	<i>Discussão dos resultados</i>	84
4.3.3.0.1	Desenvolvimento	85
4.3.3.0.2	Desempenho	86
4.3.3.0.3	Customização	87
4.3.3.0.4	Garantia de Qualidade	87
4.3.3.0.5	Banco de dados	88
4.3.4	<i>Desafios do teste para sistemas SaaS multi-tenants</i>	90
4.4	Trabalhos relacionados	92
4.5	Comparação entre programas concorrentes e sistemas multi-tenants	94
4.6	Defeitos em sistemas SaaS multi-tenants	96
4.7	Considerações finais	102
5	ESTUDOS PRELIMINARES ENVOLVENDO A TÉCNICA FUNCIONAL DE TESTE	105
5.1	Considerações iniciais	105
5.2	Avaliação do teste de aceitação do usuário para sistemas SaaS multi-tenants	106
5.2.1	<i>Planejamento</i>	108
5.2.1.1	<i>Configuração do experimento</i>	108
5.2.1.2	<i>Formulação das hipóteses</i>	109
5.2.1.3	<i>Seleção das variáveis</i>	110
5.2.2	<i>Execução</i>	110
5.2.2.1	<i>Preparação</i>	110
5.2.3	<i>Análise dos dados</i>	111
5.2.3.1	<i>Estatística descritiva</i>	111
5.2.3.2	<i>Teste de hipóteses</i>	112
5.2.3.3	<i>Discussão sobre a natureza dos defeitos identificados</i>	115
5.2.4	<i>Ameaças à validade</i>	116
5.3	Investigação do teste automatizado envolvendo sistemas multi-tenants com diferentes arquiteturas de dados	117
5.3.1	<i>Ferramentas e aplicações utilizadas no experimento</i>	118
5.3.2	<i>Seleção de variáveis</i>	119
5.3.3	<i>Execução</i>	119
5.3.3.1	<i>Preparação</i>	119
5.3.4	<i>Análise de Dados</i>	120
5.3.4.1	<i>Estatística Descritiva</i>	121

5.3.5	<i>Ameaças à validade</i>	127
5.4	Considerações finais	128
6	TESTE ESTRUTURAL APLICADO A SISTEMAS MULTI-TENANTS	131
6.1	Considerações iniciais	131
6.2	Modelo de teste e critérios estruturais	132
6.2.1	<i>Critérios de cobertura</i>	137
6.3	Abordagem de teste proposta	139
6.4	Considerações finais	142
7	AVALIAÇÃO DA APLICABILIDADE DOS CRITÉRIOS PROPOSTOS	145
7.1	Considerações iniciais	145
7.2	Estudos de Caso	146
7.2.1	<i>MtShopFails</i>	147
7.2.2	<i>MtShopSafe</i>	153
7.2.3	<i>iCardapio</i>	156
7.3	Considerações finais	161
8	CONCLUSÕES	163
8.1	Contribuições e limitações da pesquisa	165
8.2	Lições aprendidas	166
8.3	Oportunidades futuras	166
8.4	Publicações	167
8.4.1	<i>Trabalho submetido em periódico</i>	167
8.4.2	<i>Trabalhos completos publicados em anais de congressos</i>	168
8.4.2.1	<i>Outros trabalhos completos publicados em anais de congressos</i>	169
	REFERÊNCIAS	171
APÊNDICE A	TRECHOS DOS CÓDIGOS-FONTE DO <i>MTSHOP-FAILS</i>	181
APÊNDICE B	TRECHOS DOS CÓDIGOS-FONTE DO <i>MTSHOP-SAFE</i>	191
APÊNDICE C	TRECHOS DOS CÓDIGOS-FONTE DO <i>ICARDAPIO</i>	203
APÊNDICE D	CASOS DE TESTE E DEFEITOS IDENTIFICADOS NO <i>MTSHOPFAILS</i>	215

APÊNDICE E CASOS DE TESTE E DEFEITOS IDENTIFICADOS
NO *ICARDAPIO* 223

INTRODUÇÃO

1.1 Contexto

A computação em nuvem é um modelo computacional que possibilita o acesso ubíquo, conveniente e sob demanda a um conjunto compartilhado de recursos (como redes, servidores, armazenamento, aplicações e serviços) que podem ser rapidamente disponibilizados com o mínimo de esforço de gerenciamento ou interação com o provedor de serviços (MELL; GRANCE, 2011). A base desse modelo surgiu a partir da contribuição de várias tecnologias como programação paralela, sistemas distribuídos e virtualização (RU; KEUNG, 2013) e tem se tornado um tema de interesse em pesquisa por demonstrar um futuro promissor quanto à redução de custo, otimização e a oportunidade de criar novos modelos de negócios (TAI *et al.*, 2010).

Segundo Vaquero *et al.* (2008), uma nuvem pode ser compreendida como um repositório de recursos computacionais virtualizados facilmente acessíveis. Os recursos computacionais podem ser reconfigurados dinamicamente para se ajustar a diferentes cargas de trabalho, otimizando sua utilização mesmo em circunstâncias desfavoráveis. Esse repositório de recursos é tipicamente explorado utilizando um modelo do tipo “pagamento-por-uso”, em que os fornecedores de serviços oferecem garantias customizadas no formato de *Service Level Agreements* (SLA)¹. A ideia da computação em nuvem na prática consiste em permitir que usuários adquiram recursos computacionais sob demanda e o pagamento desse serviço seja feito de acordo com o volume de utilização, similar ao modelo de cobrança utilizado por companhias de água encanada e energia elétrica (ARMBRUST *et al.*, 2010).

Com essa variedade de tipos de recursos sendo disponibilizados, surgiram diferentes modelos de entrega de serviços na nuvem. Dentre eles, podem-se destacar: o software como um serviço, do inglês *Software as a Service* (SaaS); a plataforma como um serviço, *Platform as a Service* (PaaS) e infraestrutura como um serviço, *Infrastructure as a Service* (IaaS) (MELL;

¹ <http://searchitchannel.techtarget.com/definition/service-level-agreement>

GRANCE, 2011). SaaS refere-se às aplicações residentes na infraestrutura da nuvem providas como um serviço aos consumidores (ARMBRUST *et al.*, 2010). PaaS caracteriza-se pelo fornecimento de uma plataforma para desenvolvimento e execução de aplicações. Com o apoio de uma plataforma na nuvem, desenvolvedores podem implementar e controlar aplicações, bem como suas configurações. IaaS consiste em uma camada que oferece uma infraestrutura computacional transparente, de modo que especialistas possam adquirir e gerenciar serviços de processamento, comunicação entre equipamentos, armazenamento, dentre outros recursos computacionais (RIMAL; CHOI; LUMB, 2009).

Os ambientes de computação em nuvem são diferentes de um ambiente tradicional em termos de implantação, configuração, execução e gerenciamento das aplicações (CHANA; CHAWLA, 2013). No modelo SaaS, as aplicações podem ser oferecidas como serviço para muitas organizações e com o mínimo de intervenção humana, no sentido de mantê-las disponíveis em níveis variados de demanda. Do ponto de vista dos provedores desses serviços, é importante que os recursos computacionais ao serem oferecidos sejam amplamente compartilhados. Além disso, o gerenciamento de múltiplas versões de aplicações residentes em locais dedicados mas que possuem funcionalidades comuns muitas vezes é ineficiente. Para as organizações, além da redução de custos espera-se alinhar a solução ao seu modelo de negócio. Nesse cenário, admite-se um padrão arquitetural chamado de *multi-tenancy* ou multi-inquilino. Esse modelo é um dos mecanismos fundamentais para um ambiente em nuvem e pode ser considerado em vários níveis, tais como dados, hardware e aplicação. Para sistemas SaaS, *multi-tenancy* é um dos principais fatores que os diferem de aplicações web convencionais. Esse padrão permite uma única instância da aplicação ser compartilhada de forma transparente entre múltiplas organizações, ou *tenants*, e seus usuários finais. Além disso, oferece a possibilidade de configuração exclusiva conforme as necessidades. Recursos do sistema, como serviços, aplicações, banco de dados, ou hardware também são compartilhados com o objetivo de reduzir custos (KABBEDIJK *et al.*, 2015).

Multi-tenancy é fundamental para que as nuvens compartilhem recursos de TI de maneira econômica e segura (BOBROWSKI, 2016). A dinamicidade do ambiente de nuvem em prover recursos computacionais sob demanda, alta escalabilidade devido a virtualização e a alta capacidade de sistemas multi-tenants em satisfazer necessidades específicas de várias organizações têm levado a migração para esse modelo. Entretanto, existem vários desafios ligados à qualidade, desconfiança das empresas em relação à segurança e dúvidas sobre as melhorias em termos de manutenção. De outro lado, muitas empresas fornecedoras de software vislumbraram os benefícios da multi-tenancy e conseguiram ampliar sua importância no mercado, como é o caso da Salesforce². O Salesforce.com é provavelmente o caso de sucesso mais conhecido nesse domínio (PINTO *et al.*, 2016), trata-se do CRM (*Customer relationship management*) mais utilizado e conta com uma plataforma de desenvolvimento de aplicativos em nuvem chamada Force.com que permite a customização de aplicações para atender requisitos de negócio específicos. Esse

² <http://www.salesforce.com/sales-cloud/overview/>

sistema conta com um modelo de receita de assinatura e cobra mensalmente clientes por usuário.

O desenvolvimento de software inclui uma série de atividades e mesmo utilizando técnicas, métodos e ferramentas adequadas, defeitos ainda podem persistir no produto final. Para contornar essa dificuldade, a atividade de teste é uma das atividades mais empregadas para melhorar a qualidade do software em complemento a outras atividades, como o uso de revisões e técnicas formais de especificação e de verificação (BARBOSA *et al.*, 2000). A atividade de teste consiste em um processo de execução de um programa com o objetivo de encontrar erros (JOVANOVIĆ, 2009; MYERS; SANDLER, 2004). Para realização de testes de forma efetiva, critérios de testes são empregados com a finalidade de sistematizar a atividade de teste e auxiliar na seleção de dados de teste. Assim, o desafio além de escolher um critério de teste adequado, consiste em selecionar casos de testes com maior probabilidade de encontrar defeitos que porventura existam. Apesar dos desafios impostos pela atividade de teste, ela é considerada uma atividade essencial no ciclo de vida de qualquer projeto de software.

A validação de sistemas multi-tenants inclui desafios em razão do compartilhamento de recursos, alta capacidade de customização e a dinamicidade do ambiente de nuvem (PINTO *et al.*, 2016). Determinadas tarefas podem incorrer em altos riscos e incertezas pois certos mecanismos fundamentais para esses sistemas podem se diferenciar significativamente e a comunidade ainda não possui uma arquitetura de referência consolidada. Muitas pesquisas nesse campo são voltadas para requisitos não funcionais ou possuem enfoque nos impactos que a inclusão de uma nova feature de um tenant podem gerar aos demais. Uma perspectiva a ser explorada no teste de sistemas multi-tenants é considerar questões que vêm sendo investigadas no contexto de teste de programas concorrentes, especificamente aquelas que exploram a comunicação de tarefas e a manipulação de estruturas de dados compartilhadas de forma que se possa garantir uma execução confiável em vários segmentos ao mesmo tempo (GOTTLIEB; ALMASI, 1989; SOUZA *et al.*, 2008; VERGÍLIO; SOUZA; SOUZA, 2005; SOUZA *et al.*, 2013). Não foram encontradas na literatura abordagens de teste de software para tais sistemas, explorando por meio de critérios de cobertura, características inerentes aos programas concorrentes. Critérios de teste de cobertura podem contribuir com este cenário adicionando mecanismos para medir a qualidade durante o desenvolvimento e validação dos sistemas.

A proposta definida durante este doutorado consiste em uma abordagem de teste para sistemas SaaS multi-tenants, especificamente para explorar possíveis desvios nos fluxos de execução devido ao comportamento não determinístico, um desafio conhecido na área de programação concorrente. Defeitos no isolamento de recursos dos tenants e problemas com o gerenciamento de sessões podem abrir brechas em termos de privacidade dos dados, além de execuções defeituosas que são difíceis de identificar e caracterizar suas circunstâncias. Teste de software para esse tipo de aplicações é uma emergente e promissora área de pesquisa, mas ainda carece de critérios de teste para revelar defeitos críticos que podem levar a diversos prejuízos.

1.2 Motivação

Desenvolver aplicações com alta qualidade e conduzir testes em sistemas multi-tenants com eficiência têm se tornado um importante fator para o sucesso das aplicações desenvolvidas nesse domínio. Além disso, existe a criticidade dessas aplicações, que exige testes mais robustos e com maiores garantias de que defeitos críticos não sejam liberados no release final para as organizações. Durante o desenvolvimento desses sistemas, geralmente são empregados frameworks que nem sempre oferecem todas as funcionalidades essenciais e garantem que determinados mecanismos atuem segundo as expectativas. Isso leva ao desenvolvimento de estratégias próprias, exigindo que os testes sejam executados considerando vários cenários de uso como a execução de múltiplas versões designadas aos tenants.

Portanto, com base no problema caracterizado, a tese aqui apresentada pode ser enunciada da seguinte forma: apoiar a condução da atividade de teste em sistemas SaaS multi-tenants com uma abordagem de teste que explore estruturas de dados compartilhadas de forma que se possa garantir uma execução confiável em vários segmentos ao mesmo tempo, que seja capaz de auxiliar na identificação de defeitos e contribua para a qualidade de aplicações neste domínio.

1.3 Objetivos

1.3.1 *Objetivo geral*

O objetivo central desta tese é estudar e definir uma abordagem de teste estrutural para o contexto de sistemas SaaS multi-tenants, que leve em consideração aspectos como concorrência.

1.3.2 *Objetivos específicos*

- Investigar os elementos essenciais que tornam um sistema multi-tenant e com isso, definir uma classificação de defeitos preliminar;
- Definir uma abordagem de teste para esse tipo de sistemas que explore em nível estrutural a capacidade de uma aplicação em lidar com múltiplas requisições de diversos tenants;
- Estudar os mecanismos mais importantes de sistemas multi-tenants, nos quais os testes podem ser priorizados;
- Investigar a capacidade dos testes automatizados e paralelos em revelar defeitos que são difíceis de reproduzir ao considerar versões isoladas;
- Investigar mecanismos de apoio à aplicação da abordagem de teste, incluindo a geração de um modelo e critérios de teste;
- Conduzir estudos experimentais para avaliação da abordagem proposta.

1.4 Metodologia de pesquisa

Inicialmente foi realizado um mapeamento sistemático da literatura em que o objetivo foi identificar tópicos recentes de pesquisa relacionados com a arquitetura multi-tenant e os principais desafios de pesquisa e oportunidades relacionadas ao desenvolvimento, teste e evolução de sistemas SaaS multi-tenants (PINTO *et al.*, 2016). Com esse estudo inicial também foi possível elaborar uma relação de características essenciais desse tipo de aplicações. De posse disso, partiu-se para a análise de um conjunto de aplicações, um survey com a indústria e o desenvolvimento de um sistema de e-commerce SaaS multi-tenant chamado *MtShop*³ para obtenção de um embasamento técnico necessário para as pesquisas. Essas atividades foram muito importantes para compreender a essência desse tipo de desenvolvimento e a complexidade do padrão multi-tenancy. Embora não se tenha conhecimento de uma arquitetura de referência para esse tipo de aplicações, as definições dos elementos fundamentais que fazem parte de uma aplicação multi-tenant serviram para estabelecer possíveis classes de defeitos, o que levou a proposição de uma classificação de defeitos (PINTO; SOUZA; SOUZA, 2019a).

Partindo dessa classificação, definiu-se o escopo da pesquisa e uma estratégia de testes que une testes automatizados com a execução paralela com a finalidade de simular um ambiente real (PINTO *et al.*, 2018; PINTO *et al.*, 2019b). Dois experimentos preliminares envolvendo a técnica funcional de teste foram realizados. Com base nos resultados, notou-se a viabilidade desse tipo de abordagem. Entretanto, analisar somente a saída das versões para cada versão específica dos tenants não foi suficiente pois não tornou-se possível rastrear em termos estruturais, onde os defeitos eram acionados e em quais circunstâncias. Essa observação levou à definição de um modelo de teste e uma família de critérios de cobertura. Refinamentos nesta abordagem de testes foram aplicados para considerar o teste funcional e estrutural com a finalidade de explorar principalmente, as requisições e redirecionamentos que levam à novas definições de uma variável de identificação, vital para qualquer sistema multi-tenant. Como avaliação, três estudos de caso exploratórios foram conduzidos com o objetivo de avaliar a aplicabilidade da abordagem, satisfação dos critérios, cobertura arquitetural e apoio a identificação de defeitos.

1.5 Contribuições

As principais contribuições desta tese para o teste de sistemas SaaS multi-tenants são:

- Estudo sistemático sobre as tendências de pesquisa e os desafios no campo de testes para o domínio em questão;
- Desenvolvimento de um sistema SaaS multi-tenant seguindo práticas recentes de desenvolvimento;

³ <http://mtshop.herokuapp.com>

- Uma classificação preliminar de defeitos par sistemas multi-tenants;
- Evidências empíricas sobre a capacidade dos testes de aceitação ao serem automatizados e executados em paralelo para as versões específicas dos tenants;
- Desenvolvimento de uma abordagem de teste que considera a dinamicidade do compartilhamento de uma única versão entre os tenants;
- Definição de um modelo e critérios de teste de cobertura para apoiar o teste de sistemas multi-tenants;
- Evidências sobre a capacidade do modelo de teste e dos critérios propostos em termos de satisfação, cobertura arquitetural e capacidade de identificar defeitos relativos aos desvios nos fluxos de execução das versões.

1.6 Organização da tese

Esta tese esta organizada em oito capítulos. Este capítulo apresentou o contexto no qual a proposta da pesquisa está inserida, bem como a motivação, os objetivos, a metodologia de pesquisa seguida e as principais contribuições originadas com esta pesquisa.

Nos Capítulos 2 e 3 apresentam-se a Fundamentação Teórica dos temas que estão relacionados à pesquisa: Teste de Software e Computação em Nuvem/Sistemas SaaS multi-tenants. No Capítulo 3 discute-se as principais funcionalidades e trechos de código representativos do *MtShop*.

No Capítulo 4 reportam-se classificações para o teste de sistemas SaaS e um Mapeamento Sistemático da Literatura. Discute-se também sobre a relação entre programas concorrentes e esses sistemas em termos de defeitos. Como uma das principais contribuições originadas desta tese, uma classificação preliminar de defeitos é proposta.

No Capítulo 5 descrevem-se dois estudos experimentais envolvendo a técnica funcional de teste e a execução de testes automatizados e paralelos com dois sistemas SaaS multi-tenants: *iCardapio* que permite o cadastro de cardápios eletrônicos e o *MtShop*.

No Capítulo 6 abordam-se um modelo de teste, uma família de critérios de cobertura e uma abordagem de teste que apoiada em testes automatizados e paralelos é capaz de auxiliar na identificação de defeitos. A avaliação da família de critérios de cobertura proposta e da estratégia de teste são apresentados no Capítulo 7. Para isso, adotou-se duas versões instrumentadas do *MtShop* (*MtShopFails* e *MtShopSafe*) e uma do *iCardapio*.

Finalmente, no Capítulo 8 discutem-se as conclusões, contribuições e limitações da pesquisa, assim como as lições aprendidas e oportunidades futuras. Uma lista com as publicações obtidas durante esta pesquisa de doutorado encerra o capítulo.

No Apêndice [A](#) apresentam-se trechos de Códigos-fonte do *MtShopFails*, no Apêndice [B](#) do *MtShopSafe* e no Apêndice [C](#) do *iCardapio*.

No Apêndice [D](#) descrevem-se casos de teste elaborados com a técnica funcional de teste e os defeitos identificados com os testes automatizados e paralelos *MtShopFails* e no Apêndice [E](#) para o *iCardapio*.

FUNDAMENTOS DE TESTE DE SOFTWARE

2.1 Considerações iniciais

A atividade de teste é essencial para a Garantia de Qualidade de Software (*Software Quality Assurance*), entretanto trata-se de uma atividade crítica e dinâmica, uma vez que defeitos podem ser inseridos ao longo de todo o processo de desenvolvimento e quanto mais tarde são identificados, maiores são os custos em resolvê-los e contornar seus efeitos. Além disso, na maioria dos casos o domínio de entrada a ser considerado nos testes é muito amplo e até infinito, os prazos de entrega do software impedem que estratégias de teste mais exaustivas sejam aplicadas, existe o apoio limitado de ferramentas, os atributos de qualidade a serem atendidos são diversos e por isso, alguns são negligenciados. Existem muitas razões e limitações que dificultam testar o software continuamente e garantir que esse processo tenha sido suficiente. Apesar desses fatores, o software sob teste precisa ser executado com dados de teste efetivos de modo que defeitos sejam identificados (MYERS; SANDLER, 2004), mesmo que se tenha consciência que essa atividade é limitada e não garante a ausência de defeitos.

Geralmente a atividade de teste envolve a execução da aplicação com casos de teste fornecidos por um profissional de teste que observa os resultados da execução e os compara com os valores esperados, posteriormente reporta os resultados dessa comparação (HOFFMAN, 2001; MYERS; SANDLER, 2004). Desse modo, critérios de teste devem ser adotados para auxiliar na definição de quais casos de teste utilizar e quando parar a execução dos testes (OSTRAND; BALCER, 1988). A técnica funcional de teste envolve a validação do software em condições reais sob a perspectiva dos usuários, ou seja, informações relativas à estrutura interna da aplicação não são consideradas. O objetivo não se restringe em verificar os requisitos como também assegurar que o software atende as necessidades dos usuários. A técnica baseada em defeitos considera o conhecimento sobre os defeitos comuns encontrados no processo de desenvolvimento para derivar dados de teste, embora tenha um custo elevado, é capaz de apoiar a seleção de casos de teste mais eficazes.

Na técnica estrutural, os critérios e requisitos são derivados a partir das características da implementação (MYERS; SANDLER, 2004), ou seja, trata-se de um teste em que os caminhos lógicos devem ser cobertos, fornecendo-se casos de teste que põem à prova tanto conjuntos específicos de condições e/ou laços bem como pares de definições e usos de variáveis. O objetivo é atingir um nível satisfatório de cobertura estrutural do software de forma complementar às demais técnicas de teste. Neste capítulo apresentam-se os fundamentos de teste de software, incluindo os princípios da técnica funcional, técnica baseada em defeitos e a técnica estrutural. Discute-se também a importância do teste automatizado para contornar os custos associados ao teste manual e no caso de aplicações web, verificar múltiplas combinações entre sistemas operacionais e browsers. Por fim, apontam-se exemplos de ferramentas que apoiam tais técnicas de teste e algumas em específico que foram empregadas para construção de testes automatizados no contexto de algumas investigações realizadas durante o doutorado.

2.2 Teste de Software

Apesar do desenvolvimento de software utilizar técnicas, métodos e ferramentas adequadas, erros podem persistir no produto final. Para que tais erros sejam revelados antes do software ser liberado, existe uma série de atividades, coletivamente chamadas de “*Verificação, Validação e Teste*”, ou *VV&T* (DELAMARO; MALDONADO; JINO, 2007). Dentre as técnicas de verificação e validação, a atividade de teste é uma das mais empregadas para evidenciar a qualidade do software (BARBOSA *et al.*, 2000).

Com relação à padronização de termos na área de teste de software (IEEE, 1990), tem-se as seguintes definições: **defeito** (*fault*) - passo, processo ou definição de dados incorretos; **engano** (*mistake*) - ação humana que produz um resultado incorreto, um defeito; **erro** (*error*) - diferença entre o valor obtido e o valor esperado. O erro, se propagado até a saída do software, constitui uma falha. Em particular, utiliza-se “erro” de maneira flexível, muitas vezes significando defeito, erro ou até falha (DELAMARO; MALDONADO; JINO, 2007). Existem outros termos utilizados na literatura no campo de teste de software, como: dados de teste, casos de teste e conjunto de testes.

Delamaro, Maldonado e Jino (2007) descrevem um cenário típico de teste que auxilia a compreensão desses termos. Na Figura 1 o domínio de entrada de um programa P , denotado por $D(P)$ é o conjunto de valores possíveis que podem ser usados para executar P . Assumindo que o programa P recebe como parâmetros de entrada dois valores inteiros x e y , de modo que $y \geq 0$, e o programa retorna o valor de x^y e caso os argumentos não estejam conforme a especificação $S(P)$, uma falha deverá ser observada. Com relação ao $D(P)$, é formado por todos os possíveis pares de números inteiros (x, y) e a saída de P é o conjunto de todos os possíveis resultados produzidos. Um “dado de teste” para P é um elemento de entrada que pertencente ao $D(P)$. Um “caso de teste” é constituído por um dado de teste e o resultado esperado na execução com

esse dado de teste. Por exemplo, para computação de x^y , o conjunto de casos de teste pode ser: $\langle (2,3), 8 \rangle$, $\langle (4,3), 64 \rangle$ e $\langle (3, -1), \text{"erro"} \rangle$. Com o conjunto de teste T , executa-se o programa P e verifica-se o resultado obtido. Caso as saídas obtidas estejam em conformidade com esse resultado, então nenhum erro foi identificado. Se para algum teste, o resultado for diferente, então ocorreu uma falha. Como ilustrado na Figura 1, em geral fica por conta do testador, baseado na especificação do programa $S(P)$ ou outra documentação, a análise sobre a correção de uma execução.

Figura 1 – Cenário típico da atividade de teste.

Fonte: Adaptada de Delamaro, Maldonado e Jino (2007).

A atividade de teste consiste em um processo de execução do programa com dados de teste para verificar se seu comportamento está em conformidade com o esperado (DELAMARO; MALDONADO; JINO, 2007). No caso dessa execução resultar comportamento ou saídas diferentes do esperado, um defeito pode ser identificado. Em geral, o teste de software envolve basicamente o planejamento, projeto de casos de teste, execução e avaliação dos resultados alcançados (MYERS; SANDLER, 2004; BEIZER, 1990). O teste pode ser desenvolvido ao longo do processo de desenvolvimento, concretizando em quatro fases: de unidade, de integração e de sistema (PRESSMAN, 2010).

O **teste de unidade** visa identificar erros de lógica e de implementação em cada módulo de forma individual. Posteriormente, com o **teste de integração** procura-se identificar erros durante a integração da estrutura do programa como interfaces entre os módulos. Depois que o software foi construído, aplica-se o **teste de sistema** que explora um contexto mais amplo do software com a intenção de verificar se todos os elementos se combinam corretamente (PRESSMAN, 2010). Além dessas fases, destaca-se o **teste de regressão** que é mais voltado para manutenção do software. A cada modificação efetuada no sistema, após sua liberação, corre-se o risco de que novos defeitos sejam introduzidos. Por esse motivo, é importante realizar testes que mostrem que as modificações efetuadas estão corretas, funcionam como esperado e que os requisitos anteriormente testados continuam válidos (DELAMARO; MALDONADO; JINO, 2007). Há também o **teste de aceitação**, uma fase do processo de teste em que um teste de caixa-preta é realizado no software antes de sua disponibilização. Tem como finalidade verificar o sistema em relação aos seus requisitos originais e às necessidades atuais do usuário.

Em geral, a atividade de teste é capaz de detectar todos os defeitos existentes, desde que o programa seja submetido a todas as entradas possíveis (DELAMARO; MALDONADO;

JINO, 2007). Entretanto, isso poderia tornar a atividade de teste impraticável, uma vez que o domínio de entrada geralmente é amplo ou infinito. Com isso, fica difícil para um programador afirmar que o programa esteja completamente correto e livre de erros. Partindo dessa limitação, técnicas e critérios foram elaborados para tornar a condução de teste sistemática, permitindo uma avaliação quantitativa dessa atividade. Assim, pode-se utilizar critérios de teste adequados e elaborar casos de teste que tenham maior probabilidade de encontrar defeitos com mínimo de recursos. Em geral, os critérios de teste são estabelecidos a partir das técnicas: funcional, que baseia-se na especificação do produto de software; a baseada em defeitos, que utiliza os defeitos comuns cometidos durante o processo de desenvolvimento do software e a estrutural, que utiliza a estrutura interna do programa.

2.2.1 Técnica funcional

O teste funcional é uma técnica utilizada para projetar casos de teste em que o programa é considerado como uma “caixa preta”, as entradas são fornecidas e as saídas são analisadas (MYERS; SANDLER, 2004). Desse modo, os detalhes de implementação não são considerados e o software é avaliado segundo o ponto de vista do usuário, ou seja, há pouca preocupação com a estrutura lógica interna do software (PRESSMAN, 2010).

Para Myers e Sandler (2004) os principais critérios definidos para o teste funcional são os seguintes:

- *Particionamento em classes de equivalência* ou *Equivalence Partitioning Criteria* (EPC): esse critério consiste em dividir o domínio de entrada do programa em classes de acordo com a especificação do programa, de modo que o testador possa selecionar casos de teste a partir de partições, as quais representam todo o domínio. Admite-se que se um elemento de uma classe revelar um defeito, outros elementos pertencentes a mesma classe também o detectam (ROPER, 1994; COPELAND, 2004).
- *Análise do valor limite* ou *Boundary Value Analysis* (BVA): complementa o critério de particionamento em classes de equivalência, auxiliando a definição de casos de teste que exploram condições limites, ou seja, valores que estejam exatamente sobre, acima ou abaixo dos limitantes das classes de equivalência. Nesse processo deve-se considerar tanto o ponto de vista das entradas quanto das saídas (MYERS; SANDLER, 2004).
- *Grafo causa-efeito*: esse critério auxilia na definição de casos de teste que exploram ambiguidades e incompletude nas especificações, explorando a combinação dos dados de entrada. Para isso emprega-se uma representação do programa chamado grafo causa-efeito, em que as possíveis combinações de entrada são as causas e as possíveis ações são os efeitos.

No caso de aplicações que possuem processamento complexo esses critérios podem não ser facilmente aplicáveis, uma vez que em combinações entre dados de teste, existe a necessidade de diretrizes para determinar dados de teste que cubram as classes de equivalência de forma mais eficiente (DELAMARO; MALDONADO; JINO, 2007).

Linkman, Vincenzi e Maldonado (2013) propuseram outro critério para técnica funcional, chamado de funcional sistemático, partindo da combinação dos critérios particionamento de equivalência e análise do valor limite. A motivação para isso, deve-se ao fato da baixa cobertura dos critérios funcionais, a falta de diretrizes para facilitar a geração e seleção de casos de teste que contribuam com a maior cobertura do código.

2.2.2 Técnica baseada em defeitos

A técnica baseada em erros utiliza o conhecimento sobre os defeitos comuns encontrados no processo de desenvolvimento para derivar dados de teste (DELAMARO; MALDONADO; JINO, 2007). O principal critério dessa técnica é a análise de mutantes que é empregada no teste de mutação.

No teste de mutação, defeitos típicos do processo de desenvolvimento são injetados com a finalidade de verificar se os casos de teste são capazes de descobri-los. Para isso, o programa sob teste é submetido à transformações sintáticas por meio de operadores de mutação. Como resultado desse processo, um conjunto de mutantes é gerado. Em seguida, parte-se para a análise de mutantes que requer a utilização de um conjunto de casos de teste e a verificação dos mutantes. A partir disso, procura-se determinar um conjunto de casos de teste que consiga revelar, através da execução da aplicação, as diferenças de comportamento existentes entre a aplicação original e seus mutantes. Esse procedimento serve para averiguar o quanto um conjunto de casos de teste é adequado e suficiente para o teste (BARBOSA *et al.*, 2000).

Existem alguns problemas com relação à análise de mutantes, um deles é a necessidade do testador decidir se um mutante está vivo ou morto. Um mutante pode ser considerado morto, caso sua execução com algum caso de teste produzir uma saída diferente do programa original. Outra questão importante é o caso dos mutantes equivalentes cujo comportamento é similar ao original. Isso indica uma fraqueza do conjunto de teste, pois não conseguiu distinguir a diferença (DELAMARO; MALDONADO; JINO, 2007). Com a execução dos mutantes pode-se ter uma ideia da adequação dos casos de teste utilizados, por meio do “escore de mutação” (*mutation score*). Dado um programa P e conjunto de teste T , calcula-se o escore de mutação $ms(P, T)$ com base na Equação 1. $DM(P, T)$ refere-se ao número de mutantes mortos pelo conjunto de casos de teste T ; $M(P)$ representa o número total de mutantes gerados a partir do programa P e $EM(P)$ é o número de mutantes gerados que são equivalentes a P .

$$ms(P, T) = \frac{DM(P, T)}{M(P) - EM(P)} \quad (1)$$

Assim, o escore de mutação pode ser obtido calculando a razão entre o número total de mutantes efetivamente mortos por T e o número de mutantes que se pode matar, dado pelo número total de mutantes gerados subtraído do número de mutantes equivalentes. Embora a análise de mutantes seja eficiente em revelar defeitos, sua prática apresenta problemas de custo relacionado ao grande número de mutantes gerados e a determinação de mutantes equivalentes.

A modelagem permite que o conhecimento sobre o sistema seja capturado e reutilizado durante as fases de desenvolvimento. Assim, um modelo pode contribuir com a tarefa de especificar como um sistema deve se comportar. Nesse sentido, outra técnica que vem ganhando espaço é o teste baseado em modelos. Nessa técnica diferentes métodos podem ser usados para derivar sistematicamente casos de teste a partir de modelos, gerando sequências de teste. Além disso, essa técnica permite avaliar propriedades importantes do sistema (SIMÃO; PETRENKO, 2008; DELAMARO; MALDONADO; JINO, 2007).

2.2.3 Técnica estrutural

Na técnica estrutural, os critérios e requisitos são derivados essencialmente a partir das características de uma implementação particular (MYERS; SANDLER, 2004; PRESSMAN, 2010), ou seja, trata-se de um teste “caixa branca”. Os caminhos lógicos são testados, fornecendo-se casos de teste que põem à prova tanto conjuntos específicos de condições e/ou laços bem como pares de definições e usos de variáveis (DELAMARO; MALDONADO; JINO, 2007). Os critérios estruturais, em geral, são baseados na complexidade, em fluxo de controle e em fluxo de dados. Um critério bastante conhecido dentre os baseados na complexidade é o critério de McCabe (MCCABE, 1976), que utiliza a complexidade ciclomática para derivar os requisitos de teste.

Critérios baseados no fluxo de controle utilizam somente características de controle da execução do programa, como comandos ou desvios para determinar quais estruturas são necessárias (MYERS; SANDLER, 2004). Para isso, uma representação gráfica do programa conhecida como Grafo de Fluxo de Controle (GFC) é utilizada. No GFC, um programa é representado por um conjunto de nós que correspondem a blocos indivisíveis e por arestas que representam o controle do fluxo de informações entre os nós. A partir desse grafo, caminhos de execução no programa são estabelecidos os quais iniciam em um nó de entrada e terminam em um nó de saída do grafo. Essa representação serve de auxílio para a condução do teste, usado como base na aplicação e avaliação dos critérios e para derivar casos de teste. Os testes são realizados com base na implementação, por meio do controle da execução do programa. Dentre os critérios baseados no fluxo de controle do programa, os mais utilizados são:

- *Todos-Nós*: exige que a execução do programa passe, pelo menos uma vez, em cada vértice do GFC, ou seja, que cada comando do programa seja executado pelo menos uma vez;

- *Todas-Arestas*: requer que cada aresta do grafo, isto é, cada desvio do fluxo de controle do programa, seja exercitada pelo menos uma vez;
- *Todos-Caminhos*: requer que todos os caminhos possíveis do programa sejam executados.

Os critérios baseados em fluxo de dados utilizam a análise do fluxo de dados (HECHT, 1977) para derivar os requisitos de teste. Nesses critérios, as associações entre a definição de uma variável e seus possíveis usos subsequentes são admitidos. As principais famílias de critérios dessa categoria foram propostos por Rapps e Weyuker (RAPPS; WEYUKER, 1985; RAPPS; WEYUKER, 1982) e os critérios Potenciais-Usos por Maldonado (1991).

Para derivar os requisitos de teste exigidos pelos critérios, Rapps e Weyuker também apresentaram algumas definições, como o “grafo Def-Uso” que é uma extensão do GFC. Nesse grafo são adicionadas informações a cerca do fluxo de dados do programa, caracterizando associações entre pontos do programa nos quais é atribuído um valor a uma variável (definição) e pontos nos quais esse valor é utilizado (referência ou uso da variável). O uso de variáveis pode ser classificado em dois tipos distintos: *c-uso*, quando a utilização da variável afeta diretamente uma computação sendo realizada ou permite que o resultado de uma definição anterior seja observado; e *p-uso*, quando a variável é usada em uma expressão em comandos de desvios, afetando diretamente os fluxos de controle do programa (MALDONADO, 1991). Os principais critérios de Rapps e Weyuker são:

- *Todas-Definições*: requer que cada definição de variável seja exercitada pelo menos uma vez, não importa se por um *c-uso* ou por um *p-uso*;
- *Todos-Usos*: requer que todas as associações entre uma definição de variável e seus subsequentes usos (*c-usos* e *p-usos*) sejam exercitados pelos casos de teste, por pelo menos um caminho livre de definição, isto é, um caminho em que a variável não seja redefinida;
- *Todos-Du-Caminhos*: requer que toda associação entre uma definição de variável e subsequentes *p-usos* ou *c-usos* dessa variável seja exercitada por todos os caminhos livres de definição e livres de laço que cubram essa associação.

Esses critérios exigem a ocorrência explícita do uso de variáveis definidas. Como uma alternativa, há os critérios Potenciais-Usos, nos quais o uso da variável não precisa estar aparente, basta que seja possível sua ocorrência. Os critérios Potenciais-Usos (MALDONADO, 1991) subdividem-se em:

- *Todos-Potenciais-Usos*: requer que pelo menos um caminho livre de definição de uma variável x definida em um nó i para todo nó e toda aresta alcançável a partir de i seja exercitada.

- *Todos-Potenciais-Usos/Du*: requer que pelo menos um Potencial-Du-Caminho com relação a variável x definida em i para todo nó e para aresta alcançável a partir de i seja exercitada.
- *Todos-Potenciais-Du-Caminhos*: requer que todos os Potenciais-Du-Caminhos com relação a todas as variáveis x definidas e todos os nós e arestas alcançáveis a partir dessa definição sejam exercitados.

Em geral, tais técnicas são complementares e os critérios contribuem para melhorar a qualidade do conjunto de testes e em consequência, a efetividade da atividade de teste. Portanto, devem ser usadas em conjunto, pois cobrem classes diferentes de defeitos. Por se tratar de uma atividade dinâmica, no sentido de exigir a execução da aplicação com os casos de teste, na maioria dos casos torna-se impraticável realizá-la manualmente, assim, geralmente ferramentas de apoio são empregadas. Nesse sentido, várias ferramentas foram desenvolvidas no grupo de pesquisa em Engenharia de Software do ICMC/USP, tais como: JFTtool¹ (*Java Functional Testing tool*) que gera dados de teste de maneira automatizada baseando-se nos critérios de particionamento em classes de equivalência, análise do valor limite e grafo causa-efeito; *JABUTi* (VINCENZI *et al.*, 2005) para teste estrutural em programas Java; *Proteum* (*PROgram Testing Using Mutants*) (DELAMARO; MALDONADO, 1996) que apoia o critério de análise de mutantes em programas C, entre outras.

Apesar da capacidade da atividade de teste, existem várias limitações como: correitude, executabilidade, equivalência de programas e correção coincidente. Como não existe um algoritmo de teste de propósito geral para provar a **correitude** de um programa, diz-se que um programa com domínio de entrada D está correto com relação a uma especificação se o comportamento do programa está de acordo com o comportamento esperado para todos os itens de dados d , pertencentes ao seu domínio de entrada D . Entretanto, utilizar todas as entradas possíveis de um determinado domínio, na maioria dos casos é considerado impraticável. Com relação a **executabilidade**, dado um programa com domínio de entrada D , diz-se que um caminho é executável se existe ao menos um item de dado $d \in D$ que leve a execução desse caminho. Todavia, podem existir caminhos não-executáveis. A **equivalência de programas** existe se o comportamento de dois programas é idêntico para todos os itens de dados pertencentes ao domínio de entrada. Assim, não se pode garantir que duas sequências de comando (caminhos) que computam a mesma função em programas distintos são equivalentes. A **correção coincidente** refere-se ao fato de um programa apresentar um resultado correto para um item de dado $d \in D$, satisfazendo a um requisito de teste e não revelando a presença do erro. No entanto, se outro dado de entrada é selecionado, o resultado pode ser incorreto e a presença do erro pode ser identificada.

¹ <http://ccsl.icmc.usp.br/pt-br/projects/jfttool>

2.2.4 *Automatização de testes de aceitação do usuário*

Projetos de software complexos exigem um investimento considerável em teste de software para garantir qualidade. Em geral, esse custo está associado ao esforço e tempo gasto para planejar, executar e reportar os resultados dos testes, impactando no custo global de desenvolvimento. Por exemplo, o teste de aplicações web tem um alto custo pois requer atenção a compatibilidade do browser do cliente, as aplicações mudam com frequência e existem muitas entradas possíveis. Além disso, o teste depende do desempenho do browser e os clientes e servidores se comunicam via Internet. Diante disso, a automatização de testes permite reduzir o tempo gasto e contribui para redução dos custos a medida que novas funcionalidades são desenvolvidas e validadas. Nesse processo, existem abordagens para geração de dados de teste e ferramentas que apoiam a implementação de testes para simular ações dos usuários, como é o caso do teste de aceitação do usuário, comentado anteriormente. As especificações podem ser comparadas com o sistema com o intuito de verificar se o software reflete as expectativas de seus usuários.

Em geral, os testes de aceitação são muito utilizados para testar se o aplicativo está funcionando corretamente e sua automatização permite seguir um método mais preciso, eficiente e confiável, uma vez que os testes manuais são demorados, caros e sujeitos a erros. Com os testes automatizados pode-se simular múltiplos cenários de utilização e verificar se os requisitos foram atendidos ([PUSULURI, 2006](#)) e no caso dos testes de regressão, a redução dos custos se torna ainda mais perceptível pois como o objetivo é verificar se os requisitos continuam válidos quando novas funcionalidades são incorporadas, basta aplicar as mudanças correspondentes nos scripts e executar o teste automatizado para cobrir tanto as features aparentemente não afetadas como também as novas alterações. Além de automatizar esse tipo de teste, pode-se torná-lo paralelo para executar em várias instâncias com configurações de cenários de teste diferentes. Essa é uma prática crescente em serviços que oferece o teste de aplicações na nuvem ([AHMAD; ANDRAS, 2018](#); [LNĚNÍČKA; ČAPEK *et al.*, 2018](#)), uma vez que garante que os testes sejam beneficiados ao dividir as tarefas para reduzir o tempo para execução.

Testes de aceitação do usuário automatizados usam um software para controlar a execução e realizar a comparação dos resultados alcançados com os esperados, além disso permite testar pré-condições, controlar e reportar funções anômalas. Geralmente, são implementados partindo de um processo manual que esteja em prática e que admite um processo de teste formalizado. Existem várias vantagens na utilização de testes automatizados, a maioria delas está vinculada com o nível de reusabilidade dos testes e com o desempenho da execução. Muitas aplicações são desenvolvidas para web em uma era de processos de software altamente iterativos e responsivos, exigindo cada vez mais abordagens de teste que sejam eficientes e com um custo relativamente baixo. Nesse cenário propício para metodologias ágeis, a automação de testes está frequentemente se tornando um requisito para projetos de software em resposta às demandas como: testes de regressão frequentes, feedback rápido aos desenvolvedores, documentação disciplinada

dos casos de teste e principalmente, uma alternativa para encontrar defeitos que não tenham sido identificados por testes manuais. Existem várias ferramentas comerciais e open-source disponíveis para auxiliar no desenvolvimento da automação de testes.

*Selenium*² é um conjunto de ferramentas para apoiar a automação dos testes e provavelmente a solução open-source mais utilizada. A maioria dos engenheiros de qualidade que utilizam o Selenium concentram-se na utilização de apenas parte das ferramentas que mais atendem a necessidade de seus projetos. Cada ferramenta fornece opções para atender diferentes problemas dos testes automatizados, resultando em um conjunto de funcionalidades orientadas às necessidades dos testes de aplicações web. Assim, tais operações são flexíveis, permitindo várias opções para localizar elementos da interface ao mesmo tempo que o Selenium consegue comparar os resultados dos testes previstos com o comportamento real da aplicação (SELENIUM, 2019). Isso é possível pois essa suíte permite executar os testes para diferentes sistemas operacionais e browsers, além disso o teste automatizado pode ser controlado por muitas linguagens de programação e frameworks de teste com o *JUnit*³, framework open-source com suporte à criação de testes automatizados na linguagem Java. A seguir descrevem-se as principais ferramentas que fazem parte do Selenium:

- *Selenium IDE (Integrated Development Environment)* é uma ferramenta de prototipagem para criar scripts de teste, disponibilizado como um plugin para Firefox e Chrome e fornece uma interface fácil de usar para o desenvolvimento de testes automatizados. O *Selenium IDE* possui um recurso de gravação, que registra as ações do usuário conforme são executadas e as exporta como um script reutilizável em uma das muitas linguagens de programação que podem ser executadas posteriormente;
- *Selenium RC (Remote Control)*: por muito tempo foi o principal projeto até o desenvolvimento do *Selenium WebDriver* e permitia a execução de scripts desenvolvidos em várias linguagens de programação;
- *Selenium WebDriver*: é uma biblioteca que permite controlar a execução do browser partindo de um programa Java ou em outra linguagem. Além de apoiar a execução de scripts, possui comunicação direta com os browsers para simular as ações dos usuários de forma automatizada. Para isso, a ferramenta utiliza suporte nativo de cada browser. A forma como essas chamadas diretas são feitas e os recursos que elas suportam dependem do browser, portanto “drivers” específicos são disponibilizados para cada browser. No *Selenium RC* funcionava seguindo essa ideia para cada browser suportado e “injetava” funções em javascript no browser quando ele era carregado e em seguida, utilizava javascript para conduzir os testes de aceitação do usuário. O *Selenium WebDriver* não utiliza essa técnica, mas sim direciona o navegador diretamente usando um suporte integrado para automação;

² <https://www.seleniumhq.org/>

³ <https://junit.org/junit5/>

- *Selenium Grid*: permite que a solução *Selenium RC* seja dimensionada para grandes conjuntos de testes e que devem ser executados em vários ambientes. O *Selenium Grid* permite executar os testes em paralelo e em diferentes máquinas remotas. No caso de um largo conjunto de testes torna-se possível aumentar substancialmente o desempenho ao dividir o conjunto de testes e executá-los ao mesmo tempo usando diferentes máquinas. Além disso, se for necessário executar o conjunto de testes em várias configurações de ambientes, diferentes máquinas remotas suportando e executando os testes ao mesmo tempo. Em geral, o Selenium Grid melhora muito o tempo para executar o conjunto de testes, fazendo uso do processamento paralelo.

A maioria dos provedores de TaaS (*Testing as a Service*) oferecem testes automatizados com suporte ao Selenium, como é o caso do *SauceLabs*⁴, *BrowserStack*⁵ e *TestingBot*⁶. O teste na nuvem geralmente é adotado para teste de carga, workload e testes de performance (GAO; BAI; TSAI, 2011). Testes envolvendo a técnica funcional, como teste de unidade, teste de integração, teste de sistemas e teste de aceitação também são explorados no domínio de computação em nuvem (AKERELE; RAMACHANDRAN; DIXON, 2013). Os principais benefícios desse tipo de teste estão relacionados ao uso de recursos de processamento, máquinas virtuais e a própria infraestrutura de nuvem (GAO *et al.*, 2013). O ambiente de nuvem muda seu estado em tempo de execução e na proporção em que instâncias de máquinas virtuais são alocadas. Como uma consequência, o teste pode se tornar uma tarefa complexa pois a efetividade do escalonamento e balanceamento de carga não pode ser totalmente preditivo e controlado mesmo quando se tem uma única aplicação sendo verificada (BAI *et al.*, 2011). Isso é diferente para aplicações web convencionais, no qual o teste paralelo é essencialmente utilizado para confirmar múltiplas combinações com sistemas operacionais e browsers (GARG; DATTA, 2013) e visa resolver problemas relativos a compatibilidade *cross-platform*.

2.3 Considerações finais

Neste capítulo foram apresentados os principais conceitos de teste de software envolvidos neste trabalho. Discutiu-se sobre as preocupações em selecionar um conjunto de casos de teste mais relevante para identificação de defeitos e que embora não seja possível garantir a ausência deles, recomenda-se empregar técnicas e critérios de teste adequados antes de disponibilizar uma solução de software aos usuários. A adoção de critérios de testes auxilia o estabelecimento de uma medida de confiabilidade e uma condição de parada ao processo de definição de casos de teste. Por fim, foram apresentadas as principais razões que levam a evolução de determinadas tarefas de um processo manual de testes para uma abordagem automatizada. Discutiu-se aspectos

⁴ <https://saucelabs.com/>

⁵ <https://www.browserstack.com/>

⁶ <https://testingbot.com/>

de automatização focando em testes de aceitação, pois serão usados na abordagem proposta nesta tese.

Com o objetivo de caracterizar as pesquisas no campo de teste de aplicações na nuvem, no próximo capítulo descrevem-se os conceitos de Computação em Nuvem, o modelo de entrega de serviço SaaS (Software as a Service) e um padrão arquitetural chamado multi-tenancy. Destacam-se os elementos essenciais desse padrão e determinadas decisões no projeto que impactam na forma de implementar uma solução multi-tenant. Partindo desses fundamentos, uma aplicação de e-commerce chamada *MtShop* foi desenvolvida para constituir uma base técnica necessária às pesquisas.

COMPUTAÇÃO EM NUVEM E SISTEMAS SAAS MULTI-TENANTS

3.1 Considerações iniciais

Neste capítulo descrevem-se os principais conceitos que fundamentam esta tese. Inicialmente são apresentados os conceitos de Computação em Nuvem, incluindo os principais modelos de entrega de serviços. Discutem-se as particularidades das aplicações disponibilizadas como um serviço no modelo de SaaS (*Software as a Service*) e sobre o padrão arquitetural *Multi-tenancy*. Esse padrão é considerado como um dos principais mecanismos que diferenciam aplicações web convencionais das aplicações projetadas para nuvem e permite atender múltiplas organizações ao mesmo tempo que uma única instância da aplicação é compartilhada. Embora diversos benefícios possam ser obtidos em termos de manutenção, redução de custos e gerenciamento, as estruturas compartilhadas e a alta capacidade de customização em um sistema SaaS multi-tenant abrem margens para vários desafios na atividade de testes. Para compreender tais desafios destacam-se neste capítulo os elementos esperados nesse tipo de aplicação, embora não se tenha uma arquitetura de referência consolidada. Mediante a esse conhecimento e com o intuito de constituir uma base técnica, um sistema de e-commerce chamado *MtShop* foi desenvolvido. Algumas telas dessa aplicação e trechos de código representativos tornam os conceitos mais claros e a forma como a aplicação lida com requisições simultâneas ajudam a ilustrar de forma prática a essência desse modelo de aplicações.

3.2 Computação em Nuvem

A computação em nuvem surgiu a partir de outras tecnologias, como computação paralela, computação distribuída e virtualização (RU; KEUNG, 2013). Esse modelo tem se tornado uma das principais tendências tecnológicas, uma vez que transforma a maneira como recursos

computacionais são fornecidos aos clientes, visando a redução de custo, qualidade em termos de disponibilidade e desempenho, otimização e a oportunidade de criar novos modelos de negócios (TAI *et al.*, 2010). Nesse modelo, um conjunto de recursos pode ser oferecido como serviço e acessado de qualquer lugar de forma eficiente e sob demanda, exigindo o mínimo de intervenção no gerenciamento por parte do fornecedor.

Com a variedade de tecnologias envolvidas e tipos de recursos oferecidos, várias definições podem ser encontradas para computação em nuvem. Vaquero *et al.* (2008) conduziram uma revisão na literatura com o intuito de propor uma definição mínima, mas contendo todas as características essenciais. Para isso, mais de 20 definições foram selecionadas. Segundo os autores, as nuvens são extensos grupos de recursos virtualizados, facilmente utilizáveis e acessíveis (hardware, plataformas de desenvolvimento e/ou serviços). Esses recursos podem ser reconfigurados dinamicamente para se ajustar a uma carga (escala) variável e serem utilizados de forma otimizada. Esse conjunto de recursos é tipicamente explorada por meio de um modelo “pagamento-por-uso”, no qual garantias são oferecidas pelo provedor de infraestrutura por meio de SLAs customizadas. Nesse modo de pagamento, os custos desses serviços para os consumidores podem ser proporcionais à utilização, tempo de uso ou até mesmo por meio de uma assinatura mensal, ou seja, não é necessário pagar por uma licença integral de uso de software ou outro recurso fornecido.

O modelo de computação em nuvem envolve os seguintes elementos: **cliente** (do inglês, “*cloud consumer* ou *customer*”) - uma pessoa ou organização que seja um cliente de uma nuvem, o qual também pode ser uma nuvem que oferece serviços a outros consumidores; **dispositivo** (“*client*”) - uma máquina ou aplicação de software que acessa uma nuvem por meio de uma conexão de rede; e o **provedor da nuvem** (“*cloud provider* ou *provider*”) - uma organização que fornece serviços na nuvem. Além desses elementos, o NIST (*National Institute of Standards and Technology*) fornece uma definição de cinco características essenciais da computação em nuvem (BADGER *et al.*, 2012):

- *Auto serviço sob demanda*: um consumidor pode utilizar recursos da computação em nuvem que estão disponíveis de forma unilateral e são oferecidas sob demanda, tais como: armazenamento, rede, o uso de software, e assim por diante, de maneira que esses recursos podem ser cedidos automaticamente o quanto forem necessários sem recorrer à cada provedor.
- *Amplo acesso à rede*: os recursos estão disponíveis via rede e acessíveis por meio de mecanismos padronizados que permitem a utilização de plataformas heterogêneas (como telefones celulares, tablets, laptops, entre outros).
- *Pool de recursos*: os provedores de recursos na nuvem servem múltiplos consumidores usando o modelo *multi-tenancy*, em que diferentes recursos físicos e virtuais são atribuídos e realocados dinamicamente de acordo com a demanda do consumidor. Os clientes

geralmente não possuem controle ou conhecimento sobre a localização exata dos recursos disponibilizados. Exemplos de recursos incluem armazenamento, processamento, memória e banda de rede.

- *Rápida elasticidade*: os recursos podem ser fornecidos rapidamente de forma elástica, em alguns casos automaticamente. Para o consumidor, os recursos disponíveis muitas vezes parecem ser ilimitados e podem ser utilizados em qualquer quantidade e momento.
- *Medição de serviço*: mecanismos da nuvem controlam automaticamente os recursos oferecidos e otimizam sua utilização. Medições são realizadas de acordo com o tipo de serviço, de modo que os recursos são monitorados e controlados, fornecendo transparência tanto para o provedor quanto para o consumidor.

Com relação aos modelos de implantação ou de desenvolvimento da nuvem, quatro podem ser destacados: nuvem privada, comunitária, pública e híbrida (BADGER *et al.*, 2012; MELL; GRANCE, 2011). No caso da nuvem privada, a infraestrutura de nuvem é utilizada exclusivamente por uma organização, compreendendo múltiplos consumidores (ou unidades de negócio). Questões envolvendo a segurança e privacidade dos dados tornam esse modelo uma escolha adequada para muitas instituições. Na nuvem comunitária, a infraestrutura da nuvem é oferecida exclusivamente para o uso de uma comunidade específica de consumidores que possuem interesses compartilhados (objetivos, requisitos de segurança, considerações e políticas de privacidade). Geralmente uma nuvem privada ou comunitária é administrada local ou remotamente pela própria empresa, por terceiros ou uma combinação entre eles. Na nuvem pública, a infraestrutura da nuvem é disponibilizada para uso aberto pelo público em geral. A nuvem pode ser proprietária, gerenciada e operada por uma empresa, por uma universidade, organização do governo, ou alguma combinação. Na nuvem híbrida, a infraestrutura da nuvem é uma composição de duas ou mais infraestruturas de nuvens distintas (privada, comunitária ou pública) que permanecem entidades únicas, mas unidas por uma tecnologia padronizada ou proprietária que permite a portabilidade dos dados e aplicações.

3.2.1 Modelos de entrega de serviços

Dependendo de quais recursos são compartilhados e entregues aos consumidores, existem três modelos de serviços mais conhecidos. Na terminologia de computação em nuvem, quando hardware como processadores, armazenamento e rede são oferecidos como serviço, isso é chamado de infraestrutura como serviço, do inglês “*Infrastructure as a Service*” (IaaS). *Amazon Elastic Compute Cloud*¹ (Amazon EC2) e *Amazon Simple Storage Service*² (Amazon S3) são exemplos de IaaS. Quando ferramentas e plataformas de programação como Java, Python, .Net, MySQL e APIs são oferecidas como serviço, tem-se uma plataforma como serviço, “*Platform as*

¹ <https://aws.amazon.com/pt/ec2/>

² <https://aws.amazon.com/pt/s3/>

a Service” (PaaS). Microsoft Azure³, Heroku⁴ e Google App Engine⁵ são considerados PaaS. Quando aplicações são entregues como serviço, isso pode ser chamado de software como serviço, “Software as a Service” (SaaS) (MAHMOOD; SAEED, 2013; BADGER *et al.*, 2012; MELL; GRANCE, 2011), tais como: Google Apps⁶, Facebook⁷, Oracle⁸, Salesforce⁹, dentre outros.

O modelo IaaS consiste em uma camada que oferece ao usuário uma infraestrutura computacional transparente, cuja entrega de infraestrutura computacional é tratada como serviço. O cliente adquire sob demanda serviços de processamento, armazenamento e demais recursos computacionais (RIMAL; CHOI; LUMB, 2009). A infraestrutura inclui instalação de rede, recursos virtualizados e máquinas virtuais que são fornecidos aos clientes. Nesse modelo, os consumidores não precisam se preocupar com a alocação de mais recursos na infraestrutura virtualizada, uma vez que esse processo pode ser dinâmico e automatizado (DILLON; WU; CHANG, 2010). No entanto, os consumidores podem criar, destruir e gerenciar máquinas virtuais e armazenamento. Além disso, é possível medir a utilização dos recursos ao longo de um período de tempo. O modelo PaaS caracteriza-se pelo fornecimento de uma plataforma de desenvolvimento de aplicações que é disponibilizada sobre uma infraestrutura de nuvem (RIMAL; CHOI; LUMB, 2009).

O modelo SaaS refere-se às aplicações residentes na infraestrutura da nuvem e que são fornecidas como um serviço aos consumidores por meio da Internet (ARMBRUST *et al.*, 2010). Uma aplicação oferecida como serviço requer residência em uma plataforma. Nesse sentido, o modelo PaaS apoia tanto o desenvolvimento, incluindo o ambiente de programação e ferramentas de gerenciamento quanto a hospedagem de aplicações no ambiente de nuvem. Este projeto se concentra no modelo SaaS.

As aplicações são executadas sobre uma plataforma e infraestrutura de nuvem (ARMBRUST *et al.*, 2010) e o consumidor não gerencia ou controla a infraestrutura da nuvem, incluindo rede, servidores, sistemas operacionais e armazenamento, pois essas questões são responsabilidades do provedor da infraestrutura da nuvem. O provedor do SaaS é responsável pelos recursos individuais de uma determinada aplicação, com a exceção para configurações limitadas que são fornecidas pelos próprios aplicativos. O provedor de um SaaS deve garantir que o software satisfaça requisitos de qualidade envolvendo a nuvem e a aplicação.

Chong e Carraro (2006) apontam duas categorias principais de software como serviço: serviços de linha de negócios e orientados ao cliente. No primeiro caso, a aplicação é oferecida para empresas e organizações de todos os tamanhos. Serviços de linha de negócios geralmente são soluções de negócios grandes e personalizáveis direcionadas para facilitar processos de

³ <https://azure.microsoft.com/pt-br/>

⁴ <https://www.heroku.com/>

⁵ <https://cloud.google.com/appengine/>

⁶ <http://www.google.com/intx/pt-BR/enterprise/apps/business/>

⁷ <https://www.facebook.com/>

⁸ <https://www.oracle.com/cloud/saas.html>

⁹ <http://www.salesforce.com/sales-cloud/overview/>

negócios como finanças, gerenciamento da cadeia de suprimentos e relações com o cliente. Normalmente esses serviços são vendidos aos clientes como assinatura. Serviços orientados ao cliente são oferecidos ao público em geral, às vezes são vendidos como assinatura, mas geralmente são fornecidos sem custo e financiados por anúncios.

Na Figura 2 ilustram-se os papéis envolvidos em relação aos modelos de serviços na computação em nuvem. Tais papéis são importantes para definir responsabilidades para os diferentes atores em uma solução de computação em nuvem (MARINOS; BRISCOE, 2009). Um provedor pode oferecer uma plataforma e uma infraestrutura como serviço para um desenvolvedor, o qual pode fornecer ao usuário final, aplicações como serviço. Uma aplicação SaaS é desenvolvida e executada em uma plataforma específica.

Figura 2 – Papéis e modelos de serviços na computação em nuvem.

Fonte: Adaptada de [Marinos e Briscoe \(2009\)](#).

Na seção seguinte apresentam-se mais detalhes sobre o padrão *multi-tenancy* por ser um dos mecanismos mais importantes na Computação em Nuvem e que mais difere aplicações projetadas para nuvem em relação às aplicações web convencionais. Nesta tese será dado enfoque a esse padrão.

3.3 SaaS (Software as a Service) e Multi-tenancy

Com os avanços na tecnologia e as mudanças nos modelos de negócios, novas abordagens são propostas mudando a forma como as aplicações são projetadas, construídas e entregues aos usuários. O amplo acesso à Internet, arquiteturas orientadas a serviços (PERREY; LYCETT, 2003) e as ineficiências de gerenciar aplicações em locais dedicados contribuem para a adoção de um modelo em que a entrega seja baseada em serviços, consistindo no software como serviço. Um dos mecanismos principais das soluções SaaS é a capacidade de compartilhar recursos

computacionais ao disponibilizar um produto de software para diferentes clientes. Na nuvem, um cliente ou inquilino (*tenant*) pode ser um usuário, um grupo de usuários ou uma organização. Para atingir essa capacidade, a arquitetura de um SaaS precisa fornecer o compartilhamento de instâncias e banco de dados, dentre outros recursos. Um estilo arquitetural conhecido com essa finalidade é a Arquitetura *Multi-tenant* (AMT). Para a comunidade de arquitetura de software, *multi-tenancy* é compreendida como a capacidade de atender múltiplas organizações simultaneamente, por meio de uma instância do software (BEZEMER; ZAIDMAN, 2010).

Uma aplicação *single-tenant* requer um conjunto dedicado de recursos para atender satisfatoriamente as necessidades de uma organização, enquanto que uma aplicação *multi-tenant* pode satisfazer a necessidade de múltiplas organizações empregando os recursos de hardware e software para gerenciar uma única instância da aplicação. Nesse sentido, os tenants utilizam um serviço que permite o isolamento virtual da aplicação, ou seja, as organizações podem utilizar e customizar uma aplicação como se a mesma fosse uma instância separada, pois seus dados e as alterações permanecem isoladas (SALESFORCE, 2008).

Os princípios de multi-tenancy tem base em vários modelos, como *mainframes*, *Application Service Provider* (ASP) e multi-usuário. No domínio de software, o termo *multi-tenancy* é relativamente recente na literatura, com a primeira citação explícita em um artigo de Chong e Carraro (2006). Segundo os autores, SaaS tem um grande impacto na indústria de software, pois o software como serviço transforma a maneira como as pessoas constroem, vendem, compram e utilizam software. Chong e Carraro propuseram um modelo de maturidade para SaaS que possui quatro níveis. A maturidade nesse modelo não se trata de uma proposição sequencial em que todas aplicações devem evoluir para atingir o quarto nível. Um aplicativo pode possuir apenas alguns atributos de um dos níveis e assim, atender todos os requisitos necessários de um negócio. Ressalta-se que os dois últimos níveis descrevem algumas características da multi-tenancy como requisitos para um SaaS maduro. Na Figura 3 ilustram-se os quatro níveis de maturidade: nível 1 - *ad-hoc*/personalizado, nível 2 - configurável, nível 3 - configurável e eficiente para vários tenants e nível 4 - escalonável, configurável e eficiente para vários tenants.

- No **nível 1**, o SaaS assemelha-se ao modelo ASP em que cada cliente tem a sua própria versão personalizada do aplicativo hospedado e executa a sua própria instância do aplicativo nos servidores. Essa instância é totalmente independente de quaisquer outras instâncias ou processos que estejam sendo executados para os seus outros clientes.
- No **nível 2**, tem-se uma instância separada do aplicativo para cada tenant, enquanto que no nível 1 cada instância é personalizada individualmente para o tenant, no nível 2 todas as instâncias utilizam a mesma implementação de código e o fornecedor atende as necessidades dos clientes fornecendo opções de configuração detalhadas que permitem ao cliente alterar a aparência e o comportamento do aplicativo para seus usuários. Apesar de serem idênticas umas às outras no nível do código, cada instância permanece totalmente

Figura 3 – Níveis de maturidade para um SaaS.

Fonte: Adaptada de [Chong e Carraro \(2006\)](#).

isolada das demais.

- No **nível 3**, o fornecedor executa uma única instância que serve a todos os clientes, com metadados configuráveis fornecendo uma experiência de usuário e recursos exclusivos para cada um. Políticas de autorização e de segurança garantem que os dados de cada cliente sejam mantidos separados dos outros clientes e que, da perspectiva do usuário final, não exista qualquer indicativo de que a instância do aplicativo esteja sendo compartilhada entre vários tenants. Essa abordagem elimina a necessidade de fornecer espaço no servidor para tantas instâncias quanto ao número de clientes do fornecedor, permitindo o uso mais eficiente dos recursos de computação do que o nível 2.
- No **nível 4** de maturidade, o fornecedor hospeda vários clientes com carga balanceada de instâncias idênticas, com os dados de cada cliente mantidos separados e com metadados configuráveis fornecendo uma experiência do usuário e um conjunto de recursos exclusivos para cada cliente. Um sistema de SaaS é escalonável para um número de clientes arbitrariamente grande, uma vez que a quantidade de servidores e instâncias no lado do fornecedor pode aumentar ou diminuir conforme necessário.

Multi-tenancy é ampla e não se restringe às aplicações, pois inclui nível de dados e hardware. Por esse motivo, diferentes definições são aceitas tanto na academia quanto na indústria. Considerando essas duas perspectivas, [Kabbedijk et al. \(2015\)](#) realizaram um mapeamento sistemático das definições. Como resultado, foram identificadas 43 definições que contribuíram para que os autores apresentassem uma mais completa:

Multi-tenancy consiste em um padrão arquitetural que permite uma única instância da aplicação ser compartilhada de forma transparente

entre múltiplas organizações, ou tenants, e seus usuários finais. Além disso, oferece a configuração exclusiva conforme as necessidades dos tenants. Recursos do sistema, como serviços, aplicações, banco de dados, ou hardware também são compartilhados com o objetivo de reduzir custos (KABBEDIJK *et al.*, 2015).

A partir dessa definição, alguns pontos podem ser destacados, como: (i) compartilhamento de forma transparente, (ii) configuração exclusiva e a (iii) redução de custos. Multi-tenancy permite que uma única instância da aplicação esteja em execução, possua interfaces e lógicas diferentes que acessam dados distintos para servir múltiplas organizações e usuários. Do ponto de vista do usuário, a aplicação é executada como se estivesse em ambiente dedicado, de modo que havendo falha para um usuário, os demais não devem ser afetados. A existência de outras aplicações sendo executadas na mesma nuvem permanece transparente tanto para prestadores de serviços como para usuários. Isso permite cobrir uma ampla variedade de requisitos de usuários e com isso, os custos para manter uma aplicação SaaS são reduzidos naturalmente até atingir uma relação contínua entre número de usuários e custo, ou seja, tanto o proprietário quanto as organizações que alugam a aplicação, incluindo seus usuários finais são beneficiados. Chong e Carraro (2006) chamam isso de efeito “cauda longa” (do inglês, *long tail*), ao se referir à redução do custo em virtude do crescente número de usuários e posteriormente, atingir um equilíbrio.

Multi-tenancy possibilita alcançar níveis efetivos de economia. Um conjunto de recursos de hardware, relativamente baixo, é necessário para atender às necessidades de todos os usuários. Além disso, um conjunto pequeno de profissionais podem gerir de forma eficiente uma pilha de software e de hardware. Por exemplo, desenvolvedores podem construir e manter uma única base de código sob uma única plataforma (sistema, bancos de dados, etc.). Virtualização e compartilhamento de recursos são os principais aspectos dessa arquitetura para maximizar o uso de recursos e isolar os dados dos tenants.

Com o apoio da multi-tenancy, os tenants podem criar extensões customizadas aos objetos padrões de dados ou criar objetos completamente novos. Dados específicos dos tenants são mantidos de forma segura em um banco de dados compartilhado, de modo que um tenant não tenha acesso aos dados dos demais. Em teoria, um tenant pode personalizar a interface do aplicativo e lógica de negócios em tempo real, sem afetar a funcionalidade ou disponibilidade do aplicativo (SALESFORCE, 2008). Desenvolver um aplicação compilada estaticamente e torná-la executável atendendo tais desafios é uma tarefa difícil. Assim, a natureza de uma aplicação multi-tenant é dinâmica, ou polimórfica, para cumprir as expectativas individuais de vários tenants e seus usuários. Uma única instância da aplicação SaaS é executada atendendo múltiplos tenants, assumindo a utilização de um conjunto de recursos de hardware e software compartilhados. Nessa arquitetura orientada a metadados (SALESFORCE, 2008), tanto os metadados específicos de cada tenant quanto os metadados que descrevem as funcionalidades comuns são transformados em componentes que oferecem algum tipo de funcionalidade relacionada com a lógica de negócios, inclusões ou modificações nas interfaces, entre outras. Por exemplo, um cliente de

um aplicativo de rastreamento de faturas pode exigir que cada fatura seja aprovada por um gerente; um segundo cliente pode exigir que cada fatura seja aprovada por dois gerentes em sequência; um terceiro pode exigir que dois gerentes aprovem cada fatura, permitindo que trabalhem em paralelo. De modo geral, com a alta capacidade de customização oferecida em um SaaS, os clientes podem configurar a aplicação de acordo com suas regras específicas de negócio (CHONG; CARRARO, 2006).

3.3.1 Elementos fundamentais no desenvolvimento

Aplicações estáticas tradicionais não são capazes de atender desafios que são tratados com multi-tenancy. Uma aplicação multi-tenant deve ser dinâmica por natureza ou polimórfica para atender completamente as expectativas de vários tenants e seus usuários (CHONG; CARRARO; WOLTER, 2006). Dessa forma, usuários finais têm a impressão que a aplicação foi construída de forma exclusiva para sua organização, pois existem regras de negócio específicas e as interfaces são customizáveis. Quando na verdade, existem muitos tenants que estão utilizando a mesma instância da aplicação. Implementações de tais aplicações variam significativamente e, por essa razão, decisões apropriadas devem levar em conta os objetivos de mercado e requisitos da aplicação.

Questões relacionadas aos tenants podem afetar decisões em relação à arquitetura de dados. Um modelo de implementação adequado deve ser definido considerando uma escala entre o isolamento e o compartilhamento. Segundo Chong e Carraro (2006) três modelos podem ser destacados: (i) *Bancos de dados Separados* - cada tenant possui sua própria infraestrutura de persistência, ou seja, seu próprio banco de dados; (ii) *Bancos de dados Compartilhados mas Schemas Separados* - os tenants compartilham um banco de dados comum mas cada tenant tem seu conjunto de tabelas (schema) e (iii) *Bancos de dados e Schemas Compartilhados* - dados para todos os tenants são armazenados no mesmo conjunto de tabelas e identificados por meio de uma coluna que discrimina o tenant, ou seja, uma chave é utilizada para identificar os registros de cada tenant. As duas primeiras opções são provavelmente as menos intrusivas, fornecem alto nível de isolamento e não variam significativamente na forma que são implementadas na aplicação. A terceira reduz os custos com recursos e manutenção em grande parte. Entretanto, ela pode ser mais vulnerável do ponto de vista de ameaças à privacidade dos dados, uma vez que as tabelas são utilizadas por todos os tenants.

Quando diferentes bancos de dados são definidos para cada tenant, recursos computacionais e o código da aplicação são geralmente compartilhados entre todos os tenants na nuvem, mas cada tenant possui seu próprio conjunto de dados que permanece logicamente isolado a partir dos dados que pertencem a todos os tenants (CHONG; CARRARO; WOLTER, 2006). Metadados associam cada banco de dados com seu tenant correspondente, e a segurança na conexão com o banco de dados evita que algum tenant acidentalmente ou de forma maliciosa acesse dados de outros tenants. No segundo modelo, cada tenant possui seu conjunto separado de tabelas em uma

base de dados comum. Tenants podem estender o modelo de dados facilmente como se estivesse utilizando uma abordagem totalmente isolada. As tabelas são criadas a partir de um conjunto *default*, mas uma vez que elas são criadas, os tenants não precisam se restringir a esse conjunto, ou seja, podem adicionar ou modificar as colunas e até novas tabelas conforme desejado. Na terceira abordagem, o mesmo banco de dados e o mesmo conjunto de tabelas são utilizadas para hospedar os dados de todos os tenants. Uma única tabela inclui registro de múltiplos tenants armazenados em qualquer ordem, entretanto, é necessário uma coluna “Tenant ID” para associar um dado registro com seu tenant adequado. Essa abordagem pode requerer esforços adicionais de desenvolvimento na área de segurança para garantir que os tenants não acessem os dados dos demais, mesmo em caso de defeitos inesperados ou ataques.

Implementações de sistemas SaaS multi-tenants podem diferenciar significativamente. No entanto, uma característica predominante nesse tipo de desenvolvimento é a identificação dos tenants (KREBS; MOMM; KOUNEV, 2012). Ao construir uma aplicação para atender vários tenants, é necessário isolar os tenants em quase todas as partes da aplicação por meio de filtros. Isso serve para direcionar quais features devem ser utilizadas na composição de uma versão específica de um tenant (*tenant-specific application*). Um identificador de tenant (*tenant identifier*) é geralmente utilizado para identificar cada tenant e seus recursos. Nesse cenário, existem pelo menos três abordagens para apoiar essa identificação que são conhecidas pelos desenvolvedores: (i) utilização de *http headers* customizadas ou OAuth¹⁰ com tokens; (ii) identificador na URI (*http://app.com/tenantid/ ...*) e (iii) identificados no *hostname* (*http://tenantid.app.com*). Na primeira alternativa, a URL pode ser completamente diferente para cada versão disponibilizada pelo sistema SaaS, enquanto na segunda o identificador é incluso no domínio principal. Por fim, na terceira alternativa, o identificador é exibido no início da URI.

Multi-tenancy é uma solução prática quando permite aplicações se tornarem altamente customizáveis e atualizáveis. Entretanto essas capacidades levam a várias questões complexas. De um lado, uma aplicação multi-tenant permite cada tenant criar extensões customizadas para objetos padronizados e inserir novos objetos customizados. Dados específicos dos tenants precisam ser mantidos de forma confiável mesmo se eles estiverem em um banco de dados compartilhado. Administradores de um tenant podem customizar as interfaces e as regras de negócio em tempo real de modo que as mudanças não impactem as funcionalidades ou a disponibilidade da aplicação para os demais tenants. A arquitetura orientada por metadados ou *Meta-data-driven architecture* (WEISSMAN; BOBROWSKI, 2009; DOMINGO *et al.*, 2010) é um conceito importante para apoiar a geração de componentes específicos dos tenants a partir de metadados e as mudanças com praticamente nenhum risco que afetaria os demais clientes. Os metadados podem ser aplicados para associar cada banco de dados com seu tenant correspondente, e a segurança do banco de dados impede que qualquer tenant acesse de forma acidental ou maliciosa os dados dos demais. Um aplicativo orientado por metadados precisa de

¹⁰ <https://oauth.net/2/>

uma separação clara entre os dados da aplicação compilada em tempo de execução, os metadados que descrevem as funcionalidades básicas de uma aplicação e os metadados que correspondem a cada customização e dados de um tenant. Esses limites tornam possível a geração de aplicações específicas para os tenants por meio de uma compilação dinâmica e como resultado, o esforço para integrar as versões e os dados que elas gerenciam pode ser simplificado.

A título de ilustração, dependendo dos requisitos de um sistema SaaS deve-se especificar quais Entidades, *Site properties* e *Timers* devem ser isolados entre os tenants e quais são compartilhados. Uma Entidade deve ser *single-tenant* caso seus dados precisem ser compartilhados por todos os tenants e multi-tenant se for específico de um tenant. Por exemplo, em uma aplicação e-commerce as entidades: *Customer*, *Product* e *Invoice* possuem seus dados isolados por tenant, uma vez que cada tenant possui seus próprios clientes, produtos e faturas. Por outro lado, entidades como *CurrencyExchangeRate* e *Country* têm seus dados compartilhados entre todos os tenants, uma vez que as taxas de câmbio e os países são os mesmos para todos os tenants. Geralmente, entidades estáticas são compartilhadas entre os tenants. Assim como as Entidades, os *Site Properties* permitem especificar se certos dados estão isolados ou compartilhados entre os tenants (OUTSYSTEMS, 2019).

Em uma aplicação e-commerce as organizações de clientes podem ser espalhadas por todo o mundo, portanto, cada tenant deve ter sua própria configuração para a moeda padrão: o valor do *Site Property* “Moeda Padrão” é isolado por tenant. Por outro lado, o número de vezes que um usuário pode perder a senha no processo de log-in é o mesmo para todas as organizações de clientes: o valor do *Site Property* “Número de tentativas” é compartilhado entre os tenants. Os *Timers* permitem especificar se serão executados no escopo de um tenant específico (executar uma vez para cada tenant e manipular apenas dados para esse tenant) ou executar no tenant padrão (manipular dados compartilhados entre os tenants). Por exemplo, em uma aplicação e-commerce cada organização de clientes possui seus próprios clientes e produtos, portanto, um *batch job (timer)* para consolidar dados de relatórios é isolado entre os tenants. Entretanto, como todos os tenants compartilham as mesmas taxas de câmbio, um *timer* para atualizar taxas de câmbio é compartilhado entre todos os tenants. Assim, existem dois casos: (i) o *timer* é executado uma vez para cada tenant. Quando o temporizador é executado em um tenant, ele só tem acesso a dados de propriedade desse tenant e (ii) se o *timer* for executado uma vez sendo capaz de ler e manipular dados compartilhados por todos os tenants (OUTSYSTEMS, 2019).

Atualmente a migração para o modelo de nuvem é considerada como um caminho inevitável para muitas organizações, seja em virtude de evoluções nos modelos de negócio, redução de custo com o compartilhamento de recursos ou outros benefícios como a alta escalabilidade, disponibilidade e alocação dinâmica de recursos computacionais. Entretanto, existem muitos desafios em virtude das particularidades da multi-tenancy para SaaS, como a dinamicidade do ambiente, a alta capacidade de customização e a garantia do isolamento dos dados. Desde um gerenciamento inadequado das sessões por tenant até um defeito crítico em camadas de infraes-

trutura podem comprometer a segurança dos dados ou afetar sua disponibilidade. Além disso, como a computação em nuvem surgiu a partir de outros modelos como computação paralela, computação distribuída e virtualização, possivelmente existem muitas preocupações recorrentes em sistemas SaaS e portanto, esse campo requer mais investigações e definições de abordagens que contribuam com o desenvolvimento de aplicações de alta qualidade.

3.3.1.1 Aplicações web e SOA (Service-Oriented Architecture) versus SaaS

Aplicações desenvolvidas para nuvem podem ser categorizadas com base no nível de multi-tenancy exigido. Aplicações hospedadas na nuvem são aquelas que podem ser executadas na infraestrutura da nuvem. Para essas aplicações, multi-tenancy concentra-se na camada de infraestrutura, ou seja, apenas a infraestrutura deve ser compartilhada pelos provedores para satisfazer múltiplas aplicações e clientes. Aplicações otimizadas para nuvem são as que utilizam o potencial da nuvem. Essas aplicações atendem requisitos rigorosos e oferecem o máximo de retorno sobre o investimento. Nesse caso, a arquitetura multi-tenant é explorada em diferentes camadas, como infraestrutura, aplicação e banco de dados, utilizando a capacidade de uma plataforma PaaS.

O ambiente de computação em nuvem é diferente de um ambiente tradicional em termos de desenvolvimento de aplicações, configuração, execução e gerenciamento. Aplicações tradicionais e sistemas SaaS diferem consideravelmente e podem ser comparadas com base no tipo de usuário, multi-tenancy, segurança, etc. (MAHMOOD; SAEED, 2013). No Quadro 1 descrevem-se as principais diferenças, sob uma perspectiva técnica. Muitas diferenças possuem um impacto direto na atividade de teste. Por exemplo, no caso da multi-tenancy, os tenants podem desenvolver novos componentes ou estender os previamente disponibilizados, ou seja existe a possibilidade de certas mudanças impactarem outras versões destinadas a outros tenants. Desse modo, torna-se necessário estratégias para isolamento de uma versão particular em um ambiente controlado para testar novos componentes (SURESHKUMAR; KANNAN; PURNIMA, 2013; VASHISTHA; AHMED, 2012). Com relação ao tempo de inatividade, mesmo que uma aplicação web seja desenvolvida com controle de versões, ao aplicar alterações no ambiente de produção, provavelmente os usuários ficarão com determinadas propriedades e/ou funcionalidades indisponíveis, ainda que momentaneamente. No caso de aplicações na nuvem, com a alta disponibilidade oferecida pelo provedor de serviços e determinados mecanismos da nuvem, esse tempo pode ser reduzido significativamente. Muitos testes de aplicações desenvolvidas para nuvem podem ser conduzidos em emuladores ou ambientes simulados, mas para garantir que a aplicação sob teste está em conformidade quanto aos requisitos de qualidade esperados é indispensável realizar verificações no ambiente real da nuvem.

Vijayanathan e Shubha (2011) apontam os atributos chave que diferenciam SaaS de aplicações convencionais sob o ponto de vista dos negócios, os quais são descritos no Quadro 2. Custos relativos ao desenvolvimento, incluindo aquisição, instalação e manutenção podem ser

Quadro 1 – Diferenças entre sistemas SaaS e aplicações web tradicionais sob o ponto de vista técnico.

Parâmetros	Aplicações web tradicionais	Sistemas SaaS
Base de usuários	Conhecida em tempo de projeto	Pode não ser conhecida e pode ser dinâmica
Multi-tenancy	Não exigida	Assumida
Segurança	Definida pela arquitetura da aplicação	Contratos de serviços fornecidos pelos provedores da nuvem
Implantação	Apenas por meio de ferramentas tradicionais	Requer conhecimento e utilização de ferramentas e APIs específicas
Tempo de inatividade	Atualizações e melhorias estão associadas com tempo de inatividade	Sem tempo de inatividade
Infraestrutura	Estruturada e controlada	Desestruturada e gerenciada pelo provedor da nuvem
Componentes	Componentes co-alocados no mesmo ambiente	Componentes geralmente estão espalhados em uma ou muitas nuvens
Teste	Em ambiente controlado	Aplicação (integração) é testada na nuvem para assegurar a “orquestração” perfeita entre os serviços de uma ou muitas nuvens

Fonte: Adaptada de [Mahmood e Saeed \(2013, p. 149\)](#).

reduzidos ao utilizar um provedor de serviços na nuvem, uma vez que o custo é proporcional à utilização e uma única instância de uma aplicação pode ser utilizada para atender múltiplas organizações sem que requisitos como a disponibilidade sejam comprometidos. Além disso, manutenções podem ser aplicadas por uma equipe reduzida ([SALESFORCE, 2008](#)). Com relação aos custos operacionais, no modelo SaaS são mais gerenciáveis, uma vez que dados quantitativos relacionados com a demanda por recursos geralmente são disponibilizados pela plataforma ao fornecedor da aplicação. [Vijayanathan e Shubha \(2011\)](#) também destacam o risco de uma aplicação web convencional se tornar de “prateleira”, no sentido de que quando uma aplicação é desenvolvida para ser utilizada por diversas organizações, provavelmente muitas funcionalidades não irão refletir os requisitos de uma organização específica. Mas replicar essa aplicação, realizando alterações implica em ter várias versões e alto custo de manutenção. Por isso, existe o risco do software se tornar de “prateleira” ao manter a mesma versão da aplicação para diversas organizações, assumindo que muitas funcionalidades estejam em conformidade com regras genéricas de negócio. No caso de aplicação SaaS, especificamente as desenvolvidas utilizando a arquitetura multi-tenant esse risco pode ser reduzido, visto que a alta capacidade de customização permite realizar alterações em nível de regras de negócio e interfaces de acordo com as especificações particulares de cada tenant.

Segundo [Abubakr \(2010\)](#), na prática as aplicações web e aplicações disponibilizadas como serviço (SaaS) são similares ao considerar que são oferecidas via Internet. Entretanto, as aplicações web convencionais não oferecem a mesma riqueza de funcionalidades e personalização oferecida em aplicações projetadas para nuvem. Muitos desenvolvedores acreditam

Quadro 2 – Diferenças entre sistemas SaaS e aplicações web tradicionais sob o ponto de vista dos negócios.

Atributos	Aplicações web tradicionais	Sistemas SaaS
<i>Time to market</i>	aplicações são construídas e hospedadas no <i>data center</i> do cliente ou de uma empresa externa	pode começar a sua atividade imediatamente com aplicações prontamente disponíveis que são configuráveis
Custos	envolve alto custo com desenvolvimento	pode reduzir custos no desenvolvimento, no caso dos custos relacionados à manutenção e aquisição
Custos operacionais	preso ao fornecedor com um intervalo de tempo mínimo	maior controle
Riscos de “prateleira” (do inglês, “Shelf ware risk”)	evidente ao gerenciar múltiplas versões	pode ser reduzido com <i>multi-tenancy</i> pelo enfoque na alta capacidade de customização
Custo total de propriedade	alto, principalmente quando são disponibilizadas muitas versões	baixo, visto que custos com atualizações e manutenção são mínimos

Fonte: Adaptada de [Vijayanathan e Shubha \(2011\)](#).

erroneamente que ao desenvolver uma aplicação web e hospedá-la em um provedor de serviços em alguma nuvem, utiliza completamente a capacidade da nuvem, se tornando uma aplicação SaaS. Entretanto, uma aplicação web se torna uma aplicação para nuvem, ao possuir de fato certas características, como: multi-tenancy para apoiar vários requisitos e necessidades dos clientes e tecnologia de virtualização - que desempenha um papel importante na nuvem.

[Laplante et al. \(2008\)](#) apontam as diferenças entre SaaS e *Service-Oriented Architecture* (SOA). A principal diferença é que SaaS consiste em um modelo de entrega de software, enquanto que SOA é um modelo para desenvolvimento. O modelo SaaS separa as propriedades do software a partir do usuário, em que o proprietário é um fornecedor que hospeda o software e permite que o usuário execute o software sob demanda considerando alguma arquitetura do lado do cliente via Internet ou Intranet. No modelo SOA, os componentes constituintes do software são serviços reutilizáveis. Um conjunto de serviços interagem entre si por meio de interfaces padronizadas e protocolos de comunicação. Nesse modelo, o desenvolvimento de sistemas segundo os princípios de SOA requer mudanças na forma como são construídos e como interagem. Essa estratégia arquitetural permite selecionar partes customizadas de software conforme as funcionalidades do negócio correspondente. SOA também pode ser considerada como um framework consistente para conectar essas partes de forma estática e dinâmica.

Apesar dessas diferenças, SaaS e SOA são modelos arquiteturais direcionados para sistemas de informação em larga escala. Ao utilizar o modelo SaaS, um provedor pode oferecer um sistema como um serviço e utilizando SOA, um serviço pode ser adotado como um componente de serviço para construir novos sistemas, os quais também podem ser oferecidos como novos serviços. SaaS pode auxiliar na disponibilização de componentes para utilização no modelo SOA.

O modelo de rede (*network model*) em SOA pode ser visto como uma lista de serviços potenciais a serem utilizados para o desenvolvimento de um sistema, enquanto que no modelo SaaS tem-se uma lista de serviços de negócio a serem fornecidos.

Para os engenheiros, SOA consiste em um modelo arquitetural que descreve padrões para interações entre componentes de serviços, enquanto que SaaS descreve padrões de interações entre os componentes constituintes que não são necessariamente serviços. Em ambos modelos, tem-se uma alta dependência de tecnologia em termos de plataforma. No modelo SaaS, o desenvolvedor precisa decidir qual plataforma utilizar para implementar os serviços e ao desenvolver uma aplicação no modelo SOA, torna-se necessário optar pela tecnologia para desenvolver *web services*. Além disso, linguagens e protocolos precisam ser definidos, tais como: WSDL para descrição, UDDI para publicação e SOAP para comunicação. Monitoramento, gerenciamento de comunicações, verificação das interações entre componentes são exigidas nos dois modelos.

3.4 Desenvolvimento de um sistema SaaS multi-tenant

A dificuldade em encontrar soluções completas, disponíveis e que evidenciassem os principais elementos da arquitetura multi-tenant levaram ao desenvolvimento de um sistema SaaS no segmento de e-commerce chamado “*MtShop*”. As seguintes tecnologias foram utilizadas: Spring Boot¹¹, AngularJS¹², Bootstrap¹³, jQuery¹⁴, Cloudinary¹⁵ para gerenciamento de imagens na nuvem, JPA, MySQL, Maven, Hibernate e Java. Essa aplicação encontra-se em processo de registro junto ao INPI (Instituto Nacional da Propriedade Industrial) via Agência USP de Inovação¹⁶. Em relação aos níveis de maturidade (CHONG; CARRARO, 2006), o *MtShop* pode ser classificado no nível 3 em que uma única instância da aplicação serve a todos os tenants. Informações gerais sobre a estrutura e complexidade do projeto foram calculadas pela ferramenta SonarQube¹⁷ e os dados são apresentados na Tabela 1.

Para a identificação de tenants no *MtShop* o tenant ID foi incluído na URI para tornar mais simples o lado do cliente e quanto a persistência, bancos de dados isolados são utilizados em cada versão da aplicação. Na Figura 20 ilustra-se um diagrama entidade relacionamento (ER) para a aplicação. Os produtos disponíveis no e-commerce são visualizados para qualquer visitante, mas para adicionar algum produto ao carrinho, ele precisa se registrar como cliente (*Customer*) e realizar o login. Um cliente pode realizar checkout, revisar seu pedido, submeter pedido e realizar logout. Caso um usuário possua privilégios de “administrador”, então ele poderá visualizar a lista de clientes, gerenciar os usuários e ainda realizar operações CRUD (acrônimo

¹¹ <http://projects.spring.io/spring-boot/>

¹² <https://angularjs.org/>

¹³ <http://getbootstrap.com/>

¹⁴ <https://jquery.com/>

¹⁵ <http://cloudinary.com/>

¹⁶ <http://www.inovacao.usp.br/>

¹⁷ <https://www.sonarqube.org/>

Tabela 1 – Informações sobre o *MtShop* coletadas por meio da SonarQube.

Code lines	
Java	2.3 k
CSS	133
JavaScript	132
JavaServer Pages (JSP)	913
XML	171
Complexity	
Complexity/classes	5.1
Complexity/method	2.4
Implementation units	
Number of classes	59
Number of methods (Java)	331

de *Create, Read, Update e Delete*) para os produtos (*Products*). Um cliente pode incluir alguns itens em seu carrinho (*Cart*), os quais são definidos como “item pedido” (*CartItem*). Feito isso, ele pode revisar seu pedido (*Order*) e posteriormente, submetê-lo. Uma instância do carrinho pode ter diversos itens e cada um deles refere-se a somente um produto.

Figura 4 – Diagrama entidade relacionamento para o *MtShop*.

Na Figura 5 ilustram-se as principais camadas do *MtShop*: *Application container*, *View (templates)* e *Database*.

1. *Application container* concentra a principal parte da aplicação que foi desenvolvida com o apoio do Spring Boot, inclui classes para definição dos modelos, classes que apoiam as operações CRUD de objetos, classes para mapeamento das requisições com métodos e assim por diante. A maioria dessas classes foram implementadas utilizando anotações e considerando o padrão *Model-View-Controller* (MVC). Partindo das requisições dos tenants na camada *Controller* é possível identificar o tenant e fornecer a versão da aplicação

Figura 5 – As principais camadas do MtShop.

correspondente. Para isso, um mecanismo chamado “*Multi-tenancy Interceptor*” intercepta as requisições e captura o tenant ID.

2. *View*. Com o tenant identificado, uma template engine (*multitemplating*) como Thymeleaf¹⁸ pode ser empregada para realizar uma composição dinâmica dos fragmentos/features que devem gerar a versão da aplicação esperada pelo tenant. No caso do *MtShop*, apenas templates JSP nativos foram utilizados para encapsular os layouts das páginas web e realizar a composição dos fragmentos. Usuários do “Tenant 1” podem acessar uma aplicação diferente quando comparada com a aplicação designada para o “Tenant 2”, em termos de interfaces e features específicas.
3. *Database*. Uma das principais preocupações nesse processo de composição de versões está vinculada à metodologia definida para a persistência de dados. Com a mesma importância para a camada anterior, a identificação do tenant permite estabelecer a conexão com o banco de dados ou schema adequado. Um mecanismo chamado “*Multi-tenant connection provider*” foi implementado com essa finalidade que utiliza meta-dados para as conexões. Na Figura 6 descrevem-se parte dos meta-dados usados para as configurações de conexão (em fase de desenvolvimento, por isso a conexão é local) para o “*datasource1*”, utilizadas pelo tenant 1. Aliado ao “*Multi-tenancy Interceptor*”, o “*Multi-tenant connection provider*” é responsável por estabelecer a conexão com o banco de dados ou schema designado para todos os tenants ou para um específico.

Embora o número de tenants que pode ser configurado para o *MtShop* seja ilimitado, três versões são fornecidas atualmente com os seguintes identificadores: “tenant_1”, “tenant_2” e “tenant_3”. Esses identificadores nos permite acessar lojas diferentes: “Victor’s Motorcicle”,

¹⁸ <https://www.thymeleaf.org/>

```

...
spring.multitenancy.datasource1.url=jdbc:mysql://localhost:3306/tenant_1
spring.multitenancy.datasource1.username=root
spring.multitenancy.datasource1.password=admin
spring.multitenancy.datasource1.driver-class-name=com.mysql.jdbc.Driver
...

```

Figura 6 – Configurações de conexão para o tenant 1 no *MtShop*.

“Women’s Clothing & Fashion” e “Electronics Online Stores”, respectivamente. Vale ressaltar que para o *MtShop*, o número de tenants foi definido pelo desenvolvedor e isso é uma feature “hard coded”, ou seja, dados e propriedades não podem ser modificadas sem modificar a aplicação. Assim, um novo tenant pode ser incluído na aplicação desde que certas mudanças sejam aplicadas, tais como: inclusão de um novo identificador para o tenant, propriedades de conexão e uso predicativo desse novo identificador nos filtros da aplicação. Na Figura 7 ilustram-se as páginas iniciais das versões fornecidas a partir de um única instância em execução.

Figura 7 – Páginas iniciais das três lojas (versões) configuradas no *MtShop*.

3.4.0.1 Interfaces e funcionalidades

Com relação as interfaces, as versões são ligeiramente distintas em termos funcionais, apesar de utilizarem determinadas unidades de implementação específicas para cada tenant. A aplicação foi desenvolvida compreendendo a possibilidade de aplicar filtros de modo que requisitos de interfaces e de regras de negócio possam se restringir a uma determinada versão. A seguir ilustram-se as principais telas e funcionalidades da versão designada ao “tenant 1”.

Na Figura 8 ilustra-se uma lista dos produtos disponíveis para compra contendo as informações básicas de cada um dos mesmos, tais como: nome, categoria, condição e preço. Com um clique no ícone “info” na última coluna é possível visualizar detalhes de cada produto.

Figura 8 – Página de lista de produtos disponíveis em uma das versões do *MtShop*.

Na Figura 9 tem-se a visualização dos detalhes do primeiro produto exibido anteriormente. Essa tela é acessível para qualquer usuário ou visitante. Entretanto, as funcionalidades “Order Now” e “View Cart” são funcionais somente para usuários que tenham realizado o login. Na primeira funcionalidade é possível incluir o produto no carrinho de compras e na segunda, visualizar todos os itens incluídos no carrinho.

Figura 9 – Página de visualização de dados de produtos em uma das versões do *MtShop*.

Na Figura 10 ilustra-se a tela referente ao carrinho de compras. Nessa interface é possível excluir todos os itens do carrinho por meio do botão “Clear Cart”, remover um item específico do carrinho com o botão “remove”, voltar para a lista de produtos com o botão “Continue Shopping” ou enviar o pedido com um clique no botão “Check out”. Nessa última operação, o usuário tem acesso a uma tela de confirmação do pedido.

Figura 10 – Página do carrinho de compras em uma das versões do *MtShop*.

Na Figura 11 descrevem-se os detalhes do pedido para confirmação, como endereços do usuário (faturamento e envio), itens do pedido, data do pedido e o valor total do pedido. Nessa tela as seguintes operações são fornecidas: voltar ao carrinho de compras por meio do botão “Back”, cancelar o pedido com o botão “Cancel” ou confirmar o pedido com o “Submit Order”.

Figura 11 – Página de confirmação do Pedido em uma das versões do *MtShop*.

Ao confirmar o pedido, uma tela de confirmação é exibida ao usuário com a mensagem: “*Thank you for your business!*”, conforme ilustrada na Figura 12.

Na Figura 13 ilustra-se a tela de registro de usuários, nela é possível informar dados básicos como usuário e senha, bem como informações de endereço para faturamento e entrega.

Na Figura 14 ilustra-se a tela de login. Vale ressaltar que na autenticação em qualquer uma das versões, identifica-se o perfil do usuário com o intuito de redirecioná-lo para uma página com funcionalidades administrativas ou para página inicial da aplicação com a informação no menu indicando que o usuário está autenticado e consequentemente pode realizar compras.

Figura 12 – Página de confirmação da compra em uma das versões do *MtShop*.

The screenshot displays a 'Register Customer' form. The top navigation bar is dark with 'Home', 'Products', and 'About Us' on the left, and 'Login' and 'Register' on the right. The form title is 'Register Customer' with a subtext 'Please fill in your information below:'. The form is divided into two main sections: 'Basic Info' and 'Billing address'. The 'Basic Info' section contains fields for 'Name' (filled with 'Victor Santiago'), 'Email' (filled with 'victorhdb@gmail.com'), 'Phone' (filled with '3538219864'), 'Username' (filled with 'victorhugo'), and 'Password' (filled with '*****'). The 'Billing address' section contains a 'Street name' field (filled with 'Rua Raimunda Marques') and an 'Apartment Number' field which is currently empty.

Figura 13 – Página de registro de usuários em uma das versões do *MtShop*.

Adicionalmente, as rotas (URLs) que requerem “perfil de administração” são configuradas para não permitir que um usuário com “perfil de cliente” ou um visitante consiga acessar.

Ao acessar a página de login e informar os dados corretos de um usuário com perfil de administração ocorre um redirecionamento para uma página contendo várias funcionalidades de gerenciamento de produtos e a clientes. Na Figura 15 tem-se a tela administrativa que permite a inclusão, edição e remoção de produtos.

Figura 14 – Página de login em uma das versões do *MtShop*.

Figura 15 – Página de gerenciamento de produtos (Administrador) em uma das versões do *MtShop*.

3.4.0.2 Trechos de código representativos

Para ilustrar os principais mecanismos para multi-tenancy na aplicação, apresentam-se a seguir alguns trechos de código das funções do sistema. No Código-fonte 1 ilustra-se um trecho da classe *MultiTenancyInterceptor*, especificamente o método *preHandle()* que intercepta todas as requisições da aplicação para extração do tenant ID, possibilitando os subsequentes usos desse identificador por uma respectiva versão da aplicação. A variável *pathVars* mapeia valores para chaves provenientes das requisições (linha 5), enquanto a variável “*auth*” recebe valores referentes a autenticação da sessão corrente (linha 6). No trecho das linhas 7 a 20 tem-se condições para invalidar a sessão caso um usuário esteja autenticado mas acessa uma URL com um identificador diferente do qual sua conta está vinculada. A variável *userDetails* contém

dentre outras informações, o valor do tenant ID, obtida com o *getTenant()*, e com isso é possível comparar se um usuário previamente autenticado está tentando acessar uma versão da aplicação destinada a outro tenant.

Caso um usuário autenticado se mantenha na versão designada para o tenant ao qual faça parte, ocorre apenas uma redefinição da variável de identificação do tenant (linha 4) partindo do tenant ID proveniente da autenticação do usuário. Isso permite que o fluxo de execução da aplicação tenha continuidade, fornecendo uma resposta às requisições do usuário, mantendo-se com o uso do identificador adequado. De outro lado, caso *auth* for nulo e *pathVars* conter o valor para *tenantid* ocorre também uma (re)definição da variável de identificação no contexto de execução. Isso precisa ocorrer para que a aplicação consiga determinar qual *datasource* (banco de dados ou schema) deve conectar, quais features devem ser utilizadas na composição da versão e quais unidades de implementação vinculadas às interfaces devem ser admitidas.

Código-fonte 1 – Trecho de código da classe *MultiTenancyInterceptor*

```
1: public class MultiTenancyInterceptor extends
    HandlerInterceptorAdapter {
2:
3: @Override
4: public boolean preHandle(HttpServletRequest req,
    HttpServletResponse res, Object handler) throws Exception {
5: Map<String, Object> pathVars = (Map<String, Object>) req.
    getAttribute(HandlerMapping.URI_TEMPLATE_VARIABLES_ATTRIBUTE
    );
6: Authentication auth = SecurityContextHolder.getContext().
    getAuthentication();
7: if (auth != null && !auth.getClass().equals(
    AnonymousAuthenticationToken.class)) {
8:     CustomUserDetails userDetails = (CustomUserDetails) auth.
    getPrincipal();
9:     if(!userDetails.getTenant().equals(String.valueOf(pathVars.
    get("tenantid")))){
10:         res.sendError(HttpServletResponse.SC_UNAUTHORIZED, "
    Authentication Failed");
11:         SecurityContextHolder.clearContext();
12:         HttpSession session = req.getSession(false);
13:         if (session != null) {
14:             session.invalidate();
15:         }
16:         TenantContextHolder.setTenantId(userDetails.getTenant());
17:     }
```

```
18:         if (pathVars.containsKey("tenantid")) {
19:             TenantContextHolder.setTenantId(String.valueOf(pathVars
                .get("tenantid")));
20:         }
21:     return true;
22: }
```

No Código-fonte 2 ilustra-se um trecho do código da classe *TenantContextHolder* que apoia a definição do tenant corrente por meio da chamada do método *setTenantId()* (linhas 16 e 19 do Código-fonte 1) após a extração do identificador proveniente da URL, conforme comentado anteriormente. A classe *ThreadLocal* nos permite armazenar dados que serão acessíveis apenas por uma thread específica por tenant. No contexto do *MtShop*, essa classe é usada para guardar o identificador na variável *CONTEXT* e com esse valor, manter o acesso de um determinado usuário restrito aos recursos do tenant ao qual pertence.

Código-fonte 2 – Trecho de código da classe *TenantContextHolder*

```
1: public class TenantContextHolder {
2:     private static final ThreadLocal<String> CONTEXT = new
        ThreadLocal<>();
3:     public static void setTenantId(String tenant) {
4:         CONTEXT.set(tenant);
5:     }
6:     public static String getTenant() {
7:         return CONTEXT.get();
8:     }
9:     public static void clear() {
10:        CONTEXT.remove();
11:    }
12: }
```

Garantir a conexão com o datasource correto é uma das principais questões em uma aplicação multi-tenant. No Código-fonte 3 tem-se um trecho da classe *DataSourceBasedMultiTenantConnectionProviderImpl*. O objetivo dessa classe é definir associações via *HashMap* entre um identificador válido e um datasource que contenha todas as informações para conexão com um banco de dados, como exibido no método *load()*. O método *selectAnyDataSource()* serve para conectar a um dos datasources válidos quando a aplicação ainda não recebeu alguma requisição que conduza a uma definição da variável de identificação do tenant, por esse motivo utiliza-se o valor de uma variável global chamada *DEFAULT_TENANT_ID*. No caso do *MtShop* o uso dessa variável é indiferente, pois foi definida para uso em futuras versões da aplicação.

Outro método importante desta classe é o *selectDataSource()* que recebe como parâmetro

o valor do Tenant ID capturado pelo *MultiTenancyInterceptor*. O objetivo desse método é fornecer o datasource adequado que está previamente associado a um determinado identificador (método *load()*).

Código-fonte 3 – Trecho de código da classe *DataSourceBasedMultiTenantConnectionProviderImpl*

```
1: @Component
2: public class DataSourceBasedMultiTenantConnectionProviderImpl
    extends
        AbstractDataSourceBasedMultiTenantConnectionProviderImpl {
3: private static final long serialVersionUID =
    8168907057647334460L;
4:
5: @Autowired
6: private DataSource dataSource1;
7: @Autowired
8: private DataSource dataSource2;
9: @Autowired
10: private DataSource dataSource3;
11: private Map<String, DataSource> map;
12:
13: @PostConstruct
14: public void load() {
15:     map = new HashMap<>();
16:     map.put("tenant_1", dataSource1);
17:     map.put("tenant_2", dataSource2);
18:     map.put("tenant_3", dataSource3);
19: }
20: @Override
21: protected DataSource selectAnyDataSource() {
22:     return map.get(DEFAULT_TENANT_ID);
23: }
24: @Override
25: protected DataSource selectDataSource(String tenantIdentifier)
    {
26:     return map.get(tenantIdentifier);
27: }
28: }
```

Com a versão do sistema designada a um tenant específico conectada ao banco de dados/schema esperado, parte-se para o atendimento das requisições. Diversas classes da camada

Controller atuam para fornecer as seguintes páginas: home, about, login, detalhes de um produto, lista de produtos, registro de clientes, registro de produtos, administração de clientes e produtos, carrinho de compras e confirmação do pedido. No Código-fonte 4 tem-se um trecho da classe *CartController*. Na linha 2 define-se um mapeamento dessa classe com todas as requisições contendo a URL: “/tenantid/customer/cart”, ou seja, é esperado o valor para o tenant ID acrescido de “/customer/cart” para que essa classe seja acionada. Nas linhas 7-13 tem-se um método que dado um tenant ID e um usuário autenticado (linha 9), recupera-se da base de dados o ID para o carrinho (*cartId*) com a chamada “*customer.getCart().getCartId()*”. Por fim, esse método realiza um redirecionamento para “tenantid/customer/cart/cartID”, mantendo o identificador do tenant e incluindo o ID para o carrinho.

Nas linhas posteriores (15-20), o método *getCartRedirect()* retorna a página “cart” (cart.jsp) e nesse processo, incorpora ao “*model*” (linhas 17-18) os valores de tenant ID e o ID do carrinho. Assim como na linha 11, o objeto *model* recebe como atributos os objetos que serão utilizados na renderização da página.

Código-fonte 4 – Trecho de código da classe *CartController*

```
1: @Controller
2: @RequestMapping("/{tenantid}/customer/cart")
3: public class CartController {
4:     @Autowired
5:     private CustomerService customerService;
6:
7:     @RequestMapping
8:     public String getCart(@PathVariable String tenantid,
9:         @AuthenticationPrincipal CustomUserDetails activeUser, Model
10:         model){
11:         Customer customer = customerService.getCustomerByUsername(
12:             activeUser.getUsername());
13:         int cartId = customer.getCart().getCartId();
14:         model.addAttribute("tenantid", tenantid);
15:         return "redirect:/{tenantid}/customer/cart/"+cartId;
16:     }
17:
18:     @RequestMapping("/{cartId}")
19:     public String getCartRedirect(@PathVariable String tenantid,
20:         @PathVariable(value = "cartId") int cartId, Model model){
21:         model.addAttribute("cartId", cartId);
22:         model.addAttribute("tenantid", tenantid);
23:         return "cart";
24:     }
```

21: }

No Código-fonte 5 tem-se um trecho da classe *CustomAuthenticationToken* que apoia os módulos de autenticação da aplicação. A ideia nessa classe é fornecer um token de autenticação que inclua o identificador do tenant, ao invés de conter somente o “username” e uma chave hash da senha. Isso deve-se a várias razões, como a possibilidade de um usuário autenticado em uma versão designada a um “tenant A”, tentar acessar a versão definida para um “tenant B” e permanecer autenticado e assim, obter acesso à informações restritas. Como há somente uma instância em execução sendo compartilhada entre os tenants, medidas como essa auxiliam na privacidade dos dados que devem ser mantidos isolados. Adicionalmente, para cada schema existe uma tabela que contém os dados dos usuários, senhas criptografadas (por meio do *BcryptPasswordEncoder*¹⁹) e uma coluna que contém o valor do identificador do tenant.

Código-fonte 5 – Trecho de código da classe *CustomAuthenticationToken*

```
1: public class CustomAuthenticationToken extends
    UsernamePasswordAuthenticationToken {
2:     private static final long serialVersionUID = 1L;
3:     private String tenant;
4:
5:     public CustomAuthenticationToken(Object principal, Object
        credentials, String tenant) {
6:         super(principal, credentials);
7:         this.tenant = tenant;
8:         super.setAuthenticated(false);
9:     }
10:    public CustomAuthenticationToken(Object principal, Object
        credentials, String tenant,
11:        Collection<? extends GrantedAuthority> authorities) {
12:        super(principal, credentials, authorities);
13:        this.tenant = tenant;
14:        super.setAuthenticated(true); // must use super, as we
            override
15:    }
16:    public String getTenant() {
17:        return this.tenant;
18:    }
19: }
```

¹⁹ <https://tinyurl.com/y6dyktdn>

O *MtShop* foi desenvolvido para constituir embasamento técnico e apoiar a compreensão dos elementos fundamentais da arquitetura multi-tenant. Atualmente, a aplicação encontra-se implantada no ambiente de nuvem *Heroku*²⁰, uma plataforma que permite criar, executar e operar aplicativos completamente na nuvem. A aplicação é extensível e a inclusão de novas funcionalidades podem torná-la comercial.

3.5 Considerações finais

Multi-tenancy é um padrão arquitetural que permite fornecedores de software atender múltiplos clientes e compartilhar recursos de software de forma transparente para reduzir custos. Além disso, permite configurar o sistema de forma exclusiva às necessidades específicas de cada organização partindo de um único produto em execução. Para isso, a virtualização e o compartilhamento de recursos são vistos como elementos essenciais para maximizar a utilização de recursos. As soluções SaaS empregam essa arquitetura amplamente pois maximiza os retornos, evita a preocupação com o gerenciamento de múltiplas versões em locais dedicados e ainda permite que cada cliente utilize um software de acordo com suas necessidades e com a impressão de que foi construído de forma exclusiva. Apesar dessas vantagens serem bem vistas no mundo competitivo de software, existe uma série de desafios que estão ligados à complexidade por trás desse modelo de aplicações disponibilizadas como um serviço.

A complexidade não se resume à dinamicidade do ambiente de nuvem mas também fatores como isolamento de dados e estratégias para garantia de qualidade. Como as estruturas são compartilhadas é importante garantir que uma dada execução de uma versão não gere impactos às destinadas a outros tenants. Assim, desde o atendimento de uma requisição gerada por um acesso de um usuário final até as operações mais críticas e específicas de um tenant precisam ser verificadas pois defeitos podem gerar sérios efeitos colaterais e colocar em risco os dados de outros tenants. De posse disso, no próximo capítulo apresentam-se um panorama sobre o teste de sistemas SaaS, os resultados de um Mapeamento Sistemático da Literatura para identificar as tendências das pesquisas e desafios nesse domínio e uma discussão sobre os principais trabalhos relacionados. Uma classificação de defeitos para esse tipo de sistemas também é discutida partindo de seus elementos essenciais.

²⁰ <https://heroku.com>

TESTE DE SOFTWARE APLICADO A SISTEMAS SAAS MULTI-TENANTS E UMA CLASSIFICAÇÃO PRELIMINAR DE DEFEITOS

4.1 Considerações iniciais

Apesar do crescente número de pesquisas envolvendo o teste de aplicações na nuvem, desafios e oportunidades, a maioria das propostas é voltada para requisitos não funcionais, como desempenho, segurança e escalabilidade (GHUMMAN; LASSIG, 2013; INCKI; ARI; SÖZER, 2012; GAO; BAI; TSAI, 2011). Isso indica uma carência por trabalhos direcionados aos aspectos funcionais (SARASWATHI; BHUVANESWARI, 2014). As aplicações desenvolvidas para nuvem possuem particularidades que incluem não somente o ambiente, como também a forma como são desenvolvidas e oferecidas aos usuários finais. Dentre essas diferenças, podem-se destacar: a multi-tenancy, virtualização, alta capacidade de customização e a forte dependência de uma PaaS específica não somente para executar, como também desenvolver aplicações. Muitos fatores podem implicar em desafios para a condução da atividade de teste. Portanto, uma taxonomia de defeitos para esse tipo de sistemas é fundamental para orientar a condução dos testes e priorização de elementos críticos.

Neste capítulo são apresentadas classificações para o teste de sistemas SaaS, um Mapeamento Sistemático da Literatura, os principais desafios no campo de teste que foram identificados e os principais trabalhos envolvendo o teste desses sistemas. Posteriormente, discute-se sobre a relação entre programas concorrentes e sistemas multi-tenants, uma vez que existem certas similaridades exploradas neste trabalho. Uma classificação preliminar de defeitos também é apresentada, uma das principais contribuições originadas desta tese.

4.2 Teste de Software aplicado a sistemas SaaS multi-tenants

Com o avanço da computação em nuvem, cada vez mais sistemas SaaS são desenvolvidos para fornecer diversos serviços. Essa tendência implica em uma alta demanda por testes para esse tipo de sistemas que assegurem algum nível de confiabilidade. Segundo [Vijayanathan e Shubha \(2011\)](#), o teste de sistemas SaaS compreende a validação de sistemas SaaS com respeito a vários aspectos: fluxos de trabalho, multi-tenancy, integridade, confiabilidade, facilidade para desenvolvimento, escalabilidade, disponibilidade, precisão, implantação, usabilidade, testabilidade, portabilidade, tolerância a falhas e atualização em tempo de execução.

[Riungu, Taipale e Smolander \(2010\)](#) coletaram algumas definições para teste na nuvem sob uma perspectiva industrial: *“Teste na nuvem está alinhado com o conceito de nuvem e SaaS. O teste fornece a capacidade de utilizar a capacidade da nuvem e isso implica nos mesmos benefícios que a nuvem traz para os clientes”*; *“Testes na nuvem emprega ambientes de computação em nuvem e procura simular o tráfego de um usuário do mundo real como um meio de carga ou teste de estresse em web sites”* e *“Teste na nuvem é a resposta para o teste de desempenho não apenas realístico que se origina na infraestrutura de um dos clientes. Quando esse tipo de teste é utilizado, pode-se tirar proveito do hardware e largura de banda, as quais simulam as condições reais”*.

[Gao et al. \(2013\)](#) sugerem uma definição mais completa:

O teste de sistemas SaaS envolve diferentes tipos de atividades de validação em um processo para assegurar a qualidade do SaaS em uma infraestrutura de nuvem, fornecendo o especificado por meio de serviços sob demanda em uma infraestrutura de nuvem. Da mesma forma que no teste de software convencional, o processo de teste de um SaaS deve validar as funções dos componentes, integração e serviços funcionais, bem como requisitos não funcionais, tais como: desempenho, confiabilidade, disponibilidade, escalabilidade, segurança e tolerância a falhas. Contudo, o teste de SaaS precisa garantir a qualidade incluindo multi-tenancy, customização/configuração, compatibilidade das interfaces, conectividade e atualizações contínuas e dinâmicas ([GAO et al., 2013](#)).

O ambiente para teste de sistemas SaaS precisa atender requisitos como elasticidade e escalabilidade na infraestrutura com fornecimento automático de recursos; apoiar a execução e validação automatizada; disponibilizar serviços para *upload/download* e migração de todo material utilizado para a condução da atividade de teste e permitir a simulação e geração de carga. Em geral, para essas aplicações os testes de sistema são os mais utilizados ([MAHMOOD; SAEED, 2013](#)). No Quadro 3 os mais conhecidos são sintetizados.

De acordo com o grupo *Software Testing in the Cloud* (STITC) ([MAHMOOD; SAEED, 2013](#) apud [STITC, 2013](#)), existem três categorias de teste na computação em nuvem: teste na

Quadro 3 – Testes utilizados em aplicações na nuvem.

Testes	Descrição
Integração do sistema/ Aceitação do usuário	A plataforma da nuvem precisa ser integrada com todas os serviços de plataformas e infraestruturas de modo que os usuários possam construir seus dados online.
Interoperabilidade	Capacidade de mover aplicações residentes na nuvem de uma infraestrutura para outra. Muitas aplicações precisam ter a capacidade de operar em múltiplas plataformas e ambientes.
Desempenho	A elasticidade para as aplicações é fundamental pois permite utilizar o mínimo de recursos e produzir o máximo de utilização para os usuários finais. O desempenho da nuvem deve permanecer mesmo com o aumento de requisições.
Estresse/recuperação de falhas	No caso de ocorrência de falhas, o tempo de recuperação deve ser o menor possível. Os serviços devem ser recuperadas com mínimos efeitos colaterais ao negócio do cliente.
Teste de segurança	Acesso não autorizado a dados deve ser estritamente proibido. Integridade e segurança dos dados compartilhados devem ser mantidos.

Fonte: Adaptada de [Mahmood e Saeed \(2013, p 150-151\)](#).

nuvem (*Testing in the cloud*), teste da nuvem (*Testing of the cloud*) e migração de teste para nuvem.

Teste na nuvem: recursos fornecidos por uma infraestrutura de computação em nuvem são empregados para facilitar a execução simultânea de casos de teste em um ambiente virtualizado, ou seja, consiste no uso da nuvem para o teste. Esse tipo de teste tem sido muito utilizado para o teste de configuração e de carga;

Teste da nuvem: o teste da nuvem envolve o teste de aplicações que são desenvolvidas para um ambiente de nuvem;

Migração de teste para nuvem: a migração do teste para nuvem refere-se a mover o processo, artefatos e infraestruturas para facilitar testes na nuvem ou da nuvem. Conduzir essa migração não é necessariamente a melhor solução para todas as dificuldades no teste. Custos, as características da aplicação sob teste e os tipos de testes necessários para a aplicação precisam ser consideradas ([PARVEEN; TILLEY, 2010](#)).

[Gao, Bai e Tsai \(2011\)](#) complementam essas categorias, discutindo quatro formas de teste de sistemas SaaS, cada uma com enfoques e objetivos diferentes.

Teste de sistemas SaaS na nuvem: garante a qualidade de um SaaS em uma nuvem com base em seus requisitos funcionais e não funcionais;

Teste de uma nuvem: a qualidade de uma nuvem é validada a partir de uma visão externa levando em conta os recursos e capacidades disponibilizadas pela nuvem;

Teste dentro de uma nuvem (*Testing inside a cloud*): nessa categoria, os testes verificam a qualidade de uma nuvem a partir de uma visão interna, incluindo infraestruturas e capacidades específicas da nuvem. Somente provedores da nuvem podem executar esse tipo de teste, pois possuem acesso a essas infraestruturas e funcionalidades internas de gerenciamento;

Teste sobre a nuvem (*Testing over cloud*): geralmente esses testes são realizados pelos fornecedores de aplicações. Nessa categoria o teste está diretamente relacionado com a verificação de requisitos e especificações de serviços em nível de aplicação.

A partir dessa classificação, os autores descrevem três tipos de atividades:

Teste dirigido ao SaaS: esse tipo de atividade de teste é geralmente realizado dentro da nuvem por engenheiros da nuvem ou provedores de um sistema SaaS. O objetivo principal é assegurar a conformidade das funções. Esses engenheiros realizam testes unitários, teste de integração, validação das funções do sistema e teste de regressão, bem como avaliação de desempenho e escalabilidade. Como as nuvens e os sistemas SaaS fornecem determinadas APIs de serviço e interfaces de conectividade para seus clientes, uma tarefa necessária para os engenheiros consiste em validar tais APIs e a conectividade no ambiente da nuvem;

Teste de aplicações em tempo real na nuvem: geralmente, esse tipo de atividade de teste serve para verificar em tempo real os sistemas em uma nuvem, sob o ponto de vista do tráfego e acesso dos usuários em larga escala. A prática de utilizar a capacidade da nuvem para esses fins é comum, uma vez que o ambiente de nuvem proporciona vários recursos computacionais de forma escalável;

Teste de aplicações sobre a nuvem: esse tipo de teste refere-se as atividades para garantir a qualidade dos relacionamentos de uma aplicação em diferentes nuvens. Quando as aplicações são desenvolvidas para serem implantadas e executadas em diferentes nuvens, novas tarefas de teste são necessárias para garantir a sua qualidade. Diferente dos tipos anteriores, o objetivo principal desse tipo é caracterizado por assegurar a qualidade “ponta a ponta” da aplicação sobre as nuvens. Assim, a integração em nível de sistema, validação das funcionalidades, avaliação de desempenho e verificação da escalabilidade, devem lidar com diferentes tecnologias na nuvem. Isso dificulta as tarefas de verificação de compatibilidade do sistema, interoperabilidade e conectividade.

As tarefas pertencentes a esses três tipos de teste são detalhadas no Quadro 4, considerando o enfoque dos testes de funções de serviço; teste de integração; teste de APIs e conectividade; teste de escalabilidade e desempenho; teste de segurança; teste de compatibilidade e interoperabilidade; e teste de regressão.

Quadro 4 – Visão comparativa dos principais tipos de teste para aplicações na nuvem.

	Enfoque do teste	Teste dirigido ao SaaS dentro da nuvem	Teste de aplicações disponíveis em tempo real (<i>online</i>) na nuvem	Teste de aplicações sobre a nuvem
Teste de funções de serviço	Funções de serviço baseadas em APIs e interfaces	Teste do sistema SaaS/funções relativas à nuvem	Teste em tempo real de funções de serviços em aplicações na nuvem	Teste de funções de serviço de aplicações sobre a infraestrutura da nuvem
Teste de integração	Interações do SaaS e conexões da nuvem	Componentes específicos de provedores e integrações de serviços em uma nuvem privada ou pública	Integração entre clientes em tempo real e servidores <i>back-end</i> na nuvem	- Integração ponta a ponta de aplicações em várias nuvens - Integração com aplicações legadas
Teste de APIs e conectividade	Interfaces de APIs e conectividade de protocolos	APIs de SaaS/nuvem e teste de conectividade na nuvem	Teste de APIs de serviços centralizados no usuário	Teste de APIs de serviços na aplicação e conectividade envolvendo várias nuvens
Teste de escalabilidade e desempenho	Escalabilidade e desempenho baseando-se em SLA	Teste de escalabilidade e desempenho do SaaS/nuvem segundo o SLA	Teste na nuvem considerando a perspectiva dos usuários	Escalabilidade e desempenho ponta a ponta em nível de sistema no ambiente da nuvem de acordo com SLA
Teste de segurança	Dados do SaaS, processos, funções e privacidade dos usuários	Segurança nas funcionalidades fornecidas pelo SaaS/nuvem e privacidade dos usuários na nuvem	Segurança dirigida ao usuário e privacidade na nuvem	Segurança em nível de sistema de ponta a ponta sobre as nuvens
Teste de compatibilidade e interoperabilidade	Validação das interfaces com os usuários e compatibilidade em diferentes plataformas e <i>browsers</i>	Teste de compatibilidade do SaaS, protocolos de conectividade e interfaces com os usuários	Testes de interoperabilidade, compatibilidade de plataformas e <i>browsers</i> sob o ponto de vista dos usuários	Teste de compatibilidade da aplicação, interoperabilidade ponta a ponta e conectividade da aplicação com sistemas legados de outras nuvens
Teste de regressão	Mudanças e impactos em funcionalidades fornecidas pelo SaaS/nuvem e conectividade das APIs	Teste de regressão dirigida ao SaaS/nuvem	Revalidação sob a perspectivas dos usuários	Teste de regressão em nível de sistema

Fonte: Adaptada de Gao, Bai e Tsai (2011).

Um ambiente satisfatório para teste de sistemas SaaS requer uma infraestrutura de nuvem (ou plataforma) para apoiar completamente a atividade de teste, com diversos recursos escaláveis, infraestruturas e ferramentas, de modo que possam ser alocadas dinamicamente considerando requisições estáticas e dinâmicas. Existem duas opções possíveis: simulação do ambiente de teste na nuvem (*in-house*) ou por meio de um provedor de serviços na nuvem. Além de identificar os tipos de teste aplicáveis, a equipe de testes também deve se concentrar nos requisitos específicos da aplicação a serem testados por estarem em um ambiente de nuvem, tais como: questões relacionadas ao gerenciamento de sessões de usuários, vulnerabilidades de segurança, no caso

da multi-tenancy - restringir o acesso dos usuários a somente seus dados, além da experiência dos testadores (MAHMOOD; SAEED, 2013). Além disso, o teste de sistemas SaaS possui particularidades relacionadas com interesses de qualidade, requisitos e propriedades específicas. Dentre os aspectos mais importantes, pode-se destacar: acordo em nível de serviço (SLA); modelo de cobrança e faturamento de serviços; e simulação de tráfego e manipulação de dados em larga escala (GAO; BAI; TSAI, 2011).

Com relação ao SLA, os sistemas SaaS fornecem diversos tipos de serviços para seus usuários finais a partir de um acordo bem definido. Naturalmente, esse acordo torna-se parte do teste, no sentido de garantir que requisitos de qualidade especificados sejam atendidos. Quanto aos modelos de cobrança e faturamento, recursos computacionais e infraestruturas, ferramentas, e tarefas de teste oferecidos como serviço podem ser alterados com base nos modelos de custos pré-definidos, ou seja, o engenheiro pode migrar para um novo ambiente de teste se o custo for mais atrativo. Por fim, aplicar e simular o tráfego e acesso de usuários em larga escala em interfaces de conectividade é importante para o teste, particularmente no teste de desempenho e validação de funcionalidades.

Existem poucos trabalhos que apresentam a diversidade das questões envolvidas no teste de sistemas SaaS. Nesse sentido, Gao *et al.* (2013) realizaram uma revisão da literatura para reunir as necessidades que o teste de sistemas SaaS precisa satisfazer. Com isso, apontaram um processo de teste contendo seis etapas:

Etapa #1: Teste unitário de componentes requer a condução de testes unitários caixa-preta e caixa-branca para assegurar a qualidade dos componentes em um sistema SaaS;

Etapa #2: Teste de integração de componentes permite que engenheiros realizem a integração de componentes de serviços subjacentes ao sistema SaaS para assegurar que os componentes podem operar juntos adequadamente;

Etapa #3: Testes funcionais com base nos tenants requer validação das funcionalidades do sistema SaaS para cada *tenant*, assegurando seu comportamento correto, fluxos de operação e resultados esperados;

Etapa #4: Testes de sistema com base nos tenants exige a verificação do sistema em termos de requisitos de qualidade dos serviços, para garantir a confiabilidade em nível de sistema, desempenho, disponibilidade, segurança e assim por diante;

Etapa #5: Teste de features com base nos tenants requer a validação de características específicas do sistema SaaS, incluindo *multi-tenancy*, customização, composição e integração. Além disso, verificação de faturamento, monitoração e medição da utilização devem ser verificadas.

Etapa #6: Teste contínuo com base nos tenants verifica dinamicamente o sistema SaaS para garantir sua qualidade após evoluções. **Teste de recuperação de falhas e desastres** visa

assegurar que um sistema SaaS pode ser recuperado adequadamente quando ocorrem falhas no sistema e desastres especificados previamente. **Teste de serviços de atualização em tempo real** verifica se um SaaS continua operando adequadamente após alterações para novos *tenants*. **Teste de comportamentos baseando-se em estados** destina-se a validação de comportamentos dinâmicos para um sistema SaaS durante sua execução. Além desses testes, pode-se destacar, **avaliações referentes a qualidade dos serviços** durante o tempo de execução de um sistema SaaS para verificar questões como desempenho, confiabilidade, disponibilidade, escalabilidade e teste de carga do sistema. Essa última etapa é a mais importante, pois lida com as mudanças do sistema SaaS que também é conhecida como teste de regressão em tempo real.

Embora esse processo descreva em linhas gerais as preocupações do teste de sistemas SaaS, [Gao et al. \(2013\)](#) não apresentam uma aplicação prática desse processo ou discutem como as etapas poderiam ser aplicadas. A ideia desse processo restringe-se em indicar em alto nível de abstração, quais aspectos precisam ser verificados progressivamente durante os testes. Em geral, os testes na nuvem exemplificam os testes sob demanda e podem ser considerados como o futuro dos testes de serviços. No entanto, novas necessidades de negócio e desafios associados devem ser adequadamente compreendidos antes de migrar o teste para o ambiente da nuvem. Além disso, as organizações precisam de infraestrutura adicional e diferentes habilidades ([MAHMOOD; SAEED, 2013](#) apud [GHAG, 2012](#)).

4.3 Mapeamento Sistemático da Literatura

Um Mapeamento Sistemático da Literatura (MSL) é um método adequado para mapear um determinado tópico quando existem poucas evidências ou o tópico de pesquisa é amplo e fragmentado. Desse modo, um MSL foi realizado ([PINTO et al., 2016](#)) sobre multi-tenancy no contexto de sistemas SaaS seguindo as diretrizes propostas por [Kitchenham \(2004\)](#). Oitenta e nove estudos primários foram selecionados para responder duas questões de pesquisa sob a perspectiva da academia. A análise dos resultados tiveram enfoque na apresentação das frequências das publicações para diferentes categorias de pesquisa. Como principal contribuição, uma definição sobre os principais desafios foi apresentada com o intuito de apoiar o direcionamento de novas pesquisas no campo da arquitetura multi-tenant. O MSL foi conduzido considerando três fases principais: (i) planejamento, (ii) execução e (iii) discussão dos resultados.

4.3.1 Planejamento

Nessa fase, o protocolo de revisão foi definido contendo: (i) questões de pesquisa, (ii) estratégia de busca, (iii) critérios de inclusão e exclusão e (iv) processo de extração de dados e metodologia para sintetizar os dados coletados. O principal objetivo foi obter um panorama das

dificuldades relacionadas com a arquitetura multi-tenant, alternativas propostas na literatura e oportunidades de pesquisas. Portanto, duas questões de pesquisa (QP) foram definidas:

QP₁: Quais os tópicos recentes de pesquisa relacionados à AMT podem ser encontrados na literatura?

QP₂: Quais são os principais desafios de pesquisa e oportunidades relacionadas ao desenvolvimento, teste e evolução de sistemas SaaS multi-tenants?

Uma string de busca e as bases eletrônicas também foram definidas. A string de busca foi elaborada e refinada de acordo com um conjunto inicial de artigos selecionados e baseados nas citações desses trabalhos. Durante a validação da string tais artigos deveriam ser retornados a partir das bases eletrônicas (Tabela 2). Embora esse processo seja subjetivo, esse controle permite a calibração da string e identificação de possíveis estudos relevantes.

Tabela 2 – Lista de artigos chave usados para calibrar a string de busca.

Autores	Ref.
Seungseok et al.	(KANG <i>et al.</i> , 2010)
Sengupta e Roychoudhury	(SENGUPTA; ROYCHOUDHURY, 2011)
Tsai et al.	(TSAI <i>et al.</i> , 2013)
Ru et al.	(RU; GRUNDY; KEUNG, 2014)

As seguintes palavras-chave foram consideradas na definição da string: cloud, SaaS e multi-tenancy, suas variações frequentes e operações booleanas. Na Figura 16 descreve-se a string de busca elaborada. As bases científicas consideradas foram: *ACM*, *IEEE*, *Scopus* e *Wiley Online*. Essas bases incluem as principais conferências e periódicos envolvendo computação em nuvem.

(cloud **and** (SaaS **or** "Software as a Service") **and** (multi-tenancy **or** multi-tenant **or** tenancy **or** tenant **or** tenants))

Figura 16 – String de busca.

Estudos primários relevantes foram selecionados com base nos seguintes critérios de inclusão (*CI*) e de exclusão (*CE*). Nem todos os critérios de inclusão devem ser satisfeitos para cada estudo primário; *CI_a* é o único critério obrigatório para a inclusão de artigos.

CI_a: O estudo primário apresenta pelo menos um desafio ou oportunidade de pesquisa no contexto de AMT;

CI_b: O estudo primário apresenta pelo menos uma ferramenta, framework, processo ou APIs para o contexto de AMT;

CI_c : O estudo primário atende pelo menos uma dificuldade envolvendo a AMT em termos de utilização ou migração;

CI_d : O estudo primário apresenta pelo menos uma propriedade, classificação ou avaliação de uma solução considerando a AMT;

CE_a : O estudo apresenta um desafio ou uma oportunidade de pesquisa no contexto de AMT. Entretanto, trata-se de um *short paper*;

CE_b : O estudo é uma Revisão Sistemática da Literatura;

CE_c : O artigo completo está indisponível.

Um formulário para extração dos dados foi definido para responder as questões, conforme apresentado na Tabela 3. Algumas categorias foram incluídas (Item 5) para classificar o principal domínio de cada estudo primário, por exemplo, “*Customization*” e “*Database*” são categorias pré-definidas.

Tabela 3 – Elementos do formulário de extração de dados.

Atributos	
<i>Metadatos</i>	ID, revisor e data.
<i>Elementos</i>	Título, ano, fonte (i.e. conferência ou journal) e base eletrônica.
<i>Dados a serem extraídos</i>	1) Desafios/oportunidades; 2) Ferramentas, frameworks e APIs; 3) Dificuldades na utilização da AMT e adoção/migração; 4) Especificações, classificação e avaliação de soluções para AMT e 5) Categoria do artigo.

Durante o processo de extração, os dados dos estudos primários foram coletados por três revisores, estudantes de doutorado em Ciência da Computação. Os dados foram extraídos por um revisor e verificados por outro. Esse MSL foi realizado entre maio e agosto de 2015 e os dados foram documentados e estão disponíveis¹.

4.3.2 Execução

Nesta fase os estudos primários foram identificados nas bases eletrônicas. *Scopus* retornou o maior número de trabalhos (638). *IEEE*, *ACM*, *Wiley* retornaram 168, 594 e 25, respectivamente. Na Figura 17 é apresentada a distribuição de trabalhos recuperados em cada base e após aplicar os critérios de inclusão e exclusão no processo de leitura. A partir desse conjunto inicial, 135 eram duplicados e por isso foram removidos. Na fase de seleção, com base na leitura parcial (títulos e resumos), um conjunto de 149 artigos foram selecionados com a aplicação dos critérios, posteriormente a leitura completa foi realizada e apenas 89 artigos foram selecionados. Procurou-se manter o processo alinhado às questões de pesquisa mas na tentativa de evitar que

¹ <https://goo.gl/0K68jp>

trabalhos relevantes não fossem retornados, a string de busca se tornou genérica para recuperar muitos estudos das bases, mesmo que isso trouxesse mais esforço no processo de seleção. Muitos trabalhos foram introduzidos como estudos primários, mas apenas alguns deles tinham contribuições relevantes ou maiores impactos considerando o escopo das investigações.

Figura 17 – Distribuição de artigos (fase de execução).

Para validar a aplicação dos critérios de inclusão e exclusão, cada estudo primário foi pontuado pelos revisores em ambas fases de leitura. Uma nota “0” significa exclusão e “1”, o aceite. Em casos de dúvida, o revisor pontuava com “0.5” e os demais revisores eram questionados sobre sua relevância, de modo que um consenso deveria ser alcançado para decidir pelo aceite ou exclusão. O processo de pontuação foi conduzido de forma sequencial e independente, ou seja, um revisor realizava a leitura e pontuava cada artigo sem a interferência dos demais. Na leitura parcial, todos os artigos foram pontuados e posteriormente, na leitura completa eles eram classificados pelos revisores até alcançar um conjunto de estudos relevantes para responder às questões de pesquisa.

4.3.3 Discussão dos resultados

QP₁ - Quais os tópicos recentes de pesquisa relacionados à AMT podem ser encontrados na literatura?

Para tornar mais claro o enfoque dos estudos selecionados em termos quantitativos, algumas categorias foram definidas de acordo com o domínio do artigo. Na Figura 18 apresenta-se um mapeamento contendo o número de estudos primários distribuídos de acordo com o ano de publicação e a categoria, na qual cada artigo pode ser classificado em uma ou mais categorias. Segurança, atividade de teste e experimentos podem ser considerados como questões importantes para a garantia da qualidade e portanto, tais questões estão na categoria de garantia de qualidade.

Na Figura 19 apresenta-se a disposição dos artigos selecionados a partir de workshops, journals e conferências. As conferências apresentaram-se em uma posição dominante com 63 artigos (70.8%), seguidas por journals com 18 trabalhos (20.2%) e workshops, 8 artigos (9%). Isso pode indicar que workshops ainda precisam ser formadas e os pesquisadores submetem seus

Figura 18 – Distribuição dos estudos primários de acordo com as categorias.

resultados para conferências e periódicos com um escopo maior. Nas próximas seções discute-se a ideia principal dos estudos selecionados organizados pelas categorias mostradas na Figura 18.

Figura 19 – Quantidade de artigos selecionados a partir de workshops, journals e conferências.

4.3.3.0.1 Desenvolvimento

Arquiteturas, frameworks, requisitos e gerenciamento de variabilidade, migração de aplicações web convencionais para sistemas SaaS multi-tenants são mencionados em trabalhos recentes como questões importantes para serem investigadas em pesquisas futuras. Devido a complexidade do gerenciamento e custos com manutenção, fornecedores de SaaS comumente desenvolvem uma única versão da aplicação para todos os tenants. [Truyen et al. \(2012\)](#) avaliaram a programação orientada ao contexto (*context-oriented programming*) para tais aplicações visando melhorar seu processo de desenvolvimento. A ideia principal foi permitir a customização de requisitos específicos para cada tenant, fornecendo resultados positivos em termos de esforço de desenvolvimento.

Gerenciamento de requisitos é um dos desafios relacionados ao fornecimento de aplicações satisfatórias para múltiplas organizações. [Walraven et al. \(2014\)](#) apresentaram uma alternativa baseada em linha de produtos de software e configurações específicas co-existent para os tenants com o intuito de facilitar o gerenciamento dos requisitos. Essa estratégia foi aplicada para alguns sistemas SaaS e como um resultado, os esforços para configurar e compor novas versões foram reduzidos. Aplicações SaaS multi-tenants visam fornecer diferentes

configurações para atender as demandas de seus tenants em termos de requisitos funcionais e não funcionais. Entretanto, é natural que diferentes requisitos previamente fornecidos sejam necessários por novos tenants. De um lado, novas features podem ser desenvolvidas e incluídas na aplicação, de modo que instâncias existentes continuem executando sem defeitos. De outro lado, custos relacionados aos esforços com teste de regressão precisam ser considerados durante o desenvolvimento de novas features (SENGUPTA; ROYCHOUDHURY, 2011).

Kale e Borhade (2013) forneceram um framework que cobre features como portabilidade, customização, segurança e escalabilidade visando reduzir o tempo gasto com o desenvolvimento de software. Nessa mesma linha, Nam e Yeom (2014) propuseram um framework para apoiar serviços diferentes para um amplo conjunto de tenants. De acordo com os autores, embora muitos benefícios podem ser alcançados com a AMT, poucos processos podem apoiar o desenvolvimento de aplicações considerando todas as implicações dessa arquitetura. Seguindo nessa perspectiva, Manduca *et al.* (2014) descrevem uma abordagem para o desenvolvimento de sistemas SaaS multi-tenants com uma única base de dados partindo de aplicações web convencionais. Ela emprega componentes arquiteturais e *design patterns* e mantém as funcionalidades na linguagem de programação corrente. Como uma dificuldade, os autores relatam que nem sempre a documentação da plataforma é suficiente para desenvolver esse tipo de aplicação, fator que gera uma dificuldade adicional nesse processo de migração.

Na Tabela 4 descrevem-se as principais preocupações relacionadas com o desenvolvimento de sistemas SaaS multi-tenants.

Tabela 4 – Principais questões relacionadas com o desenvolvimento de sistemas SaaS multi-tenants.

	Descrição	Ref.
Requisitos	Composição de versões das aplicações multi-tenants seguindo a engenharia de linha de produtos	(WALRAVEN <i>et al.</i> , 2014)
Implementação	<i>Context-oriented programming</i>	(TRUYEN <i>et al.</i> , 2012)
	Diretrizes para implementar novas versões da aplicação para os tenants sem impactar as previamente disponibilizadas	(SENGUPTA; ROYCHOUDHURY, 2011)
Frameworks	Processo para apoiar a portabilidade, customização, segurança e escalabilidade	(KALE; BORHADE, 2013)
	Framework para atender disponibilidade, extensibilidade e escalabilidade em aplicações multi-tenants	(MORAKOS; MELIONES, 2014)
Migração	Processo para apoiar a migração de aplicações web convencionais para sistemas SaaS com esforço relativamente baixo	(MANDUCA <i>et al.</i> , 2014)

4.3.3.0.2 Desempenho

A AMT introduziu novos desafios relacionados ao balanceamento de carga e alocação de recursos, incluindo as requisições em nível de tenant, acordos em nível de serviços, objetivos de

desempenho e qualidade dos serviços.

Sun *et al.* (2014) apontaram uma política para melhoria no balanceamento de carga para o ambiente da multi-tenancy. Em nível de persistência, Moon *et al.* (2013) apresentaram um balanceador de carga para bancos de dados que aumenta o desempenho e melhora o compartilhamento de recursos entre os tenants. Patikirikoralu *et al.* (2012) desenvolveram uma abordagem que utiliza um controle não linear reabastecido (*nonlinear replenished control*) para manter o desempenho em níveis de uso distintos para diferentes tenants, dependendo de suas prioridades. Ele permite a detecção de sobrecarga e com isso, o controle das operações dos tenants pode ser alterado dinamicamente.

Krebs, Wert e Kounev (2013) estenderam um benchmark web chamado TPC-W para incluir multi-tenancy e a possibilidade de comparar duas perspectivas: (i) multi-tenancy e (ii) virtualização. Multi-tenancy se demonstrou mais eficiente do que a virtualização considerando a taxa de transferência, número de tenants e quando a memória era um gargalo (*bottleneck*).

4.3.3.0.3 Customização

Um número considerável de artigos envolvendo o domínio de customização foi retornado das bases eletrônicas. Uma aplicação SaaS multi-tenant que atende um amplo conjunto de tenants deve tornar possível uma série de customizações (TSAI; SUN, 2013). A customização de uma aplicação complexa é uma tarefa propensa a erros, requer esforços manuais e os usuários podem não conhecer a melhor escolha. Assim, os autores realizaram um estudo sobre as possibilidades disponíveis para a customização de um sistema SaaS, e um processo semi-automático foi elaborado visando a redução de custos.

Ramachandran, Narendra e Ponnalagu (2012) observaram que a customização pode resultar em alto custo devido aos reajustes que podem ser necessários após as evoluções. Para sistemas SaaS, é necessário a configuração de instâncias específicas e gerenciamento de recursos alocados para os tenants. Para Walraven, Truyen e Joosen (2011) a customização envolvendo variações no núcleo de uma aplicação é muito cara para os provedores de aplicações na nuvem e introduzem uma complexidade adicional a esse tipo de desenvolvimento.

4.3.3.0.4 Garantia de Qualidade

A garantia de qualidade é um tópico promissor no campo de AMT (ALKHATIB *et al.*, 2014) que inclui estratégias de teste, métricas e critérios de qualidade. Para Tsai *et al.* (2013), um dos principais desafios na atividade de teste de sistemas SaaS multi-tenants é lidar com um amplo conjunto de possibilidades de composição e interações entre os componentes. Os autores forneceram uma abordagem de teste combinatório para gerar sequências de teste dinâmicas e alcançar uma alta cobertura arquitetural. A ideia principal foi identificar as composições com maiores chances de defeitos por meio de um algoritmo. Quando um novo componente é composto

e uma determinada versão torna-se disponível, o algoritmo revela defeitos nas interações entre os componentes.

A complexidade do modelo de computação em nuvem e a falta de padronização tornam a segurança uma questão crítica para provedores e clientes. De acordo com Wood e Anderson (2011), multi-tenancy tem um impacto direto no desenvolvimento de aplicações e na forma como elas são fornecidas. Nesse sentido, Almorsy, Grundy e Ibrahim (2011) elaboraram um framework que melhora a colaboração entre provedores de serviços e consumidores e gerencia a segurança de plataformas na nuvem, bem como seus serviços hospedados.

Na Tabela 5 apresentam-se as principais questões investigadas em relação à garantia de qualidade no contexto da multi-tenancy.

Tabela 5 – Principais questões identificadas em relação à garantia de qualidade.

	Descrição	Ref.
Teste combi- natório	Sequências dinâmicas de teste usadas para alcançar alta cobertura arquitetural	(TSAI <i>et al.</i> , 2013)
Teste de re- gressão	Teste contínuo com o particionamento dos dados dos tenants e geração de casos de teste com base em metadados	(TSAI <i>et al.</i> , 2010)
Segurança	Framework para gerenciamento de segurança	(ALMORSY; GRUNDY; IBRAHIM, 2011)
	SecPlac: modelo de alocação de recursos para apoiar a segurança no compartilhamento de infraestrutura entre tenants	(SALEH <i>et al.</i> , 2014)
	TOSSMA (<i>Tenant Oriented SaaS Security Management Architecture</i>): uma arquitetura para isolamento de recursos para tenants por meio da injeção de controles de autorização	(ALMORSY; GRUNDY; IBRAHIM, 2012)
	Privacidade na combinação de dados	(ZHANG; LI; SHI, 2011)
QoS	MSSOptimiser (<i>Multi-tenant SaaS Optimizer</i>): uma abordagem para selecionar serviços atendendo requisitos de qualidade	(HE <i>et al.</i> , 2012)

4.3.3.0.5 Banco de dados

Dezenove artigos selecionados no MSL citaram questões relacionadas com banco de dados como uma direção promissora de pesquisa. Saraswathi e Bhuvaneswari (2014) apresentaram duas alternativas para arquitetura de dados multi-tenant: i) uma relacionada com a autenticação e autorização e ii) uma abordagem não intrusiva para aplicações em larga escala. Os autores descreveram um processo para aplicá-las e direcionar os engenheiros no desenvolvimento de estruturas para otimização para as bases de dados no contexto de multi-tenancy.

Em outra linha de pesquisa, Maenhaut *et al.* (2014) desenvolveram uma abordagem para gerenciamento de dados de forma hierárquica e levando em conta algumas métricas de

desempenho. A principal preocupação nesse estudo foi em relação à distribuição de usuários e dados em múltiplas instâncias do banco de dados. [Yaish e Goyal \(2013\)](#) discutem aspectos importantes sobre o modelo de controle de acessos em *schemas* utilizados em sistemas multi-tenants. Um algoritmo de controle de acesso que permite aos usuários acessarem os dados concedidos com base em grupos de usuários ou funções atribuídas, melhorando a efetividade na camada de persistência para esse modelo de aplicações.

QP₂ - Quais são os principais desafios de pesquisa e oportunidades relacionadas ao desenvolvimento, teste e evolução de sistemas SaaS multi-tenants?

Embora a maioria dos estudos selecionados sejam direcionados ao desenvolvimento de sistemas SaaS multi-tenants, padrões continuam dispersos e não seguem uma abordagem metodológica. Muitas soluções são proprietárias e raramente interoperáveis ([ALKHATIB et al., 2014](#)). Diversas questões devem ser consideradas durante o teste de sistemas SaaS multi-tenants: (i) recursos são compartilhados entre os tenants e seus usuários finais, (ii) cada versão da aplicação atende requisitos específicos para cada tenant, a qual é executada como se estivesse em ambiente dedicado e pode ser composto de vários componentes e (iii) uma versão é entregue aos clientes por meio de uma *engine* que em tempo de execução realiza a combinação entre dados de customização e metadados específicos que permitem diferentes interfaces e atendimento à regras de negócio específicas.

Segundo [Alkhatib et al. \(2014\)](#), a comunidade ainda não possui métricas de qualidade efetivas para SaaS e novas estratégias de teste são necessárias para atender os desafios impostos pelo modelo de computação em nuvem. Questões envolvendo o teste de integração, métodos de validação e padrões para garantia de qualidade atendendo as interfaces de interação devem ser estabelecidos. Dada a alta disponibilidade que é essencial para SaaS, as técnicas de re-teste considerando multi-tenancy são obrigatórias sempre que o software é alterado para melhorias ou correção de bugs.

Muitos estudos sobre testes no ambiente de nuvem são direcionados para verificação de requisitos não funcionais como desempenho e segurança. Considerando os estudos selecionados, foram identificados os seguintes campos de pesquisa: (i) extensões de critérios de teste convencionais, (ii) estratégias de teste para componentes suscetíveis de customizações, (iii) alternativas para verificar as interfaces de composição e impactos na introdução de novos componentes e (iv) abordagens para o teste de regressão que requerem mais cooperação entre a indústria e a academia. Com relação a evolução dos sistemas SaaS, a comunidade ainda não forneceu abordagens bem definidas. A atividade de evolução em sistemas distribuídos podem indicar diretrizes para lidar com as execuções isoladas das versões que compartilham uma única instância da aplicação.

4.3.4 Desafios do teste para sistemas SaaS multi-tenants

O teste de software considerando multi-tenancy pode ser mais complexo do que o experimentado durante o teste de software tradicional. [Mahmood e Saeed \(2013, p. 150\)](#) argumentam que o teste tradicional de software não pode ser aplicado para verificar aplicações no ambiente de nuvem pelo fato do mesmo ser projetado sob a premissa de aplicações single-tenant e devido a complexidade que reside no alto nível de customização, multi-tenancy e o ambiente dinâmico. Nesta seção descrevem-se os principais desafios para o teste de sistemas SaaS multi-tenants a partir do Mapeamento Sistemático da Literatura.

C1: Mudanças frequentes. Modelos de negócio mudam com frequência e as organizações precisam continuar atendendo seus clientes, portanto atualizações devem ser aplicadas com um certo grau de frequência em sistemas multi-tenants ([GAO; BAI; TSAI, 2011](#)). Algumas vezes, uma atualização realizada para um tenant pode ser um requisito para outro. Desse modo, todas as alterações devem ser cuidadosamente verificadas para que não afetem os recursos que estavam funcionando adequadamente para os tenants restantes. O planejamento de uma feature para um locatário específico deve considerar as possíveis implicações em outros tenants.

C2: Desempenho. Múltiplos tenants compartilham recursos do sistema como serviços, aplicações, banco de dados e hardware, desse modo é necessário garantir que todos os tenants utilizem adequadamente tais recursos. Uma aplicação precisa ser altamente escalável e deve operar segundo a capacidade dos recursos disponíveis. Nesse sentido, se um tenant requer mais recursos, o desempenho para os tenants restantes podem ser afetados.

C3: Segurança. Embora a multi-tenancy seja baseada em recursos compartilhados, é importante assegurar níveis satisfatórios de segurança, acessibilidade e preocupações em termos de privacidade. Aplicações multi-tenants devem garantir que os usuários possuem acesso restrito aos dados que lhes convém. As versões das aplicações precisam ser verificadas simulando usuários de diferentes tenants com privilégios variados. Prevenções contra *SQL injection* e vulnerabilidades com cookies são preocupações que devem ser incluídas como parte do teste de segurança de aplicações multi-tenants. A rápida adoção de novas features na aplicação para responder à novas demandas podem contribuir para brechas na segurança. A encriptação de dados deve ser completamente testado para garantir a segurança dos dados entre vários tenants. Muitas organizações fornecem testes funcionais, de desempenho e de unidade como um serviço na nuvem, no entanto, existem poucas abordagens propostas para testes de segurança neste contexto ([SURESHKUMAR; KANNAN; PURNIEMAA, 2013](#); [PINTO et al., 2016](#)).

C4: Isolamento. [Vashistha e Ahmed \(2012\)](#) destacam que o principal desafio no campo de teste de sistemas multi-tenants está relacionado com o isolamento. Como os tenants compartilham a mesma instância da aplicação, uma vez que uma customização seja realizada para um tenant específico, os serviços designados para todos os tenants podem ser afetados e possivelmente, interrompidos durante uma alteração. Portanto, o isolamento deve ser cuidadosamente

considerado nos testes para evitar efeitos colaterais, como a indisponibilidade de serviços para as aplicações projetadas para os demais tenants. Uma alternativa é a geração de uma versão separadamente e após seu isolamento, conduzir a atividade de teste sem tornar as demais indisponíveis. No entanto, é importante ao testar uma feature para um tenant específico, considerar as implicações em outros tenants (SALESFORCE, 2008).

A identificação dos tenants é essencial para qualquer aplicação multi-tenant e defeitos nesse mecanismo podem resultar em sérios efeitos colaterais. Uma funcionalidade conhecida como *multi-tenant interceptor* geralmente é desenvolvida para interceptar as requisições e capturar informações do HTTP *header*. O principal desafio para essa funcionalidade está em assegurar que todas as interceptações que levam às definições do tenant corrente não tenha sofrido interferências de outros tenants, ou seja, é importante verificar se as operações nas sessões entre diferentes tenants estão isoladas (PINTO; SOUZA; SOUZA, 2019a).

Operações assíncronas estão presentes em sistemas multi-tenants, assim como em aplicações web segundo o modelo *request-response*. Para programas concorrentes (JOHNSEN; OWE, 2007), isso se refere a execução de processo em threads isoladas enquanto para sistemas multi-tenants o gerenciamento de operações assíncronas está relacionada com as sessões cruzadas (ou *cross-sessions* e a necessidade de isolamento de sessões por tenant. A probabilidade de desvios nos fluxos de execução podem aumentar quando não há um gerenciamento adequado dessas sessões. Por exemplo, assumindo que um usuário já registrado em uma versão “A” está comprando um determinado produto e ao mesmo tempo, um visitante da versão “B” está concluindo seu registro, um defeito pode ser reportado se por alguma razão o usuário de “A” foi redirecionado para a versão projetada para “B”, i.e, não existem razões para mudar de aplicações no meio do fluxo de execução.

C5: Falta de padronização. Um ponto crítico para testar tais aplicação é que a comunidade ainda não possui métricas de qualidade efetivas para sistemas multi-tenants e novas estratégias de teste são necessárias para atender os desafios impostos pelo modelo de computação em nuvem (ALKHATIB *et al.*, 2014). Questões no teste de integração de software, métodos de validação e padrões de garantia de qualidade atendendo as interfaces de interação devem ser estabelecidas. A alta disponibilidade é essencial para SaaS, técnicas para (re) execuções de teste considerando as particularidades da multi-tenancy são obrigatórias sempre que o software é alterado para melhorias. De acordo com Tsai *et al.* (2013), um dos principais desafios no teste de sistemas SaaS é lidar com a alta capacidade de customização e interações entre seus componentes. Para Priyadharshini e Malathi (2014), uma arquitetura de referência para o desenvolvimento e novas estratégias de testes são importantes para contornar esse tipo de desafio e superar as desconfiças da indústria.

4.4 Trabalhos relacionados

A arquitetura multi-tenant é provavelmente o elemento mais importante para um SaaS que admite as vantagens competitivas que a nuvem pode trazer para os negócios. Embora estudos que realizam a correlação entre testes automatizados e aplicações desenvolvidas para nuvem sejam escassos, existem contribuições interessantes como verificação de tais aplicações por meio de modelos (*model checking*), teste de performance usando algoritmos genéticos e execução simbólica, estratégias para testes de regressão, geração de casos de teste para cobrir múltiplas versões e abordagens para apoiar a definição de casos de teste mais efetivos para sistemas baseados em nuvem e *web services*.

Segundo [Hu, Lei e Tsai \(2016\)](#) métodos formais e técnicas de verificação podem ser aplicadas para verificar arquiteturas para aplicações na nuvem, uma vez que ferramentas de verificação e técnicas podem ser remodeladas. Os pesquisadores apresentam um serviço de verificação sob demanda chamado *Multi-tenant VaaS (Verification as a Service)* que fornece um modelo formal matemático e com o apoio de uma ferramenta de *model checking* é possível verificar a correspondência do modelo matemático e os requisitos correspondentes de um sistema sob teste. Além disso, um estudo de caso foi realizado para avaliar a abordagem proposta e ao aplicar técnicas de transformação, combinações e análises de modelo em diferentes fases para verificar o software, tarefas de verificação podem ser realizadas de forma paralela e distribuída. A verificação formal é uma das práticas mais efetivas para o teste de software, os resultados mostraram que a renderização da execução sequencial pode se tornar inviável devido ao consumo de tempo significativo com o carregamento do sistema e agendamento de tarefas. Nesse sentido, a execução paralela e distribuída pode ser aplicada para reduzir o tempo gasto com esse tipo de verificação. Nos estudos preliminares envolvendo a técnica funcional, a execução paralela dos testes tiveram o objetivo de exercitar várias aplicações simultaneamente, forçando possíveis desvios no roteamento de cada tenant (*tenant view-routing*) para identificar falhas com natureza concorrente.

Desempenho é uma das principais motivações para as organizações que pretendem adotar a arquitetura multi-tenant. [Sumalatha e Parthiban \(2018\)](#) realizaram uma avaliação do teste de desempenho de múltiplos tenants. Os resultados coletados são usados para função de *fitness*. Um algoritmo conhecido como GASE é usado combinando um algoritmo genético e execução simbólica para obter o melhor de cada tenant. Em relação as gerações *fitness*, o algoritmo genético é definido como um método de busca aleatória que tem maior capacidade de otimização e paralelismo. Valores simbólicos são usados para representar os valores das variáveis do programa como expressões simbólicas. Para avaliar a abordagem, um benchmark foi adotado onde foi calculado o rendimento máximo de uma plataforma de suporte a multi-tenancy com base na quantidade de tenants e usuários por tenant.

[Makki, Landuyt e Joosen \(2016\)](#) discutem os desafios no teste de regressão para aplicações multi-tenant. Um framework de teste e um conjunto de técnicas de automação de teste

configuráveis foram fornecidas para detecção de evoluções sob o controle de “administradores dos tenants”, ou seja, os responsáveis em aplicar alterações nas versões das aplicações para alinhá-las às necessidades das organizações. Segundo os autores, essa solução proposta é capaz de evitar efeitos colaterais indesejáveis de testes no ambiente de produção e diminuir o tempo de execução do teste. O framework de testes aproveita a execução de fluxos de trabalho anteriores para gerar automaticamente novos casos de teste e ajudar a execução de testes on-line por esses administradores. Embora os autores ressaltem a preocupação com efeitos colaterais indesejáveis, a solução proposta não cobre a perspectiva dos usuários de projetar os dados do teste. Com base nos estudos preliminares realizados durante este projeto (apresentados no Capítulo 5), acredita-se que os testes automatizados e paralelos são capazes de aumentar a eficácia dos testes de regressão. Usando ferramentas automatizadas como o *Selenium*, o teste de regressão pode ser realizado para monitorar a execução das funcionalidades e validar a aplicação sob o impacto das atualizações.

[Tsai et al. \(2013\)](#) apresentaram um projeto de um TaaS (Testing as a Service) para teste combinatório de sistemas SaaS utilizando álgebra e um algoritmo de raciocínio adaptativo (AR). Essa abordagem baseada em teste adaptativo permite a identificação de combinações defeituosas improváveis de serem selecionadas pelos desenvolvedores. O algoritmo AR aplica os resultados de testes anteriores para gerar novos casos de teste para encontrar defeitos em versões designadas aos tenants levando em consideração tais princípios: i) quando uma versão (ou configuração) falha em um dado cenário de teste, há pelo menos uma falha (talvez mais) na configuração do tenant; e ii) quando uma versão passa em um teste, nenhum defeito foi revelado com os casos de teste que foram definidos para uma determinada configuração para o tenant. Em geral, o teste de interação combinatória é uma estratégia para amostrar um conjunto de casos de teste a partir de um amplo espaço de busca, portanto, essa estratégia pode ser uma alternativa viável para isolar as combinações defeituosas. Durante os estudos preliminares envolvendo os testes automatizados, notou-se certa dificuldade em implementar os scripts de teste com as combinações de casos de teste. Para verificar um determinado elemento, os demais exigiam outros valores pois somente dessa forma os dados poderiam ser enviados nas requisições. Assim, identificar quais combinações são mais prováveis de revelar defeitos sem uma abordagem combinatória é uma tarefa complexa.

[Sengupta e Roychoudhury \(2011\)](#) descrevem um conjunto de questões de pesquisa no projeto, teste e manutenção de sistemas SaaS multi-tenants. Com relação ao nível de variabilidade do tenant, os seguintes problemas são destacados: i) como a atividade de teste pode ser conduzida para que os tenants existentes não sejam afetados pela inserção de um novo tenant? e ii) como garantir a eficácia do teste para que o sistema SaaS atenda as necessidades dos diferentes tenants que foram inseridos? O primeiro problema leva em conta a geração de casos de teste, particularmente os casos de teste que não existem no conjunto de testes atual, mas que devem ser testados assim que os novos tenants forem incluídos. Para a segunda questão, uma estratégia de teste é proposta para explorar a similaridade entre os tenants e ajudar a projetar um conjunto de testes adequado. Neste trabalho descreve-se também a noção de uma árvore de teste (*Test-tree*)

para tais aplicações. O nó raiz captura o conjunto de teste que precisa ser utilizado para todos os tenants. Além disso, são propostas anotações para identificação das características genéricas e específicas de cada tenant. Uma árvore de teste pode ser aplicada com o objetivo de definir casos de teste direcionados a certas arestas e nós. Por exemplo, estruturas responsáveis pela definição do identificador dos tenants e todos os usos subsequentes dessa variável poderiam receber prioridade nos testes e possíveis falhas relacionadas à execução não determinística poderiam ser identificadas.

4.5 Comparação entre programas concorrentes e sistemas multi-tenants

Software concorrente oferece certas dificuldade para a atividade de teste devido a forma que as tarefas são definidas e executadas. Alguns conceitos no campo de multi-tenancy estão ligeiramente relacionados com programas concorrentes, sistemas distribuídos e virtualização (RU; KEUNG, 2013). Assim, é provável que muitos defeitos similares podem ser identificados em aplicações desenvolvidas para nuvem. Esta seção descreve conceitos sobre programação concorrente, apresentando uma classificação de defeitos para esses programas.

O comportamento não determinístico de programas concorrentes aumenta a dificuldade da atividade de teste, uma vez que para uma mesma entrada, diferentes execuções podem levar a diferentes caminhos do programa e podem produzir diferentes comportamentos. Assim, todos os caminhos possíveis devem ser verificados para garantir a qualidade. Essa questão também reside em aplicações multi-tenants com algumas distinções. Por exemplo, no modelo de comunicação *master-slave* usado para programas concorrentes (GAMMA, 1995), a execução é distribuída dinamicamente em diversos processos subordinados (*slaves*). Sincronização e comunicação devem ser gerenciadas para evitar comportamentos incorretos. Em relação às aplicações multi-tenants, uma única instância recebe múltiplas requisições de diferentes tenants e deve respondê-las como se fosse uma única aplicação isolada para um determinado usuário final pertencente a um tenant. Isso indica que o esforço no teste deve apoiar a identificação de defeitos considerando operações assíncronas para cada tenant. As versões específicas de cada tenant são virtualmente isoladas, entretanto, existem features genéricas que são compartilhadas e que podem receber as requisições e direcionarem os fluxos de execução. Defeitos em tais features impactam todas as versões e podem permitir o acesso a dados sensíveis de várias organizações.

Verificar todos os caminhos possíveis de um programa concorrente, embora seja o ideal, não é factível para a maioria dos programas. Isso deve-se ao fato de que um número expressivo de caminhos podem ser gerados para um programa. Geralmente, o teste de programas concorrentes emprega técnicas para gerar casos de teste para cobrir possíveis caminhos de execução. Por exemplo, a técnica de teste estrutural pode ser estendida em um modelo de abstração levando em conta as particularidades de programas concorrentes, como é o caso do *PCFG* (*Parallel Control*

Flow Graph, ou Grafo de Fluxo de Controle Paralelo) que tem como princípio o uso do Grafo de Fluxo de Controle (GFC) para cada processo paralelo. Assim, threads e processos podem ser mapeados em um grafo direcionado permitindo a verificação das comunicações e sincronizações entre eles (SOUZA; SOUZA; ZALUSKA, 2014).

Os processos ou threads em programas concorrentes podem apresentar diferentes classes de defeitos. Dois erros comuns nessas aplicações são: erro de observabilidade e *locking error*. O primeiro está associado como o ambiente de teste e pode ocorrer quando o testador ou a ferramenta de teste não controla o comportamento do programa ao executar de forma paralela. O erro de observabilidade pode ocorrer quando um processo depende da sincronização com outro processo para continuar sua execução. Assim, se o modelo utilizado não fornecer mecanismos para bloquear o processo até que o sincronismo seja executado, este programa pode não assegurar um resultado correto em todas as execuções, pois nesse caso, o tempo gasto por cada processo determinará se a sincronização será executada corretamente. O *locking error* pode ocorrer quando um processo “A” não se comunica de forma adequada com um processo “B” e como uma consequência, o processo B irá aguardar por uma sincronização ou até que um tempo limite seja excedido, desde que a mensagem esperada não tenha sido recebida.

Wu e Kaiser (2011) propuseram uma classificação de defeitos típicos para programas concorrentes, como descrito na Tabela 6.

Tabela 6 – Classificação de defeitos para programas concorrentes.

Defeitos	Descrição
<i>Data Race</i>	Múltiplas threads executam operações de gravação e leitura no mesmo arquivo, e a saída da execução depende exclusivamente da ordem na qual o arquivo é acessado.
<i>Memory Inconsistency</i>	Diferentes threads possuem visões inconsistentes de uma variável comum. Por exemplo, dada uma variável “var” e duas threads “t1” e “t2”, o valor armazenado em var é considerado diferente entre t1 e t2.
<i>Atomicity Violation</i>	Execução simultânea de múltiplas threads que transgridem a atomicidade de uma determinada região do código, ou seja, dada uma parte do código que deve ser executada sem interrupção, por algum motivo é interrompida e por exemplo, algum cálculo pode ser perdido durante tal ocorrência.
<i>Deadlock</i>	Múltiplas threads são bloqueadas para sempre. Por exemplo, dadas três threads “t1”, “t2” e “t3”, a qualquer momento t1 pode ser bloqueada em espera por uma sincronização com t2, então t2 pode ser bloqueada antes de sincronizar com t1, esperando por uma sincronização com t3, nesse momento se t3 é bloqueada, esperando por uma sincronização com t1, ambas threads serão bloqueadas para sempre.
<i>Livelock</i>	Uma determinada ação de um encadeamento pode ser executada em resposta à ação de outro encadeamento, portanto, se a ação de um encadeamento “t2” ocorrer em resposta a “t1”, enquanto um ciclo de ações e reações é iniciado ambos os tópicos estarão executando um <i>Livelock</i> . No entanto, ao contrário do <i>deadlock</i> , as threads não serão bloqueadas.
<i>Starvation</i>	Uma thread é bloqueada, incapaz de acessar recursos compartilhados e nunca é escalonada para acessar regiões compartilhadas, por isso não pode progredir em suas atividades
<i>Suspension</i>	Uma thread suspende suas atividades ou espera indeterminadamente.

Esses defeitos inspiraram a proposição de uma classificação de defeitos para sistemas SaaS, descritos na próxima seção.

4.6 Defeitos em sistemas SaaS multi-tenants

Um requisito para a classificação de classes de defeitos em um dado contexto de aplicações é investigar suas características e identificar o impacto das mesmas para a atividade de teste. No caso de sistemas SaaS multi-tenants esse processo não deve negligenciar defeitos que afetam outros modelos relacionados, uma vez que eles podem estar presentes também em aplicações desenvolvidas para a nuvem. Com esse princípio, duas questões principais podem ser formuladas. Primeiro, quais são as atividades essenciais de teste para atender as particularidades desse tipo de aplicação e como essas atividades podem ser conduzidas? Segundo, quais são os defeitos possíveis ao considerar as características de um sistema SaaS multi-tenant? Embora as respostas para ambos questionamentos sejam relevantes e estejam intimamente ligadas, o enfoque foi dado ao segundo questionamento pois precisava-se estabelecer uma visão geral dos defeitos para posteriormente definir quais defeitos seriam tratados no âmbito das pesquisas do doutorado.

Uma classificação de defeitos potenciais nesta área não havia sido encontrada na literatura e devido as diferentes particularidades desse tipo de sistemas, uma classificação genérica de defeitos não se aplica. Com base nessa oportunidade, uma taxonomia preliminar de defeitos para sistemas SaaS multi-tenants foi proposta neste projeto (PINTO; SOUZA; SOUZA, 2019a). Para isso, conceitos e elementos fundamentais desses sistemas e defeitos específicos foram identificados por meio do Mapeamento Sistemático da Literatura, comentado anteriormente. Além disso, um Survey foi realizado com a indústria para obter um panorama sobre as limitações das ferramentas disponíveis para o teste desse modelo de aplicações e seus possíveis defeitos sob a perspectiva de profissionais da área. Um conjunto de aplicações *open-source* disponíveis no gitHub também foi analisado para evidenciar seus principais mecanismos, incluindo o estudo da documentação técnica de frameworks e de práticas de desenvolvimento. O desenvolvimento do *MtShop* também contribuiu para constituir base técnica necessária para apontar as principais características de tais aplicações e seus principais defeitos.

Na Tabela 7 descreve-se uma sumarização das classes de defeitos, incluindo em qual nível o teste pode ser realizado, natureza das técnicas aplicáveis e algumas atividade de teste candidatas. A taxonomia compreende 10 características (A-J) e mais de 29 classes de defeitos (*f1-f29*, incluindo as sub classes). Essa classificação concentra-se nas classes mais importantes de defeitos em diferentes estágios do ciclo de vida de uma aplicação desenvolvida para nuvem.

A coluna chamada “Testing level” indica o nível de teste que o desenvolvedor/profissional de teste deve concentrar os esforços para evitar um defeito específico. Para cada classe de defeito, níveis como teste unitário, teste de integração, teste de sistemas e de aceitação foram considerados. Por exemplo, um determinado defeito incluído na classe *f1* pode ser identificado usando teste *Alpha & Beta*. A coluna “Techniques” mostra o tipo de técnica mais adequada (branca, preta ou cinza) para identificar um dado defeito. A motivação de tais descrições é apoiar a atividade de teste de sistemas multi-tenants a partir da classificação proposta de defeitos. Para a classe *f6.Fragment composition fault*, uma técnica adequada é “caixa-branca” (do inglês, “white-box”),

Tabela 7 – Taxonomia preliminar de defeitos para sistemas SaaS multi-tenants.

Characteristics/sub-characteristics			Classes of faults			Testing level	Techniques	Testing activities	
A.Access via web browser			f1.Web faults/GUI faults			All	Grey-box	Gui-based/scenario testing; Usability testing	
B.High level of customization	1.Multi-templating	f2.Dynamic compilation fault			System	Black-box	Connectivity testing; Compatibility testing; Interoperability testing; Recovery testing; Live upgrading testing; Live state scenario testing; Live re-test		
		f3.Fragments unavailable for designed tenant			Integration	White-box			
		f4.Incorrect fragment for designed tenant			System	Black-box			
		f5.Functional fault			Unit Integration	White-box			
	f6.Fragment composition fault			Integration					
	2.Components shared	f7. Composition fault	f7.1.Timed out			System		Black-box	
			f7.2. No valid composition	f7.2.1.Incompatible components		Integration		White-box	
f7.2.2. Parts of composition missing									
C.Tenant identification			f8.Request fault			Unit		Functional testing	
			f9.Interceptor fault						Connectivity testing
D.Implementation models - database architectures	1.Isolated database		f10.Connection provider fault f11.Fault related to data isolation			Unit	Black-box	Verification of data isolation in tenant level	
	2.Isolated schema					System			
	3.Shared schemas								
E.Application container			f12.Unit fault			Unit	White-box	Functional testing; Behavior testing; Tenant-specific business rules verification	
			f13.Integration fault			Integration			
			f14.System fault			System		Component api testing	
			f15.Fault related to the use of features provided by frameworks/platforms						
F.Security	1.Variant application level	f16.Fault in filters of queries			System	Black-box	Security/reliability testing; Test involving SQL-injection and Cross Site Request Forgery (CSRF); Tests in IaaS provider level		
		f17.Tenants competing for the same database instance							
		f18.Unavailable feature							
		f19.Fault in organizational security							
		f20.Violation of administrative rules							
	f21.Illegal access the data								
G.Deployment environment			f22.Service incompatible			System	Black-box	Performance and load testing; availability testing; Installation/deployment testing	
			f23.Required resource fault						
	1.Hybrid cloud		f24.Faults involving security issues (physical, operational and programmatic); data issues (backup, usage, loss, integrity and theft); performance and bandwidth related issues			System			
	2.Private cloud								
	3.Public cloud								
4.Community cloud									
H.Connection pooling for concurrently running jobs			f25.Synchronization update fault			System		Concurrent/Parallel testing; Performance testing	
			f26.Fault from non-deterministic behavior						
			f27.Communication fault						
I.Attendance to multiple tenants			f28. Execution fault	f28.1.Service crashed			System	Functional testing; Performance testing	
				f28.2.Incorrect result	f28.2.1.Incorrect input		Unit		White-box
					f28.2.2.Service fault		System		Black-box
				f28.3 Timed out	f28.3.1.Service crashed				
					f28.3.2.Communication failure				
J.Virtualization			f29.Fault related to the virtual machines/resources			System	Black-box	Availability and Performance testing	

uma vez que deve-se levar em conta a estrutura interna do código para conduzir os testes. Por fim, na coluna “Testing activities” algumas atividades são indicadas para identificar o defeito correspondente. Células destacadas na cor cinza (f.15 and G-J) representam propriedades ou defeitos relacionados às features externas como ambiente de nuvem e/ou framework. A seguir, descrevem-se mais detalhes sobre as classes de defeitos.

A. Acesso via browser. Como um sistema SaaS é fornecido via Internet, existem muitos defeitos que são comuns em aplicações web e que também podem estar presentes em soluções SaaS (f1), tais como: defeitos relacionados com a incompatibilidade do browser, defeitos na construção dinâmica das páginas web (*client-side pages*), defeitos relacionados com as entradas das páginas do lado do servidor (*server-side pages*), armazenamento incorreto das informações no cache, defeitos no carregamento de frames e assim por diante.

B. Alto nível de customização. Tenants podem configurar um sistema de acordo com suas particularidades e muitas técnicas apoiam essa capacidade, como *Multi-templating* (1) e *Components shared* (2). Trechos de código são combinados para gerar uma versão da aplicação

para cada tenant em ambas técnicas. Na primeira abordagem, a composição é realizada por meio de templates e filtros como declarações *if-else* para incluir fragmentos externos. No segundo caso, as composições são definidas ao permitir o reuso de componentes e objetos estendidos para satisfazer necessidades específicas de um segmento de mercado particular. Por exemplo, *Force.com* utiliza uma arquitetura dirigida a metadados para fornecer uma compreensão das features usadas para disponibilizar as versões específicas para cada tenant (SALESFORCE, 2008).

Como cada versão da aplicação é compilada dinamicamente (*f2*), fragmentos indisponíveis (*f3*) ou incorretos (*f4*) podem existir para um dado tenant, defeitos funcionais (*f5*) e defeitos na composição de fragmentos (*f6*) são defeitos potenciais que podem afetar esse processo. Defeitos na composição (*f7*) podem ocorrer quando certas propriedades não são apoiadas por todas as partes da composição ou o sistema não responde devido a um *timed out* (*f7.1*). No caso de defeito no processo de composição, a *engine* responsável pela composição pode retornar uma composição não válida (*f7.2*) e isso pode ser devido a diversas razões, como componentes incompatíveis (*f7.2.1*) e partes ausentes para composição (*f7.2.2*).

C. Identificação do tenant. Tenants podem ser identificados de várias maneiras. Por exemplo, *Google Cloud Platform* fornece uma API chamada *Namespaces*² que permite mapear uma URL ao selecionar um identificador para cada tenant. Defeitos relacionados com o processo de identificação do tenant são mais frequentes em dois casos: i) quando um usuário de um determinado tenant acessa uma URL correta mas o sistema não responde adequadamente devido um defeito na requisição (*request fault* - *f8*) ou ii) se por alguma razão, o método que intercepta as requisições é incapaz de extrair o identificador do tenant, resultando em um defeito na interceptação (*interception fault* - *f9*).

D. Implementação de modelos de dados. Em relação aos modelos de implementação ou arquitetura de dados multi-tenant, admitiu-se a classificação em duas principais classes de defeitos: defeitos no provedor da comunicação (*f10*) e defeitos relacionados ao isolamento de dados (*f11*). A primeira pode ocorrer se a conexão com o banco de dados está incorreta, os serviços do banco de dados estão indisponíveis ou existem problemas no método que conecta uma aplicação específica de um tenant com seu correto schema ou banco de dados. O segundo está relacionado ao risco com a privacidade dos dados, quando por exemplo um bug resulta no compartilhamento dos dados de um tenant com os demais.

E. Contêiner da aplicação. Uma parte significativa de uma aplicação multi-tenant não é essencialmente específica para o domínio de aplicações SaaS. Logo, defeitos em vários níveis de desenvolvimento podem ocorrer, bem como em aplicações convencionais. Por essa razão, decidiu-se incluir as seguintes classes de defeitos: defeitos que podem ser revelados com testes unitários ou *Unit faults* (*f12*), defeitos identificados com a integração de funcionalidades (*f13*) e defeitos de sistema (*f14*). Muitos defeitos na aplicação podem estar relacionadas ao uso de

² <https://cloud.google.com/appengine/docs/java/multitenancy/multitenancy>

funcionalidades fornecidas por frameworks/plataformas (f15), como o uso incorreto da interface *CurrentTenantIdentifierResolver*³ fornecido pelo Hibernate⁴ para identificação do tenant corrente.

F. Segurança. A complexidade no modelo de computação em nuvem e a falta de padronização torna a segurança uma questão crítica para provedores na nuvem e clientes. Defeitos de segurança podem ter múltiplas razões, até mesmo sob a perspectiva dos desenvolvedores. Por exemplo, se o desenvolvedor esquece de incluir filtros na query que recebe os dados, isso pode resultar em vulnerabilidades de *SQL injection* por exemplo, caracterizando um defeito nos filtros das queries (f16). Se existem tenants concorrendo pela mesma instância do banco de dados (f17) e ocorrem defeitos de sincronização ou de comunicação, um tenant pode ser capaz de acessar dados dos demais. Em relação ao *Nível de variância da aplicação* (1), vários defeitos podem ameaçar a segurança para um tenant específico. Por exemplo, se um componente exigido para o gerenciamento do acesso foi disponibilizado acidentalmente (f18), os módulos dependentes da aplicação podem se tornar vulneráveis. Assim, os defeitos relacionados para a comunicação e práticas de gerenciamento da informação (f19) apoiadas pela aplicação também requerem atenção dos desenvolvedores. No caso de defeitos que violam as regras administrativas (f20), as políticas de acesso serão comprometidas. Com relação aos *2.Requisitos de privacidade*, as principais preocupações dos desenvolvedores devem se concentrar em evitar o acesso ilegal aos dados (f21). Por exemplo, um invasor pode explorar uma fraqueza na pilha de software e obter acesso aos dados.

G. Ambiente de implantação. Não se pode ignorar o fato de que as características do ambiente de nuvem, como *Nuvem híbrida* (1), *Nuvem privada* (2), *Nuvem pública* (3) ou *Nuvem comunitária* (4) pode impactar a aplicação. Por exemplo, a capacidade dos serviços fornecidos por um determinado ambiente pode ser incompatível (f22) para uma aplicação multi-tenant ou o defeito pode estar relacionado a um recurso computacional específico (f23). Em geral, a principal classe de defeito considerando o ambiente de implantação envolve a segurança, dados e questões de desempenho (f.24). De um lado, se uma aplicação multi-tenant é implantada em uma nuvem pública, os dados são armazenados em data centers dos provedores e o provedor é responsável pelo gerenciamento e manutenção deles, entretanto companhias não podem controlar a segurança de uma nuvem pública. De outro lado, em uma nuvem privada o gerenciamento, manutenção e atualizações dos data centers é responsabilidade da companhia e portanto, todos os dados podem ser protegidos por um firewall.

H. Pooling de conexão para tarefas que executam de forma concorrente. O nível de concorrência entre as versões designadas aos tenants é notável, uma vez que compartilham recursos como computação e armazenamento. Dessa forma, defeitos conhecidos em programas concorrentes, como: *Synchronization update fault* (f25), defeitos devido ao não determinismo ou *Fault from non-deterministic behavior* (f26) e defeitos de comunicação ou *Communication fault*

³ <https://tinyurl.com/ybaom54f>

⁴ <http://hibernate.org/>

(f27) podem afetar as aplicações. Por exemplo, em um determinado programa multi-thread existe uma comunicação implícita entre as threads usando variáveis compartilhadas (SARMANHO *et al.*, 2008). A sincronização, comunicação inter-thread e o não determinismo tornam o teste uma atividade complexa (YANG; POLLOCK, 1997). Múltiplas execuções de um programa concorrente com a mesma entrada pode apresentar diferentes resultados devido às sincronizações diferentes e sequências de comunicação. Ao comparar essa questão com um cenário multi-tenant, uma única instância da aplicação é compartilhada entre múltiplos clientes, portanto é essencial garantir uma execução segura, embora existam múltiplas requisições simultâneas a partir de diferentes usuários de cada tenant.

I. Atendimento a múltiplos tenants. Defeitos de execução (f28) ocorrem quando a aplicação está sendo executada atendendo múltiplos tenants mas o resultado não corresponde ao resultado pretendido e o serviço pode falhar (f28.1). Geralmente, o provedor do serviço na nuvem notifica o cliente sobre a falha. No caso do serviço entregar um resultado incorreto (f28.2.), isso pode ser devido a defeitos no software ou uma entrada incorreta (f28.2.1.). Defeitos no nível do modelos de serviço de entrega (f28.2.2) podem ocorrer dependendo da capacidade/configurações da infraestrutura ou plataforma, como *Timed out* (f28.3) devido a um “service crashed” (f28.3.1.) ou falha na comunicação (f28.3.2).

J. Virtualização. As versões da aplicação designadas aos tenants específicos são isoladas em diferentes espaços virtuais no mesmo hardware enquanto eles compartilham a mesma instância em tempo de execução. A virtualização é usada dinamicamente de acordo com a demanda por recursos (f29). Se uma aplicação é simultaneamente acessada por uma quantidade significativa de usuários, a alocação e virtualização de recursos tem um papel fundamental para permitir que a aplicação continue sendo executada mesmo em condições desfavoráveis como um número expressivo de requisições. Como esse processo é susceptível à defeitos, optou-se em incluir uma classe de defeitos relacionados com a virtualização de recursos.

Embora não tenha sido possível detalhar todos os defeitos e validá-los devido à complexidade de conduzir vários estudos de teste com diferentes técnicas dentro do escopo do doutorado, essa classificação pode ajudar a compreender os defeitos possíveis e definir técnicas mais direcionadas. Essa classificação pode contribuir com o desenvolvimento de aplicações multi-tenants com mais qualidade ao fornecer quais características requerem atenção durante os testes e assim, evitar que certos tipos de defeitos persistam no release final. Além disso, apresentam-se sugestões sobre quais tipos de testes podem ser aplicados para identificação desses defeitos ao longo do processo de desenvolvimento.

Uma preocupação no contexto de programas concorrentes é o *thread safety* (LEWIS; BERG, 1995). Um trecho de código é *thread-safe* quando pode manipular estruturas de dados compartilhadas de forma que ele possa garantir uma execução confiável em vários segmentos ao mesmo tempo. Similarmente, versões específicas podem ser executadas a partir de uma única instância de um sistema SaaS com vários tenants, a qual deve manipular estruturas

de dados compartilhadas. Considerando a natureza das seguintes classes: *Unit faults (f12)*, *Synchronization update fault (f25)*, *Fault from non-deterministic behavior (f26)*, *Communication fault (f27)*, *Service crashed (f28.1)* and *Timed out (f28.3)* e suas similaridades com programas concorrentes e aplicações convencionais, um conjunto ainda mais específico de erros e defeitos foi explorado para serem cobertos pelos estudos discutidos nos próximos capítulos. A Tabela 8 descreve tais elementos de forma separada em dois grupos: Concorrente e Sequencial, incluindo uma breve descrição. A ideia envolve a verificação sob o ponto de vista single e multi-tenant de uma aplicação SaaS, ou seja, encontrar defeitos quando uma versão é executada de forma isolada e depois, quando múltiplas versões começam exercitar as mesmas funcionalidades compartilhadas. Isso remete a necessidade em garantir que a aplicação consiga manipular eficientemente estruturas compartilhadas entre as versões sem afetar os demais tenants.

Tabela 8 – Subconjunto de erros e defeitos identificados a partir da taxonomia.

Defeito/Erro	Tipo	Detalhes
<i>Functionality Error (FE)</i>	Sequencial	Um software possui um FE se algo esperado que ele faça é confuso ou não funciona.
<i>Implementation Error (IE)</i>	Sequencial	Ocorre quando existem inconsistências da aplicação em relação a sua documentação, tais como tamanho dos campos de entrada, tipos de dados e outras restrições.
<i>Control Flow Error (CFE)</i>	Sequencial	Fluxos de controle de um software descreve o que ele fará em seguida e em qual condição. CFE ocorre quando existe um desvio funcional no fluxo de controle no mesmo tenant ou entre diferentes tenants.
<i>Suspension Error (SUE)</i>	Concorrente	As atividades das versões específicas são suspensas ou ficam esperando indeterminadamente.
<i>Missing Command Error (MCE)</i>	Sequencial	Por alguma razão, alguns recursos computacionais pode não estar disponível ou a página não é totalmente carregada em um intervalo de tempo aceitável .
<i>Race Condition (RC)</i>	Concorrente	Um recurso compartilhado depende da ordem de acesso para que as operações sejam executadas adequadamente. Por exemplo, quando um sistema SaaS recebe uma requisição, o identificador do tenant deve ser extraído por meio de http headers ou algum método para identificação do tenant é aplicado. Em condições reais, existem múltiplas requisições simultâneas a partir dos usuários dos tenants e portanto, é importante que a utilização de recursos compartilhados não impactem essa definição.
<i>Memory Inconsistency for tenant identification (MI)</i>	Concorrente	Os tenants possuem a mesma visão da variável usada para sua identificação. Tenant ID é um identificados único para cada domínio ou nome da organização. Entretanto, existem muitas maneiras de identificação dos tenants e é essencial que sejam conhecidos os prós e contras de cada método antes de partir para sua implementação.

Functionality Error (FE) pode ser encontrado quando ao testar uma funcionalidade nota-se uma inconformidade em relação ao comportamento esperado. Embora um FE possa existir em decorrência de outros problemas de acordo com a classificação proposta, ele reflete desvios no software que não necessariamente possuem uma relação com a entrada de dados

mediante ações dos usuários. Um *Implementation Error* (IE) pode ser identificado quando algum campo não atende as restrições de tamanho ou tipos dos caracteres. Critérios funcionais de teste como Particionamento em Classes de Equivalências e Análise do Valor Limite podem ser aplicados de forma complementar para apoiar a elaboração dos casos de teste e identificar possíveis inconsistências.

Fluxos de controle no contexto do software referem-se à ordem em que instruções individuais ou chamadas às funções de um programa são executadas ou avaliadas. O *Control Flow Error* (CFE) é o principal defeito sob interesse do modelo de teste e dos critérios apresentados na próxima seção. CFEs envolvem desvios que podem ocorrer dentro de features de propósito geral, ou seja, funcionalidades que atendem todos os tenants ou desvios que podem ser identificados em features específicas que satisfazem regras de um tenant particular. Esse problema leva uma versão ao redirecionamento para caminhos incorretos e isso é difícil de ser observado e de reproduzir, uma vez que pode não haver mudança no valor do identificador do tenant para a versão sob análise, mas novas definições podem ter sido aplicadas por outros tenants e com a falta de um gerenciamento adequado do isolamento dos dados e principalmente do contexto de execução de cada versão, um usuário final por exemplo pode ter acesso a uma versão diferente da qual deveria ficar restrito.

Suspension Error (SUE) também podem ser identificados nas aplicações multi-tenants quando ao tentar suspender um determinado serviço, essa tentativa resultar em um erro. O aplicativo tenta acessar um objeto de memória que não foi instanciado (ou inicializado) até o momento de sua chamada. *Missing Command Error* (MCE) acontece ao solicitar um determinado recurso não disponível. Testes paralelos e automatizados podem simular várias solicitações de vários usuários dos tenants para verificar se o aplicativo aborda de maneira adequada esse cenário. *Race Condition* (RC) e *Memory Inconsistency for tenant identification* (MI) estão ligados diretamente com o CFE. Embora a alta escalabilidade e virtualização de recursos para atender as demandas em ambiente de nuvem sejam admitidos e que ao compartilhar uma única instância entre múltiplas organizações existe um mecanismo (transparente) de escalonamento de recursos, em determinadas situações a disputa por recursos pode afetar a interceptação e definição da variável de identificação do tenant, caracterizando uma situação do defeito *Race Condition*. O defeito de inconsistência de memória (MI) pode ser visto como uma consequência do RC. O MI ocorre pois mesmo com versões isoladas e transações assíncronas, desvios podem ocorrer em fluxos de execução e uma dada versão pode usar um identificador definido para outro tenant. Em outras palavras, RC e a MI podem levar aos desvios nos fluxos de execução (CFE).

4.7 Considerações finais

Apesar da importância do teste de aplicações residentes na nuvem, a maioria das pesquisas trata da verificação de requisitos não funcionais como escalabilidade, disponibilidade,

confiabilidade, usabilidade, desempenho e segurança, ou seja, esse campo de pesquisa carece de abordagens de teste que explorem aspectos funcionais e os mecanismos internos que tornam uma aplicação multi-tenant. Com base nos estudos mencionados nesse capítulo, notou-se que os tópicos que requerem maior investigação são: (i) exploração de critérios de teste para sistemas SaaS, incluindo a condução de estudos experimentais considerando a aplicabilidade e a efetividade, (ii) estratégias de teste para componentes de customização, (iii) alternativas para verificar as interfaces de composição e o impacto na inserção de novos componentes e (iv) abordagem para automatização do teste de regressão. Uma classificação preliminar de defeitos também foi discutida explorando os elementos essenciais de soluções SaaS multi-tenants, um trabalho inicial que abre possibilidades para contribuições direcionadas a vários tipos de defeitos.

Este capítulo abordou as principais contribuições na área de teste em aplicações SaaS, mostrando uma visão geral das iniciativas e a diversidade de problemas que precisam ser tratados no modelo de computação em nuvem. Apesar das alternativas ainda serem escassas e muitas não envolverem aplicações práticas, os esforços na condução de estudos experimentais de técnicas e critérios de testes em aplicações SaaS considerando suas particularidades são bem vistos pela comunidade, uma vez que os resultados podem orientar novas pesquisas e auxiliar os desenvolvedores no desafio de atingir níveis mais efetivos de qualidade. No Capítulo 5 são apresentados alguns estudos preliminares envolvendo a técnica funcional de teste, especificamente adotando testes automatizados e paralelos.

ESTUDOS PRELIMINARES ENVOLVENDO A TÉCNICA FUNCIONAL DE TESTE

5.1 Considerações iniciais

Estudos recentes demonstram a preocupação com a atividade de teste no domínio de computação em nuvem, entretanto faltam estudos empíricos e diretrizes de quais técnicas e como ferramentas de automatização podem ser empregadas para atender determinados objetivos e considerando os elementos fundamentais para uma aplicação multi-tenant (PINTO *et al.*, 2016). Conforme discutido em capítulos anteriores, os sistemas SaaS possuem certas similaridades com outros modelos e investigar técnicas de teste e ferramentas direcionadas para tais modelos pode trazer informações relevantes em termos de aplicabilidade e efetividade. Embora os testes de aceitação automatizados sejam capazes de revelar diversos tipos de defeitos, não foram encontrados estudos empíricos envolvendo esse tipo de abordagem para sistemas multi-tenants.

Com o intuito de explorar a capacidade da técnica funcional de teste no contexto de aplicações multi-tenants, dois estudos preliminares foram realizados utilizando testes de aceitação automatizados com execução paralela. Nesses trabalhos várias versões de sistemas SaaS distintos foram verificadas simultaneamente. No primeiro, um estudo de caso foi realizado com um sistema que apoia a criação de cardápios eletrônicos para restaurantes, chamado “*iCardapio*” (MANDUCA *et al.*, 2014). O objetivo desse estudo foi verificar o impacto do aumento no número de tenants na quantidade de defeitos utilizando os testes paralelos (PINTO *et al.*, 2018). Em linhas gerais, notou-se que à medida que novos tenants são inseridos, as chances para desvios nos fluxos de execução também aumentam, além da quantidade de defeitos de natureza sequencial. As limitações desse trabalho levaram a condução de novos experimentos e caracterizam o segundo estudo. Na primeira investigação, os dados são puramente quantitativos em relação à quantidade de defeitos e a execução dos testes foi em ambiente local, ou seja, não há reflexões sobre quais tipos de defeitos foram identificados e não havia garantia que os dados

coletados não foram influenciados pelo ambiente de testes. De posse disso, no segundo estudo adotou-se além do *iCardapio*, a aplicação de e-commerce *MtShop* que possui complexidade e modelo de implementação diferente. Com relação à execução dos testes, adotou-se um TaaS que permitiu executar os testes em máquinas virtuais isoladas, garantindo a confiabilidade dos dados. Nesta evolução, procurou-se rastrear e identificar os tipos de defeitos em ambas aplicações. Tais contribuições evidenciam aspectos relevantes que podem auxiliar o desenvolvimento com mais qualidade. Destaca-se a necessidade das verificações em ambiente que simula múltiplas requisições de diferentes tenants, defeitos no acesso às versões e gerenciamento do isolamento dos recursos podem gerar muitos prejuízos e que embora o teste de aceitação não seja tão rigoroso quanto outras abordagens de teste, mostrou-se capaz de indicar a existência desse tipo de defeitos. Neste capítulo apresentam-se detalhes sobre os estudos e os resultados das execuções dos testes.

5.2 Avaliação do teste de aceitação do usuário para sistemas SaaS multi-tenants

O sistema *iCardapio* foi implantado em um ambiente de nuvem chamado Heroku¹, uma PaaS que permite os desenvolvedores construir, executar e operar aplicações completamente em ambiente de nuvem. Com base nessa aplicação, o teste de aceitação com execução paralela foi aplicado para identificar defeitos, incluindo tanto os cenários single-tenant quanto o multi-tenant. Um defeito é encontrado quando a saída de um caso de teste é diferente do resultado esperado, segundo às especificação da aplicação. Isso significa que os defeitos não são controlados, ou seja, eles são reais. Simular um cenário multi-tenant real com requisições dos tenants para uma instância compartilhada da aplicação é essencial para identificar outros tipos de defeitos. Assim, o objetivo deste estudo inicial foi reportar o quanto essa abordagem é capaz de determinar defeitos específicos para as versões das aplicações e investigar seu impacto quando as operações dos usuários são realizadas simultaneamente com diferentes tenants. Os resultados desse estudo exploratório podem ajudar os desenvolvedores em determinar possíveis defeitos, ao empregar uma estratégia de identificação e persistência dos dados dos tenants.

Considerando o escopo deste estudo, a razão para executar os testes paralelos deve-se a necessidade de simular requisições simultâneas a partir de diferentes tenants para uma instância única e compartilhada do *iCardapio*. Com relação à geração dos casos de teste, os critérios de teste funcional EPC e BVA foram aplicados. O *iCardapio* foi escolhido por ser open-source e atender múltiplos tenants a partir de uma única instância. Assim, as versões podem ser definidas e configuradas para execuções simultâneas dos testes. De acordo com os autores (MANDUCA *et al.*, 2014), essa aplicação foi desenvolvida com as seguintes tecnologias: Spring², Spring Tool

¹ <https://www.heroku.com/>

² <https://spring.io/>

Suite³, MySQL, Maven, EclipseLink⁴ e Java. Destaca-se que o código fonte está disponível⁵ e foi projetado para permitir qualquer provedor SaaS realizar o download e utilizá-lo diretamente.

Na Figura 20 ilustra-se um diagrama entidade relacionamento dessa aplicação. Um tenant refere-se um a restaurante específico que tem um identificador único, ou seja, um tenant ID chamado “*subdomain*”. Produtos possuem uma única categoria, como: *pizza*, *pasta*, *drinks* ou *desserts*. Existe somente um usuário que possui acesso autenticado, capaz de realizar as operações de registro de dados para todos os tenants possíveis.

Figura 20 – Diagrama de entidade relacionamento do iCardapio.

Informações gerais sobre a estrutura do projeto e complexidade⁶ foram coletadas com apoio da ferramenta de gerenciamento da qualidade de código SonarQube⁷ e os dados são descritos na Tabela 9.

Tabela 9 – Informações sobre o iCardapio coletadas pela SonarQube.

Linha de código	1.552
Java	843
CSS	97
JavaScript	9
JavaServer Pages (JSP)	239
XML	166
Complexidade	
Complexidade/classes	7.0
Complexidade/metodos	2.3
Unidades de implementação	37
Número de classes	20
Número de métodos	2.159

Com relação à identificação dos tenants, a aplicação utiliza o tenant ID no hostname (<http://tenantid.myapplication.com>). Entretanto, por conta de restrições no ambiente de nuvem, a identificação foi alterada para manter o tenant ID na URI (<http://myapplication.com/tenantid/...>). O isolamento dos dados dos tenants é um requisito importante ao fornecer uma aplicação multi-tenant. Como modelo de implementação para os dados, a aplicação utiliza um schema

³ <https://spring.io/tools/sts>

⁴ <http://www.eclipse.org/eclipselink/>

⁵ <https://github.com/michetti/icardapio/tree/multitenant>

⁶ <https://docs.sonarqube.org/display/SONAR/Metrics+-+Complexity>

⁷ <https://www.sonarqube.org/>

compartilhado, no qual os dados para todos os tenants são armazenados nas mesmas tabelas e podem ser identificados por meio de uma coluna que descreve o tenant.

Os testes automatizados foram executados localmente para assegurar que o ambiente de teste fosse controlado de forma mais efetiva e diferentes tipos de defeitos pudessem ser isolados. Assim, as ações dos usuários são simuladas a partir de diferentes instâncias locais do browser. Os scripts de teste podem ser executados automaticamente na nuvem por meio de um provedor TaaS.

5.2.1 Planejamento

Este estudo de caso teve como objetivo responder as seguintes questões de pesquisa:

- *QP1: Existe uma diferença significativa no número de defeitos identificados quando os cenários de teste single-tenant e multi-tenant são utilizados?* Para responder essa questão, o número de defeitos identificados foram coletados e avaliados quando somente uma versão do *iCardapio* foi submetida aos testes e depois, duas versões com os mesmos testes de forma paralela. O objetivo dessa comparação é fornecer evidências de que, em um cenário multi-tenant, fatores externos (como requisições simultâneas dos usuários para as versões) podem afetar suas execuções e resultar em um número maior de defeitos do que em um cenário single-tenant.
- *QP2: O número de defeitos identificados quando um cenário de teste multi-tenant é utilizado aumenta a medida que novos tenants são incorporados na aplicação?* Para responder essa questão, o número de defeitos foi analisado levando em conta o número de versões sob os testes paralelos. Em outras palavras, a intenção desse questionamento é verificar se o número de tenants exerce um impacto direto na possibilidade de novos defeitos.

A fase de planejamento foi dividida em três partes que são descritas nas próximas subseções.

5.2.1.1 Configuração do experimento

A condução deste estudo de caso incluiu as seguintes fases:

- Geração dos casos de teste de acordo com os critério EPC e BVA para o *iCardapio* com base em seus requisitos;
- O *Selenium* e as ferramentas do *JUnit* foram utilizadas para automatizar a execução de tais casos de teste;
- Implantação do *iCardapio* no Heroku;

- Desenvolvimento de uma infraestrutura de teste que permite a execução paralela dos testes para múltiplas versões de sistemas multi-tenants;
- Execução das classes de teste para uma única versão da aplicação fornecida por uma instância do *iCardapio*, cenário single-tenant (ST);
- Execução das classes de teste de forma paralela para várias versões a partir de uma instância do *iCardapio*, cenário multi-tenant (MT);
- Coletar, avaliar e reportar os resultados dos testes.

Foram 10 execuções dos testes em ambos cenários, esse valor foi assumido como uma medida suficiente para uma análise confiável, as repetições servem como uma medida de confiabilidade. Com relação ao número de versões, foram apenas uma para ST e para as execuções paralelas no cenário MT, os números foram definidos como 2, 5 e 10. A ordem de execução entre as versões não foi pré-determinada, o que torna um comportamento real e provável não determinístico. No caso do cenário MT, a média foi calculada com base no número de versões (n) sob teste (2, 5 e 10), como descrito na Equação 5.1. Por exemplo, a partir de 10 execuções de 2 versões ($n = 2$) realizadas de forma paralela e a detecção de seus defeitos correspondentes, o número de defeitos identificados foram adicionados e divididos por n para cada execução. Por fim, as médias por execução foram adicionadas e divididas por 10. O mesmo cálculo foi aplicado quando $n = 5$ e $n = 10$.

$$\frac{1}{10} \sum_{z=1}^{10} \left[\frac{1}{n} \sum_{i=1}^n x_i \right] \quad (5.1)$$

Em relação ao host para a execução dos testes, as configurações foram as seguintes: Ubuntu 16.04 64 bits, CPU Intel Core i5-3210M 2.5 GHz, 16 GB de RAM, 250 GB SSD e AMD Radeon HD 7670M - 1 GB. Firefox foi o browser escolhido para a execução dos testes. Deve-se destacar que no caso das execuções paralelas, o *Selenium* foi usado em múltiplos ambientes em tempo de execução, especificamente quando haviam instâncias diferentes e isoladas do browser ao mesmo tempo.

5.2.1.2 Formulação das hipóteses

A *QP1* foi formalizada como segue: **Hipótese nula**, H_0 : Não existe diferença significativa entre os cenários de teste ST e MT em termos de defeitos detectados, ou seja, os cenários de teste são equivalentes.

Hipótese alternativa, H_1 : Existe uma diferença entre os cenários de teste ST e MT em termos de defeitos identificados; isso significa que os cenários de teste não são equivalentes. As hipóteses para *QP1* podem ser formalizadas nas Equações 5.2 e 5.3:

$$H_0 : \mu_{ST} = \mu_{MT} \quad (5.2)$$

$$H_1 : \mu_{ST} \neq \mu_{MT} \quad (5.3)$$

Similarmente, a *QP2* foi também formalizada em duas hipóteses, como segue: **Hipótese nula**, H_0 : Não existe diferença estatística em termos de defeitos encontrados durante a execução dos teste, mesmo que número de versões da aplicação seja diferente e que sejam executadas (n e p) ao mesmo tempo .

Hipótese alternativa, H_1 : Existe uma diferença significativa do número de defeitos encontrados durante os testes, uma vez que existe a diferença no número de versões executadas simultaneamente. Similarmente, as hipóteses para *QP2* podem ser formalizadas nas Equações 5.4 e 5.5:

$$H_0 : \mu_{MT^n} = \mu_{MT^p} \mid n \neq p \quad (5.4)$$

$$H_1 : \mu_{MT^n} \neq \mu_{MT^p} \mid n \neq p \quad (5.5)$$

5.2.1.3 Seleção das variáveis

As variáveis dependentes são (i) o “**número de defeitos identificados com as execuções dos testes**” e (ii) “o **número das versões verificadas simultaneamente**”. As variáveis independentes são: (i) “**iCardapio**”, (ii) **cenários de teste** single-tenant e multi-tenant, (iii) o **framework de teste para execução paralela** estabelecida com o apoio do *Selenium* e *JUnit* e (iv) um **ambiente de teste** que inclui os casos de teste e scripts para simular as ações dos usuários.

5.2.2 Execução

Uma vez definido e planejado o estudo de caso, as seguintes etapas foram conduzidas: preparação, execução e validação de dados.

5.2.2.1 Preparação

Nesse estágio o enfoque foi dado para a preparação do material necessário, como formulários para a coleta de dados e as variáveis do ambiente de teste. *iCardapio* foi implantado no Heroku⁸ e uma infraestrutura de teste foi definida para a execução paralela das classes de teste e das versões dessa aplicação.

No cenário de teste single-tenant, um única thread para executar os testes foi suficiente, diferente para o cenário multi-tenant no qual muitas threads são executadas em paralelo para verificação de várias versões. Entretanto, não se pode garantir que todas as tarefas submetidas

⁸ <https://icardapio.herokuapp.com/>

seriam executadas no mesmo intervalo de tempo ou antes de algum evento, uma vez que isso depende o escalonamento das threads e isso não pode ser controlado devido ao comportamento não determinístico. Apesar dessas limitações, um framework foi desenvolvido com extensões no *JUnit*, (especificamente no *Parameterized runner*⁹) de modo que os testes pudessem ser conduzidos de forma paralela e em ambiente local. As classes de teste foram construídas com apoio do *Selenium* e *JUnit* para automatizar e simular ações possíveis dos usuários no *iCardapio*. Adicionalmente, outros objetos foram levados em conta, tais como:

- *Documentação da aplicação*: documentos de orientação técnica para auxiliar no desenvolvimento do *iCardapio*, úteis para definir casos de teste;
- *Formulário para casos de teste*: formulários a serem preenchidos com os casos de teste de acordo com os critérios EPC e BVA;
- *Formulário para coleta de dados*: um documento a ser preenchido com o número de defeitos identificados com a execução do teste de uma determinada versão da aplicação.

A plataforma usada para conduzir o experimento empregou Java como sua linguagem de implementação e a IDE Eclipse como ambiente de desenvolvimento.

5.2.3 Análise dos dados

Nesta seção discutem-se os resultados. A análise é dividida em duas áreas: estatística descritiva e teste de hipóteses.

5.2.3.1 Estatística descritiva

A estatística descritiva dos dados coletados a partir do estudo de caso são apresentados na Tabela 10. A primeira coluna contém a ordem de execução e nas colunas restantes, o número de defeitos identificados em cada versão da aplicação. Em relação ao número de casos de teste, 95 foram definidos usando os critérios EPC e BVA seguindo os requisitos do *iCardapio* e seus possíveis fluxos de execução sob a perspectiva dos usuários. Com base nesse princípio, o número de defeitos reportados neste estudo refere-se aos casos de teste que revelaram defeitos, ou seja, podem existir defeitos que não tenham sido revelados. Ressalta-se que os dados estão agrupados pelo número de versões da aplicação. Isso levou a quatro grupos sendo categorizados da seguinte forma: o primeiro com apenas “t1” representa uma única versão da aplicação, o segundo contém dados de duas versões (“t1” e “t2”, $n = 2$) executadas simultaneamente; o terceiro contém cinco versões (“t1” a “t5”) e por fim, o quarto contém dez versões (“t1” a “t10”).

Antes de aplicar métodos estatísticos é importante determinar a qualidade dos dados de entrada. Conjuntos de dados incorretos pode ser formado de erros sistemáticos ou a presença

⁹ <https://goo.gl/DS6Gkg>

Tabela 10 – Dados coletados com os testes automatizados.

Exec.	t1	t1	t2	t1	t2	t3	t4	t5
1 st	34	37	39	48	47	47	51	47
2 nd	34	38	40	50	49	49	49	48
3 rd	34	37	39	47	47	47	48	50
4 th	34	40	42	48	48	48	51	49
5 th	34	38	40	47	48	49	50	47
6 th	34	37	39	47	47	48	50	49
7 th	34	38	40	51	49	49	49	48
8 th	34	37	42	47	47	47	49	50
9 th	34	39	42	48	48	47	50	49
10 th	34	38	39	47	47	51	48	51

	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10
1 st	46	42	46	45	44	43	51	47	42	45
2 nd	48	46	49	50	42	42	43	49	47	49
3 rd	46	47	48	42	43	50	51	48	49	48
4 th	47	43	47	45	44	44	49	48	49	46
5 th	47	43	45	49	51	50	49	48	44	45
6 th	47	42	47	46	45	44	51	50	43	46
7 th	47	43	47	43	45	47	51	50	45	46
8 th	46	42	47	46	45	50	51	48	43	46
9 th	47	46	45	44	44	43	52	48	43	46
10 th	47	48	48	45	46	50	49	48	49	51

de *outliers*, que são valores muito elevados ou muito abaixo dos esperados ao comparar com os dados restantes. Por essa razão, adotou-se um *box plot* como meio para identificar os *outliers*. Na Figura 21 é ilustrado um *box plot* com base no número de defeitos identificados no cenário de teste single e multi-tenant, onde $n = 1$, $n = 2$ ($t1...t2$), $n = 5$ ($t1...t5$) e $n = 10$ ($t1...t10$). Note que no cenário de teste single-tenant, o número de defeitos foi o mesmo para todas as execuções. Em contraste, no caso do cenário de teste multi-tenant, os resultados a partir da Equação 5.1 para $n = 2$, $n = 5$ e $n = 10$ foram os seguintes: 39.05, 48.44 e 46.49 para os defeitos identificados (em média e incluindo todas as execuções), respectivamente.

5.2.3.2 Teste de hipóteses

Nessa seção discute-se o teste das hipóteses para as questões de pesquisa.

QPI - Teste de hipóteses: Como alguns testes estatísticos só são válidos se houver uma distribuição normal da população, antes de optar o teste estatístico é importante avaliar se os dados coletados possuem um nível de linearidade. Para isso foi empregado o teste *Shapiro-Wilk* para a amostra dos defeitos identificados. Com relação aos dados de $t1$ e $n = 2$ ($t1...t2$), a variável estatística W do teste *Shapiro-Wilk* foi calculada como 0.874, o valor crítico do valor de W (5%

Figura 21 – Box plot para os defeitos identificados com um crescente número de versões.

para nível de significância) foi 0.930 e o p -value foi 0.002061, assumindo $\alpha = 0.05$. Portanto, rejeitou-se a hipótese nula de que os dados representam uma população normal distribuída, visto que o valor de W é menor que o valor crítico de W e $p < 0.05$. Adicionalmente, o teste $Q-Q$ plot foi aplicada, conforme ilustrado na Figura 22.

Figura 22 – Um teste de normalidade para os dados coletados quando as versões do *iCardapio* foram submetidas ao teste automatizado.

Posteriormente, o teste *Wilcoxon-Mann-Whitney* foi utilizado, trata-se um teste não paramétrico para duas amostras não pareadas. Esse teste estatístico avalia a existência de uma diferença significativa em variáveis de nível ordinal e dependentes. A hipótese alternativa H_1 é que o número de defeitos identificados quando uma única versão (t_1) é executada e quando duas versões são executadas simultaneamente, isto é, em um cenário de teste multi-tenant com $n = 2$, as populações não são idênticas estatisticamente. Com um resultado, o p -value foi $7.186e^{-06}$; portanto, com um nível de significância 0.05, existe a diferença entre as amostras.

QP2 - Teste de hipóteses: de forma similar, o teste de normalidade *Shapiro-Wilk* foi aplicado para os dados referentes a $n = 2$, $n = 5$ e $n = 10$. Como resultados, W foi 0.922, o valor crítico de W (5% de nível de significância) foi 0.984 e p -value foi $6.797e^{-08}$, assumindo $\alpha = 0.05$. Portanto, rejeitou-se também a hipótese nula para QP2 de que os dados são provenientes

de uma população normalmente distribuída. Na Figura 23 ilustra-se um $Q-Q$ plot para os dados coletados que inclui somente o cenário multi-tenant.

Figura 23 – Teste de normalidade quando 2, 5 e 10 versões foram submetidas ao teste automatizado.

Posteriormente, aplicou-se o teste *Kruskal-Wallis*, que é um teste não paramétrico que avalia diferenças significativas em variáveis dependentes, contínuas e em nível ordinal por uma variável categórica e independente com duas ou mais amostras não pareadas. Como um resultado, o qui-quadrado do *Kruskal-Wallis* foi 68.764, $df = 2$, H foi 67.94, qui-quadrado crítico foi 5.991 e o p -value foi $1,221e^{-15}$, quando $\alpha = 0.05$. Assumindo que $p < 0.05$ e o qui-quadrado crítico é menor que o H estatístico, a hipótese nula que as medianas são equivalentes pode ser rejeitada. Portanto, existe uma diferença significativa entre as três amostras ($n = 2$, $n = 5$ e $n = 10$).

Como a hipótese alternativa H_1 assume-se que pelo menos uma das amostras é proveniente de uma população diferente das demais, para essa verificação aplicou-se o método *Bonferroni* para comparações múltiplas com $FWER = 0.05$. Isso permitiu concluir que cada amostra é estatisticamente diferente. A diferença entre $n = 10$ e $n = 2$ foi 73.025, com o limite superior de 50.58 e o limite inferior de 95.48. Para $n = 10$ e $n = 5$, a diferença foi -34.14 , com um limite superior de $-50,02$ e limite inferior de $-18,27$. Por fim, entre $n = 2$ e $n = 5$, foi de $-107,165$, com o limite superior de -131.42 e o limite inferior de -82.92 . Com relação aos agrupamentos, os ranques para $n = 5$, $n = 10$ e $n = 2$ são os seguintes: 118.19, 84.05 e 11.025, respectivamente.

Adicionalmente, uma gráfico de efeitos, ou *effects plot*, foi definido a partir dos efeitos, limites inferior e superior ao analisar essas três amostras separadamente, como mostrado na Figura 24. Para $n = 10$, os valores foram os seguintes: 46.049, 46.49 e 46.931; com $n = 2$: 38.063, 39.05 e 40.036; e por fim, para $n = 5$: 47.815, 48.44 e 49.064. Com esse gráfico é possível examinar as diferenças entre as médias de nível para os fatores analisados. Para $n = 5$ a ocorrência de defeitos foi maior que nos outros cenários, mas com um valor aproximado quando dez versões foram executadas.

Figura 24 – Grafo de efeitos para o cenário de teste multi-tenant.

5.2.3.3 Discussão sobre a natureza dos defeitos identificados

A) *Defeitos single-tenant*: antes de analisar os defeitos identificados quando somente uma versão foi definida para os testes automatizados, concluiu-se que todos eram de natureza sequencial, típicos em aplicações web. O comportamento da aplicação foi consistente em relação a sua especificação e apenas 34 casos de teste revelaram defeitos na aplicação. A identificação de defeitos no cenário single-tenant não exigiu um esforço considerável, uma vez que os testes são realizados em nível de unidade. Deve-se notar que não houve concorrência com outros tenants para os mesmos recursos, portanto é possível outros tipos de defeitos em relação ao desempenho, segurança e privacidades dos dados ainda possam ser revelados.

B) *Defeitos multi-tenant*: uma questão importante é como identificar defeitos que podem apenas ser descobertos em tempo de execução e em um cenário real. Por essa razão, testes de aceitação automatizados e executados em paralelo podem ser vistos como uma abordagem válida. No contexto de multi-tenancy é notável a existência de problemas relacionados com o comportamento concorrente nas aplicações desenvolvidas para nuvem as quais requerem investigações futuras mais aprofundadas.

O desafio de identificar um defeito neste tipo de aplicação surge principalmente a partir do processo de sincronização (ou seja, threads). Os tenants concorrem por recursos compartilhados que são disponibilizados sob demanda, ou seja, uma política de escalonamento deve compartilhar e distribuir os recursos de uma maneira equilibrada. Isso significa que não podemos controlar a ordem de execução dos processos, ou seja, a dependência dos dados e as sincronizações intra-processos de uma aplicação podem ser afetadas durante o escalonamento. Assim, uma aplicação pode realizar diferentes sequências de execução com as mesmas entradas de teste. Portanto, a atividade de teste deve verificar se as sequências de sincronização possíveis foram executadas e se as saídas obtidas estão corretas. Essa dificuldade é bem conhecida no campo da programação concorrente, como sendo um tipo de comportamento não determinístico. Existem vários problemas no teste de programas concorrentes devido a essa característica, como alto custo computacional e muitos falsos positivos (ARORA; BHATIA; SINGH, 2016).

Os resultados deste estudo de caso demonstraram que existem diferenças ao executar os mesmos casos de teste quando eles incluem tanto uma única versão quanto várias versões com os testes paralelos. Defeitos ligados às atualizações das sincronizações são provavelmente as principais razões para o número de defeitos no cenário de teste multi-tenant, pois muitas execuções dos testes levaram a caminhos inesperados. Por exemplo, quando os usuários realizam as mesmas ações em diferentes versões, caminhos de execução específicos foram afetados. Isso acontece quando um usuário realiza o *login* e *logout* em uma determinada versão da aplicação. Como esses desvios afetaram negativamente a execução esperada, vários defeitos foram reportados no ambiente de teste.

Com relação aos defeitos nas tarefas realizadas de forma concorrente pelos tenants, a execução paralela das versões permitiu identificar alguns defeitos que poderiam não ser identificados ao verificar uma versão isolada. Na Figura 25 descrevem-se os resultados da execução paralela para um caso de teste que verifica o campo “name” dos produtos. Nesta execução, duas versões foram verificadas com requisições simuladas. Note que o número de defeitos identificados é diferente para cada execução, entretanto as mesmas entradas foram utilizadas e exercitam as mesmas funcionalidades fornecidas pelas versões.

Figura 25 – Resultados da execução do teste automatizado e paralelo que verifica o campo *name* dos produtos em duas versões do *iCardapio*.

5.2.4 Ameaças à validade

Execução dos testes: a capacidade do processamento do host onde os testes foram realizadas e a robustez do provedor da nuvem podem influenciar os resultados dos testes (BAI *et al.*, 2011). Entretanto, essa influência não foi mensurada e comparada com outras configurações. Para mitigar essa ameaça, todos os testes foram conduzidos na mesma configuração de hardware e sistema operacional. Embora 10 execuções tenham sido definidas, podem existir defeitos não

identificados na aplicação ou falso positivos. Como os defeitos examinados neste estudo de caso não foram “injetados” (como em um ambiente controlado, no qual um conjunto pré-definido de defeitos tenha sido incorporado à aplicação), os casos de teste foram definidos manualmente para explorar os fluxos de execução a partir da perspectiva dos usuários e de acordo com os requisitos do *iCardapio*, e por essa razão, os defeitos foram reportados a partir desses casos de teste.

Casos de teste: projetar casos de teste é uma tarefa importante que pode afetar os dados coletados e tornar o estudo de caso menos imparcial. Um profissional de teste que tinha somente acesso à documentação e as interfaces da aplicação gerou os casos de teste para garantir a imparcialidade e evitar o viés de conhecer o código da aplicação.

Interação entre a configuração e o tratamento: é possível que o estudo realizado não seja preciso o suficiente para qualquer aplicação no mundo real. No entanto, o principal valor deste estudo empírico é que ele forneceu evidências de que se a atividade de teste inclui apenas uma versão, é incapaz de detectar os principais defeitos devido aos fatores externos, como o compartilhamento de recursos e a concorrência entre as versões que podem resultar em efeitos não determinísticos.

Nível de confiabilidade: refere-se as métricas usadas para avaliar o número de defeitos. Para lidar com essa ameaça foram admitidas 10 execuções que pode ser considerada como uma medida suficiente, visto que número de defeitos identificado foi registrado para cada execução das versões.

Capacidade estatística: caracteriza-se pela capacidade de um teste estatístico em revelar dados confiáveis. Dois testes estatísticos foram aplicados o *Wilcoxon-Mann Whitney* para analisar as diferenças entre duas amostras não pareadas e o *Kruskal-Wallis* para analisar o número de defeitos encontrados quando um número diferente de versões foram verificadas simultaneamente.

5.3 Investigação do teste automatizado envolvendo sistemas multi-tenants com diferentes arquiteturas de dados

No estudo anterior os principais resultados demonstraram a diferença entre o número de defeitos identificados levando em conta as amostras que variavam o número de versões sob teste. Entretanto, tais resultados foram quantitativos e limitados pois os testes paralelos foram executados e um único host. Nesse sentido, os scripts de teste desenvolvidos neste segundo estudo foram executados com máquinas virtuais (*Virtual Machines*, VMs) isoladas. Os defeitos reportados foram analisados considerando os conhecidos em áreas relacionadas, como programação concorrente e aplicações web (detalhados no Capítulo 4). Neste cenário, as principais contribuições desse segundo estudo são: (i) evidências experimentais do teste automatizado e paralelo para aplicações multi-tenants, (ii) análise da natureza dos defeitos

identificados em duas aplicações representativas e (iii) discussões sobre defeitos específicos que são difíceis de serem reproduzidos em um cenário single-tenant restrito, mas caso ocorram podem trazer efeitos indesejados para a privacidade dos dados.

Considerando o exposto, o objetivo deste estudo foi responder as seguintes questões de pesquisa:

- **Questão de Pesquisa 1 (QP1):** *Como o teste automatizado pode contribuir para o teste multi-tenant?* Multi-tenancy aumenta a complexidade para a atividade de teste. Como geralmente existe uma possibilidade expressiva de configurações em um SaaS multi-tenant, existem vários tipos de defeitos e portanto, verificar todos os casos é uma tarefa difícil. Assim, a ideia é avaliar a capacidade do teste automatizado em revelar defeitos principalmente relacionados com a definição da variável utilizada para a identificação do tenant e seus usos subsequentes.
- **Questão de Pesquisa 2 (QP2):** *Como diferentes modelos de implementação de aplicações podem impactar a atividade de teste?* Implementações para aplicações multi-tenant podem diferenciar significativamente em termos de arquitetura de dados, dentre outras características. Por essa razão, garantir que todos os fluxos de execução sejam verificados sem impactos às features previamente validadas para os demais tenants é uma tarefa complexa.

5.3.1 Ferramentas e aplicações utilizadas no experimento

As mesmas ferramentas utilizadas no estudo anterior foram empregadas neste experimento, mas para a execução dos testes na nuvem adotou-se o SauceLabs¹⁰ que é uma ferramenta de automação multi-browser que permite realizar testes em um ambiente de nuvem. SauceLabs fornece um ambiente completamente configurado com um conjunto pré-instalado de ferramentas para depurar os scripts de teste em tempo real (SELENIUM, 2017). A plataforma disponibilizada pela SauceLabs foi adotada pois oferece facilmente as VMs isoladas, um requisito importante para evitar possíveis interferências devido ao escalonamento de recursos computacionais, resultando em análises incorretas das amostras de defeitos.

Assim como no primeiro estudo, os critérios de EPC e BVA foram aplicados para projetar os casos de teste. As aplicações utilizadas nos testes paralelos foram a *iCardapio* e o *MtShop*¹¹, cujo desenvolvimento foi motivado pela falta de exemplos completos disponíveis e a necessidade de um embasamento técnico para as pesquisas, conforme comentado no Capítulo 3.

¹⁰ <https://saucelabs.com>

¹¹ <https://mtshopfails.herokuapp.com>

5.3.2 Seleção de variáveis

As variáveis dependentes são (i) o “**número de defeitos identificados a partir das execuções dos testes**” e (ii) “**o número de versões das aplicações *iCardapio* e *MtShop* utilizado para cada execução dos testes**”. As variáveis independentes são: (i) “*iCardapio*”, (ii) “*MtShop*”, (iii) **cenários de teste** com uma única versão e depois com as versões restantes das aplicações (iv) **framework de teste para execução paralela** estabelecido com *Selenium* e *JUnit* e (v) um **ambiente de teste** que inclui casos de teste e scripts para simular as ações dos usuários.

5.3.3 Execução

Com os estudos de caso definidos e planejados, eles foram conduzidos nas seguintes fases: preparação, execução e validação dos dados.

5.3.3.1 Preparação

Nessa fase os materiais necessários como formulários de coleta de dados e variáveis do ambiente de testes foram preparados:

- *Documentação de ambas aplicações*: documentos técnicos que direcionaram o desenvolvimento, os quais são úteis para a definição de casos de teste;
- *Formulários para Casos de Teste*: formulários para serem preenchidos com os casos de teste de acordo com os critérios EPC e BVA;
- *Formulário para Coleta de Dados*: uma planilha foi criada para a coleta de dados com o tipo e número de defeitos identificados com a execução de teste das versões.

Embora as aplicações selecionadas para o estudo experimental possam ter diversos tipos de defeitos, admitiu-se apenas aqueles classificados na Tabela 8. Três execuções de teste foram planejadas para o *MtShop* e *iCardapio*, primeiro considerando apenas uma única versão e depois com duas e três versões das aplicações de forma paralela. Primeiramente, as execuções demonstraram que esse número poderia ser assumido como uma medida satisfatória para uma análise confiável, pois ao aumentar o número de execuções o efeito foi em termos quantitativos e não na diversidade dos defeitos. A ordem de execução entre as versões não foi pré-definida da mesma forma que no estudo anterior que envolveu somente o *iCardapio*.

Para exemplificar uma parte representativa do formulário de coleta de dados, a Tabela 11 contém os resultados da execução de 8 casos de teste para o campo “name” da classe “Customer”. Segundo a especificação do *MtShop*, esse campo não pode ser nulo mas deve conter uma string de até 70 caracteres, sem incluir caracteres especiais ou de pontuação. Quando um visitante tenta se registrar sem informar o valor para esse campo, a aplicação deve retornar a mensagem: “*Fill out this field*” e caso o valor não seja válido: “*Please, check your Name*”.

Na primeira linha é definida a ordem de execução e a versão da aplicação sob teste: “Third execution [2] - *MtShop*” significa que os dados referem-se à execução de três versões da aplicação *MtShop* mas os dados coletados são da aplicação designada ao “tenant 2”. Ao executar o framework de teste para mais de uma aplicação, assume-se que as threads correspondentes são criadas em tempo de execução, de modo que cada thread é responsável exclusivamente por uma única versão da aplicação. Por exemplo, ao executar os scripts de teste para cinco versões, as threads [1]...[5] devem ser criadas. A segunda linha apresenta o nome da Classe de teste contendo seus métodos e elementos. Isso é importante para manter a rastreabilidade entre as classes de teste e seus campos. Na coluna chamada “Test Case” descrevem-se os métodos que foram projetados na classe de teste, de modo que cada método esteja associado com um caso de teste específico. Dessa forma, pode-se identificar quais casos de teste revelaram algum defeito. Na coluna “Fault” descrevem-se os tipos de defeitos identificados de acordo com a Tabela 8. As seguintes colunas apresentam o tempo gasto para cada caso de teste em “Time (s)”, “Description log” para os logs da execução dos testes, “Details” para descrever as razões para os defeitos e “Situation” para os resultados dos testes.

Tabela 11 – Exemplo do formulário utilizado para a coleta de dados.

Third execution [2] - <i>MtShop</i>					
CustomerNameTest					
TestCase	Fault	Time (s)	Description log	Details	Situation
test01	SUE	299.99	org.openqa.selenium. TimeoutException		ERROR
test02	IE	22.24	java.lang. AssertionError: expected:<false>but was:<true>	The field does not meet size requirements or character types	FAILURE
test03	IE	7.85	java.lang. AssertionError: expected:<false>but was:<true>	The field does not meet size requirements or character types	FAILURE
test04	NONE	12.01	NO LOG	-	PASS
test05	NONE	12.63	NO LOG	-	PASS
test06	RC/MI	11.78	org.junit. ComparisonFailure: expected: <...erokuapp. com/tenant_[2] /login?logout>but was:<...erokuapp. com/tenant_[1]/ login?logout>		FAILURE
test07	NONE	11.79	NO LOG	-	PASS
test08	NONE	36.38	NO LOG	-	PASS

5.3.4 Análise de Dados

Nesta seção os dados coletados durante o estudo experimental são analisados e discussões em relação às questões de pesquisas são apresentadas.

5.3.4.1 Estatística Descritiva

A seguir descrevem-se a quantidade e os tipos de defeitos identificados nas aplicação com apoio dos testes automatizados. Esses dados são analisados e sumarizados para apoiar as discussões sobre os defeitos e a complexidade envolvida no teste de aplicações na nuvem. Diferentes cenários de teste com um número variado de versões foram úteis para evidenciar a relevâncias das investigações e a criticidade de tais defeitos. Por fim, discute-se como diferentes modelos de implementação para aplicações multi-tenants comportam-se em um ambiente de teste paralelo e quais são as principais preocupações ao conduzir os testes para tais modelos.

Questão de Pesquisa 1

Para responder a *QP1* a análise foi direcionada à existência de múltiplas versões e a falta de um gerenciamento adequado de recursos compartilhados que podem resultar em mudanças incorretas nos fluxos de execução em cada versão e como os testes automatizados podem ser relevantes para identificar tais problemas. Testes paralelos podem ser aplicados para simular condições reais no ambiente de nuvem, como múltiplas requisições de diferentes usuários dos tenants. Embora existam limitações nessa abordagem de teste, é possível avaliar questões importantes envolvendo os recursos compartilhados e os comportamentos inesperados a partir da visão dos usuários.

Os dados coletados a partir de estudo de caso envolvendo o *iCardapio* são mostrados na Tabela 12, considerando três execuções dos testes. Vale ressaltar que os dados nessa tabela foram sumarizados, mas os resultados brutos das execuções para ambas aplicações está disponível¹² e organizado utilizando o formulário de coleta de dados exemplificado na Tabela 11. A primeira coluna “*Relative Class under test*” contém as Classes verificadas com o teste automatizado que insere dados nas páginas web e realiza ações simuladas dos usuários. A segunda coluna sumariza os tipos de defeitos identificados. Nas colunas restantes descrevem-se o número de defeitos em cada versão das aplicações. Tais dados são agrupados pela ordem de execução. Isso implica em três grupos sendo categorizados da seguinte forma: a primeira execução com apenas “*t1*” representa uma única aplicação, a segunda execução contém os dados a partir de suas versões do *iCardapio* chamadas “*t1*” e “*t2*”, e a terceira execução com os dados de “*t1*”, “*t2*” e “*t3*”.

Para o *iCardapio*, 86 casos de teste foram projetados segundo seus requisitos. Desse modo, o número de defeitos reportados neste estudo refere-se aos casos de teste que revelaram defeitos, 52/86 casos de teste passaram na primeira execução. Para a segunda, 43/86 não revelaram defeitos quando duas versões foram verificadas de forma paralela e na última execução, 42/86 não revelaram defeitos em média.

Embora existem certas particularidade, defeitos típicos de aplicações convencionais e defeitos devido ao comportamento não determinístico podem ser evidenciados em aplicações

¹² <https://tinyurl.com/yybmj6br>

desenvolvidas para nuvem. Quando apenas uma versão ($t1$) foi executada não houveram desvios nos fluxos de execução, RC/MI (*Race Condition/Memory Inconsistency for tenant identification*) que poderiam levar ao uso incorreto da variável de identificação dos tenants nas versões correspondentes. Entretanto, quando os mesmos testes foram executados para duas e três versões de forma paralela, os resultados foram diferentes: 4 e 6 defeitos para $t1$ e $t2$, e 7, 10 e 3 para $t1$, $t2$ e $t3$, respectivamente. Nota-se um ligeiro aumento no número de defeitos observados devido as requisições simuladas pelos testes automatizados, com exceção para a $t3$ na terceira execução. Muitos defeitos reportados estão vinculados às deficiências nos controles do isolamento das sessões e escalonamento de recursos, algo identificado no estudo anterior envolvendo o *iCardapio*. Por exemplo, quando um recurso computacional é mais consumido para um tenant do que os demais, isso pode afetar a execução da aplicação para os mesmos, resultando no SUE (*Suspension Error*). Note que na linha IE (*Implementation Error*) para *Restaurant*, quando apenas uma única aplicação foi executada, 20 defeitos foram identificados e quando três versões foram verificadas com requisições simultâneas, o número para o $t1$ foi 39, enquanto para $t2$ e $t3$ foi 20. Provavelmente, as operações para a aplicação designada para o “tenant 1” não foram completadas, resultando em erros ao priorizar as operações para as demais versões.

Tabela 12 – Dados coletados com a execução dos testes para as versões do *iCardapio*.

Relative Class under test	Fault/Error	1st Exec.			2nd Exec.			3rd Exec.		
		t1	t1	t2	t1	t2	t3	t1	t2	t3
Product [Description, Name, Price]	RC/MI	0	4	6	7	10	3			
	SUE	0	0	0	0	0	0			
	IE	12	16	13	14	13	18			
	FE	0	0	0	0	0	0			
	CFE	0	0	0	0	0	0			
	MCE	4	9	4	5	7	10			
Restaurant [Address, City, Name, Phone number, Slogan and Subdomain]	RC/MI	0	0	0	0	0	0			
	SUE	0	0	0	0	0	0			
	IE	20	22	21	39	20	20			
	FE	0	0	0	0	0	0			
	CFE	0	0	0	0	0	0			
	MCE	0	0	0	0	0	0			

Da mesma forma, a Tabela 13 sumariza os resultados para o *MtShop*. Um total de 184 casos de teste foram projetados seguindo seus requisitos. Na primeira execução com apenas uma versão da aplicação, 166/184 casos de teste não revelaram defeitos. Para a segunda execução, o valor foi de 177/184 em média, dado que duas versões foram verificadas em paralelo e na última execução, 67/184 não revelaram defeitos. Dessa forma, 90.21%, 63.58% e 36.41% foram a quantidade de casos de teste que não revelaram defeitos em cada execução dos testes, respectivamente. Ao comparar os dados coletados com os testes paralelo do *iCardapio* e *MtShop* pode-se observar que independente das diferenças nos modelos de implementação e complexidade, a maioria dos tipos de defeitos foram identificados em ambas aplicações. O problema com IEs na terceira execução pode ser notado novamente, para o campo “*Billing address*” 54 defeitos foram reportados para a aplicação projetada para o “tenant 1”, um valor considerável ao comparar com o número de defeitos encontrados nas demais versões.

Tabela 13 – Dados coletados com a execução dos testes para as versões do *MtShop*.

Relative Class under test	Fault/Error	1st Exec.			2nd Exec.			3rd Exec.		
		t1	t1	t2	t1	t2	t3	t1	t2	t3
Customer [Name, Email, Phone number, Username and Password]	RC/MI	0	0	0	2	2	0			
	SUE	0	1	0	2	7	8			
	IE	12	5	6	7	12	12			
	FE	0	0	0	0	0	0			
	CFE	1	0	0	2	1	3			
	MCE	0	7	4	5	0	0			
Billing address [Street name, Apartment Number, City, State, Country, Zip Code]	RC/MI	0	0	0	0	1	1			
	SUE	0	0	0	5	13	14			
	IE	14	6	14	54	14	14			
	FE	0	0	0	0	0	0			
	CFE	0	0	0	5	2	4			
	MCE	0	1	0	0	0	0			
Shipping address [Street name, Apartment Number, City, State, Country, Zip Code]	RC/MI	0	0	1	1	2	1			
	SUE	0	0	0	5	6	11			
	IE	20	6	33	20	20	20			
	FE	0	0	0	0	0	0			
	CFE	1	0	1	2	2	2			
	MCE	0	1	0	0	0	0			
Product [Name, Description, Price, Unit, Manufacturer]	RC/MI	0	5	2	17	18	12			
	SUE	4	1	1	4	6	5			
	IE	14	9	9	9	7	8			
	FE	7	1	1	7	7	6			
	CFE	0	3	6	5	3	9			
	MCE	3	11	9	4	9	5			

Na Figura 26 ilustram-se duas colunas com gráficos de barras indicando a quantidade e tipos de defeitos para todas as execuções. Os gráficos na esquerda para o *iCardapio* e na direita para o *MtShop*. Os dados estão agrupados pelo tipo de defeitos e a ordem de execução considerando apenas o maior número de defeitos identificado, independente da versão usada em uma dada execução. O objetivo dessa representação é destacar o aumento gradual de defeitos de natureza sequencial e concorrente à medida que variam-se o número de versões sendo verificadas. Defeitos relativos à RC/MI se tornaram mais notáveis quando as aplicações são verificadas simultaneamente. Durante a execução dos testes, houveram muitos MCE (*Missing Command Errors*), na maioria dos casos, o *Selenium driver* foi incapaz de localizar um elemento web ou algum recurso computacional não estava disponível no tempo esperado pelo usuário nas simulações. Note que o número de SUEs foi crescente devido as ocorrências de *timeout* quando um dado recursos se tornava indisponível. Os CFEs (*Control Flow Error*) ocorreram em todas as execuções para o *MtShop* pois houveram redirecionamentos incorretos e no cenário multi-tenant isso se tornou ainda mais problemático. As IEs (em teoria) permaneceram constantes em todas as execuções, uma vez que os mesmos testes foram aplicados para todas as versões designadas aos tenants que são similares, exceto pelo uso de diferentes identificadores. Para o *iCardapio* existe um agravante em relação à privacidade dos dados pois o banco de dados é completamente compartilhado entre os tenants. Dessa forma, defeitos nos fluxos de execução e permissões podem abrir lacunas comprometendo a segurança para os tenants.

Figura 26 – Distribuição das maiores ocorrências de defeitos por execução e seus tipos para *iCardapio* e *MtShop*.

Questão de Pesquisa 2

O nível de isolamento de uma aplicação SaaS pode variar significativamente dependendo dos requisitos do negócio. Em relação a arquitetura de dados, existem pelo menos três principais abordagens, cada uma delas reside em uma posição diferente na escala entre o isolamento e o compartilhamento, como mencionado anteriormente. As aplicações consideradas nesse estudo foram desenvolvidas com diferentes finalidades e modelos de implementação, o principal enfoque da *QP2* não foi compará-las mas analisar cada uma de forma isolada sob a mesma abordagem de teste.

Quando somente uma versão do *iCardapio* foi verificada, apenas defeitos de natureza sequencial foram identificados, mas na segunda e terceira execuções, desvios ocorreram devido a definição incorreta do identificador do tenant para determinadas versões. Por exemplo, após uma determinada execução um usuário do tenant “X” foi redirecionado para a aplicação projetada

para o “Y” e isso se tornou mais frequente quando três versões estavam sendo verificadas ao mesmo tempo. A utilização de uma variável compartilhada para a identificação dos tenants e o não determinismo contribuíram para o aumento no número desse tipo de defeito.

A variável compartilhada que armazena o identificador do tenant corrente apresenta uma “condição de corrida” (*race condition*) durante a execução de múltiplas versões. Dessa maneira, a probabilidade de um uso incorreto de variáveis compartilhadas aumenta dependendo do número de tenants em execução, ou seja, a disputa por essa feature foi intensificada devido ao número de tenants e o aumento nas requisições para os usuários finais. Uma base de dados totalmente compartilhada nesse cenário pode ser afetada quando essa concorrência ocorre, impactando a leitura, escrita e mudanças no sistema. O modelo arquitetural adotado no *iCardapio* é completamente compartilhado e a maioria das operações requerem a recuperação do tenant corrente várias vezes. Desvios nos fluxos de execução podem também levar ao registro incorreto de dados, permitindo que dados tornem-se disponíveis indevidamente para um tenant diferente do qual originou a operação. Em geral, esses defeitos são difíceis de serem rastreados e podem ser negligenciados, mesmo por um profissional de teste experiente. A verificação de aplicações que usam esse modelo de dados precisa considerar desde a definição de um tenant corrente até as mudanças na base de dados compartilhada.

Apesar do *MtShop* utilizar um banco de dados isolado para cada tenant, a falta de controle no isolamento de sessões por tenant colaboram para o acesso não autorizado aos dados. Isso pode também ser considerado como uma consequência de desvios nas execuções devido a variável compartilhada que guarda o valor do tenant corrente, além do não determinismo. Esse defeito foi identificado com testes paralelos, pois em todas as operações de inserção de dados as rotas (*tenant view-routes*) são verificadas. Por exemplo, quando um determinado usuário autenticado de um tenant tenta acessar uma versão projetada para um tenant diferente, a autenticação permanece válida mas isso não deveria ocorrer. Assim não restam dúvidas que isso pode comprometer a segurança da aplicação e o teste desse tipo de aplicação precisa levar em conta essa questão.

Na Figura 27 ilustram-se dois gráficos *box plots* com base nos defeitos identificados em uma única versão e no cenário de teste com duas e três versões (segunda e terceira execuções) para o *iCardapio* (A) e *MtShop* (B). Esse tipo de gráfico é um método para exibir graficamente grupos de dados numéricos, permite visualizar a distribuição dos dados e fornece uma maneira complementar de definir uma perspectiva do caráter dos dados. O tamanho da box representa a dispersão entre os resultados. Note que a menor dispersão entre os resultados ocorre com um maior número de tenants, isso pode indicar que os testes paralelos com um maior número de versões representa uma maior confiabilidade, pois dessa forma pode identificar um número próximo de uma distribuição linear. Isso se torna ainda mais claro na terceira execução (*3rd Execution*) das versões do *MtShop* (B).

A quantidade de defeitos identificados em média para ambas aplicações seguiu uma ordem crescente. Para o *iCardapio*, a média na primeira execução foi 4 defeitos por classes de

Figura 27 – Box plots para o número de defeitos em média para o *iCardapio* (A) e o *MtShop* (B) nas três execuções.

teste e a distribuição de defeitos foi verificada na segunda execução, a ocorrência de defeitos foram concentrados entre 4 e 6 em média. Com base nos dados coletados na terceira execução, a mediana ficou próxima ao primeiro quartil, portanto os dados são positivamente assimétricos. Em outras palavras, os defeitos podem ter ocorrido apenas poucas vezes, de outro lado, poucos defeitos tendem a ocorrer com maior frequência.

Os *outliers* foram identificados na Parte B e com a análise dos dados, ele representam defeitos difíceis de serem reproduzidos quando somente uma versão é verificada de forma isolada. Complementando a *QP1*, os efeitos da execução paralela dos testes refletem que existem problemas nos usos subsequentes da variável para identificação dos tenants.

Considerando os resultados do *MtShop* é possível destacar algumas inferências. Na primeira execução, a mediana foi 3 defeitos por classes de teste quando apenas somente uma versão do *MtShop* foi verificada, enquanto que a segunda execução mostra um ligeiro aumento no número de defeitos. Na terceira execução, os resultados foram diferentes dos anteriores. A mediana ficou próxima ao primeiro quartil, os dados são positivamente assimétricos a variação da caixa ficou entre 4 e 5 defeitos. O primeiro quartil do último box está alinhado com o terceiro quartil do box definido para a segunda execução. Em geral, a execução de um número crescente de versões demonstra que os dados podem ser confiáveis. Ao analisar as principais razões para tais defeitos, a hipótese é que o número de defeitos revelados pode aumentar exponencialmente à medida que as versões são verificadas de forma paralela, ou seja, isso “força” a ocorrência de certos defeitos, caso existam.

A estratégia de simular múltiplas requisições de forma paralela utilizando VMs foi capaz de demonstrar a existência de desvios nos fluxos de execução nas rotas dos tenants (*tenant view-routing*). Considerando os desafios discutidos no Capítulo 4 essa estratégia pode ser útil para apoiar o teste de regressão para múltiplas versões das aplicações que podem ser afetadas pelas *Mudanças frequentes (C1)*. No caso dos desafios envolvendo *Desempenho (C2)*, notou-se que em determinadas execuções paralelas o escalonamento de recursos não seguiu uma abordagem

balanceada. Portanto, é importante realizar testes de carga para verificar se um determinado tenant requer mais recursos que os demais e se isso está resultando em timed out ou erros de suspensão (SUE).

Problemas envolvendo *Segurança (C3)*, como acesso não autorizado para rotas administrativas e features de autenticação podem ser identificadas ao empregar testes automatizados. Nesse cenários, testes de intrusão (*penetration testing*) podem ser mais conclusivos ao fornecer dados sobre possíveis vulnerabilidades nesse tipo de aplicação. Em relação aos desafios de *Isolamento (C4)* ferramentas de monitoramento e teste são importantes para identificar problemas relacionados aos recursos que devem ser isolados para um tenant em particular. Embora os testes empregados possuem várias limitações, tornou-se possível identificar os problemas relacionados ao isolamento de sessões por tenant que podem resultar em sérios efeitos colaterais. Com relação ao desafio de *Falta de padronização (C5)*, notou-se a dificuldade em encontrar ferramentas que atendem aos desafios que a multi-tenancy impõe para a atividade de teste. A maioria das aplicações em repositórios públicos como o GitHub não seguem um modelo de referência, um fator que poderia ajudar na priorização de funcionalidades a serem verificadas

5.3.5 Ameaças à validade

Ameaças externas referem-se à generalização dos resultados a partir do experimento para outros contexto, como aplicações reais. Considerando o contexto sob análise, um estudo controlado torna-se necessário de modo que seus resultados possam ser mapeados e descritos de forma satisfatória. Em geral, esse modelo de avaliação em ambiente industrial é uma tarefa difícil na maioria dos casos, raramente existe o controle da aplicação e recursos disponíveis da mesma forma que em ambiente acadêmico. Nesse segundo estudo, duas aplicações exemplos (*toy examples*) foram submetidas ao teste paralelo e automatizado. Essas aplicações possuem as principais features da multi-tenancy ao incorporar frameworks que são utilizados na indústria, além de artefatos essenciais para suas operações. Elas possuem diferentes tipos de modelo de implementação que contribuem para resultados mais compreensivos e confiáveis.

Ameaças internas são influências que podem afetar variáveis independentes com respeito à mudanças, nas quais o pesquisador não possui conhecimento ou controle. Neste estudo existem alguns aspectos que podem influenciar os resultados, como por exemplo somente uma das aplicações foi desenvolvida durante o doutorado, consequentemente existe aspectos internos desconhecidos que podem tornar difícil a rastreabilidade de execução, embora o código esteja disponível. Para mitigar essa ameaça, as aplicações admitidas foram utilizadas em ambientes controlados com o mesmo cenário de avaliação considerando as mesmas variáveis, repetições e recursos computacionais disponíveis.

Validade de construção refere-se ao relacionamento entre a teoria e a observação de um estudo experimental. Em outras palavras, essa validade está relacionada ao fato que o experimento precisa refletir a realidade da influência dos tratamentos sobre os resultados observados. Acredita-

se que a experiência em projetar os casos de teste poderia ser uma ameaça aos resultados. Para lidar com essa ameaça, um especialista em teste foi selecionado para projetar os casos de teste e ele não teve acesso ao código de ambas aplicações.

Validade de conclusão compreende as características que afetam a capacidade de inferir conclusões corretas sobre os relacionamentos entre os tratamentos e os resultados do experimento. Testes estatísticos são ameaças comuns para estudos experimentais pois quando são realizados de forma inadequada podem reproduzir resultados incorretos. Para contornar essa ameaça foram empregados testes como base no número das amostras do experimento, o tipo de dados obtido e a normalidade dos dados. Com relação ao risco de favorecer uma certa hipótese durante a análise dos dados, o estudo foi exploratório e não houve interesse nesse favorecimento.

5.4 Considerações finais

Os avanços na computação em nuvem e o crescimento da adoção de sistemas SaaS em vários segmentos de negócio requerem estratégias que contribuem com a alta qualidade desse tipo de aplicações. O objetivo do primeiro estudo foi verificar a relação entre o número de versões e a quantidade defeitos ao empregar o teste paralelo, visto que dessa forma estruturas compartilhadas da aplicação poderiam ser exercitadas ao simular a execução simultânea das versões. No segundo estudo o objetivo principal foi investigar o teste paralelo e automatizado levando em conta as diferenças entre as arquiteturas de dados e implementações, incluindo os impactos para a atividade de teste. Para isso, dois estudos de caso foram realizados adotando aplicações diferentes. Adotou-se uma estratégia para apoiar os testes paralelos que evitasse a interferência do escalonamento de recursos com o uso de um TaaS. As ferramentas de teste, dados coletados dos experimentos e detalhes sobre o teste automatizado estão disponíveis para o primeiro¹³ e para o segundo¹⁴ estudo.

Os resultados demonstraram que essa abordagem de teste é útil para identificar problemas no gerenciamento de certos recursos compartilhados, principalmente aqueles relacionados com a definição do identificador do tenant corrente e seus usos subsequentes. Independente da arquitetura de dados adotada, problemas nessas funcionalidades podem ocorrer devido à falta de isolamento adequado das sessões para as versões. Defeitos identificados em ambas aplicações foram classificados e destacados no segundo estudo, destacando as similaridades com aplicações convencionais e concorrentes. Os resultados para *QP1* no segundo estudo seguiu a mesma tendência que o primeiro, ao variar o número de versões nos testes paralelos, as mudanças no meio do fluxo de execução para os tenants também aumentou. Enquanto para *QP2* definida no segundo estudo, esses problemas se refletiram em ambas aplicações, independente da arquitetura de dados e modelo de implementação. Com base nesses resultados, testes paralelos demonstraram-

¹³ <https://goo.gl/ESNCGD>

¹⁴ <https://tinyurl.com/y5k5wuxl>

se como uma alternativa válida para identificação desse tipo de defeito que pode ser difícil de ser reproduzido em um cenário multi-tenant.

A técnica de teste estrutural pode ser empregada de forma complementar para identificar a natureza dos defeitos. Nesse sentido, a evolução desses estudos preliminares contribuíram para a definição de um novo modelo para o teste estrutural de sistemas SaaS multi-tenants com base em grafos multi-camada para capturar informações sobre os fluxos de dados e controle das aplicações SaaS. Com base nesse modelo, uma família de critérios de teste foi elaborada para exercitar as requisições e redirecionamentos após as definições da variável de identificação para os tenants. No Capítulo 6 apresentam-se tais contribuições.

TESTE ESTRUTURAL APLICADO A SISTEMAS MULTI-TENANTS

6.1 Considerações iniciais

A necessidade de desenvolver software com alta qualidade exige constante aperfeiçoamento de metodologias e práticas de engenharia de software. Esse esforço se torna ainda mais desafiador quando o modelo de aplicações em questão possui certas particularidades e as técnicas convencionais não as satisfazem totalmente. No caso de sistemas SaaS que adotam multi-tenancy, a alta capacidade de customização e a dinamicidade em que as versões específicas são geradas em tempo de execução exigem um esforço adicional, uma vez que uma única instância precisa atender simultaneamente as especificações de diversas organizações e isso pode contribuir para a possibilidade de defeitos críticos. Embora as operações para cada versão sejam assíncronas, existem módulos compartilhados que podem levar ao indeterminismo, um desafio conhecido no teste de programas concorrentes e que também aumenta a complexidade da atividade de teste em aplicações desenvolvidas para nuvem. Assim como ocorre para outros contextos, técnicas de teste empregadas em aplicações convencionais são refinadas e novos modelos de teste são propostos para atender tais desafios.

Este capítulo apresenta as principais contribuições originadas da tese para o teste de sistemas SaaS multi-tenant: um modelo de teste estrutural e uma família de critérios de cobertura aplicáveis ao domínio de aplicações tratado neste trabalho. Esse modelo inclui elementos importantes para o cenário multi-tenant compreendendo: (i) análise estática do programa em teste evidenciando elementos estruturais e (ii) o levantamento de elementos requeridos de acordo com critérios de testes adaptados para tais sistemas. Informações sobre as operações entre as versões designadas aos tenants e as features genéricas são obtidas a partir de uma versão instrumentada da aplicação sob teste. Os critérios foram definidos para explorar os fluxos de controle e de dados de tais aplicações, considerando aspectos sequenciais e paralelos. A

principal contribuição dessa família de critérios é fornecer uma medida de cobertura que possa ser empregada para avaliar o progresso da atividade de teste. Isso é importante para avaliar a qualidade dos casos de teste, bem como indicar que o software foi suficientemente testado visto que ocorrem evoluções frequentes e não se tem até o momento estratégias de teste que consideram a estrutura e as particularidades das aplicações desenvolvidas para nuvem. Para contornar essa dificuldade, uma abordagem de teste também é proposta para a identificação de defeitos relacionados aos desvios nos fluxos de execução.

6.2 Modelo de teste e critérios estruturais

Modelos de teste são importantes para representar o comportamento das aplicações em tempo de execução ou permitir uma visão estática para descrever o fluxo de controle entre seus comandos. Por exemplo, no contexto de programas concorrentes as contribuições apontadas por [Yang, Souter e Pollock \(1998\)](#) para permitir a representação dos aspectos de comunicação entre processos em um programa concorrente foram estendidas por [Vergílio, Souza e Souza \(2005\)](#) para permitir a aplicação de critérios de teste para programas paralelos em ambientes de passagem de mensagem. Neste estudo inicial é proposto a construção de um Grafo de Fluxo de Controle (GFC) para cada processo paralelo, da mesma forma que é feito para programas sequenciais. Com base nesses grafos, um *PCFG* (*Parallel Control Flow Graph*, ou Grafo de Fluxo de Controle Paralelo) é construído, composto por arestas de sincronização. Essas arestas representam a criação dos processos paralelos e a comunicação entre esses processos.

Com base nesses fundamentos utilizados para programação concorrente e a complexidade da arquitetura multi-tenant, foi elaborada uma notação levando em conta os *Multi-layer Graphs*, ou grafos multicamada ([OSELIO; KULESZA; HERO, 2013](#)), gerando o grafo chamado de *PCFG_{MT}* (*Parallel Control Flow Graph for multi-tenant SaaS applications*) ou grafo de fluxo de controle paralelo para sistemas SaaS multi-tenants. Essa é uma das principais contribuições desta pesquisa de doutorado para apoiar a representação dos elementos e dos fluxos de execução para esse tipo de aplicação.

Os grafos multicamadas são empregados em diversas áreas do conhecimento, na computação por exemplo é utilizada na neurociência computacional ([DOMENICO, 2017](#)). Um grafo multicamada é basicamente um conjunto de grafos, de modo que cada um pode representar diferentes estruturas de conexão. Nesse modelo dinâmico podem existir arestas com diferentes significados e diferentes instantes de tempo ou outras características distintas. Geralmente os grafos não requerem o mesmo conjunto de nós (ou vértices), mas pode-se assumir que sempre terão o mesmo conjunto, onde esse conjunto refere-se à união dos conjuntos dos nós de todos os grafos. Assim, um grafo multicamada é um grupo de grafos, em que cada grafo pode ter tipos diferentes ou o mesmo tipo de aresta, de modo que de acordo uma determinada perspectiva pode-se visualizar um conjunto específico de arestas. Cada grafo pertencente ao grafo multicamada é

chamando de grafo componente.

Definição Um grafo convencional pode ser representado por $V=(N, E)$ em que N é um conjunto de nós e E um conjunto de arestas representadas por tuplas de nós $(u, v) \in N$. Da mesma forma, em um grafo multicamada tem-se N como um conjunto de nós, sendo que $E_1, E_2, E_3, \dots, E_k$ são conjuntos de arestas. Um grafo multicamada T é o conjunto de grafos $T = \{T_1(N, E_1), T_2(N, E_2), \dots, T_k(N, E_k)\}$. Os grafos T_i são chamados grafos componentes. O número de grafos componentes k é chamado de multiplicidade do grafo. Para acomodar a presença de mais de um tenant, os grafos componentes são utilizados para representar os fluxos de execução de cada versão correspondente. Assim, o $PCFG_{MT}$ em termos de grafo multicamada pode ser definido por uma tripla, em que $T = (N, E, D)$, onde N refere-se ao conjunto de nós de um sistema SaaS em termos de um GFC, D compreende um conjunto de camadas ou grafos componentes $D = (t_0, t_1, t_2, \dots, t_k)$ e cada elemento de E contém uma tupla (u^{t_0}, v^{t_k}) com $u^{t_0}, v^{t_k} \in N$.

O objetivo de definir cada aresta com a informação d em um $PCFG_{MT}$ deve-se ao fato de tornar mais claro as análises de cobertura e sua rastreabilidade, uma vez que d indica a versão da aplicação de um tenant específico. Nesse modelo de fluxo de dados foram adicionadas informações sobre a definição e uso da variável de identificação do tenant e seus recursos, portanto seu valor deve permanecer sem alterações durante todos usos subsequentes em uma dada versão da aplicação, ou seja, em uma determinada camada d . Entretanto, novas definições nesta variável são aplicadas para que versões diferentes sejam disponibilizadas a outros tenants. As (re)definições dessa variável ocorrem em consequência de requisições e redirecionamentos, tais informações também precisam ser incluídas aos grafos. A ideia principal desse modelo é representar uma aplicação multi-tenant em termos de GFCs e nos moldes de um grafo multicamada e assim capturar as informações necessárias para testar o programa considerando suas particularidades.

A principal motivação de elaborar uma notação, criar um modelo de teste e uma família de critérios foi permitir que critérios de teste sequenciais pudessem ser aplicados em sistemas SaaS multi-tenants. O modelo de teste possui as seguintes considerações:

1. a aplicação SaaS $MT = \{t_1, t_2, t_3, \dots, t_n\}$ é um conjunto de n versões designadas a tenants distintos que são executadas de forma paralela, compartilham recursos mas são isoladas virtualmente;
2. o número n de versões é fixo e conhecido em tempo de compilação. Assim, para construir o modelo é fundamental conhecer o número de tenants que compartilham a mesma instância do sistema SaaS;
3. as versões não são criadas no momento da inicialização da instância. Isso significa que a medida que ocorrem novas requisições de um dado usuário pertencente a um tenant, tem-se a compilação dinâmica que gera a versão correspondente a partir das funciona-

lidades e meta-dados específicos. Desse modo, várias versões podem ser compiladas simultaneamente;

4. as versões compartilham um módulo de interceptação de requisições, onde ocorrem as (re)definições do identificador único de cada tenant. Não se espera comunicação do tipo inter-processos, ou seja, comunicação entre as versões. A interceptação é acionada em decorrência de requisições e redirecionamentos, portanto os nós com essa finalidade precisam ser identificados no grafo.

Adaptações foram necessárias para representar, dentre outras coisas, a comunicação entre os módulos direcionados à definição da variável de identificação e seus subsequentes usos em domínios de execução diferentes, isto é, versões diferentes que são executadas de forma paralela. Além disso, o modelo desenvolvido concentra-se na aplicação dos testes para validar não somente uma versão de forma isolada como também o sistema SaaS como um todo em termos de cobertura arquitetural.

Com a construção de um $PCFG_{MT}$ as seguintes informações podem ser obtidas:

- N : conjunto de nós do $PCFG_{MT}$;
- E : conjunto de arestas do $PCFG_{MT}$;
- D : conjunto de camadas do $PCFG_{MT}$;
- N_d : conjunto de nós n_i^d vinculados a uma versão da aplicação em relação à d ;
- N_{post} : conjunto de nós n_i^p que colaboram para a definição do identificador do tenant;
- N_{red} : conjunto de nós que colaboram com o envio/tratamento dos redirecionamentos que incluem o identificador do tenant;
- N_{req} : conjunto de nós que colaboram com o envio/tratamento das requisições que incluem o identificador do tenant;
- E_{red} : conjunto de arestas de redirecionamento que envolvem os nós de N_{red} ;
- E_{req} : conjunto de arestas de requisições que envolvem os nós de N_{req}

A partir de N , quatro subconjuntos são definidos: N_d , N_{post} , N_{red} e N_{req} . De modo que $N_{post} \neq N_d$, $N_{red} = \{(n_1^r, n_2^r, \dots, n_k^r) | n_i^r \in N_d \vee N_{post}\}$ e $N_{req} = \{(n_1^q, n_2^q, \dots, n_k^q) | n_i^q \in N_d \vee N_{post}\}$, ou seja, os nós de N_{red} e de N_{req} são formados por nós de N_{post} ou N_d , mais não são interseções pois N_d e N_{post} não possuem nós em comum. Os nós de N_d fazem algum tipo de uso ou colaboram para o uso do identificador do tenant, portanto estão associados a uma camada d que especifica o fluxo de execução para uma versão da aplicação destinada a um dos tenants. Os nós de N_{post}

colaboram para definição do identificador do tenant e conforme discutido no Capítulo 3, fazem parte de um mecanismo de interceptação, ou seja, é possível rastrear em um sistema SaaS onde essa operação está sendo realizada. Os nós de N_{red} podem ser identificados nos métodos que realizam um redirecionamento para a aplicação incluindo o valor do identificador. Enquanto que os nós de N_{req} estão associados aos métodos que realizam o mapeamento entre uma requisição e uma página web que deve ser fornecida ao usuário e para isso, o valor do identificador do tenant também precisa fazer parte da requisição. Geralmente, os métodos ligados aos nós de N_{red} e N_{req} pertencem às classes da camada *Controller* (em um modelo MVC) e do mecanismo de interceptação.

Vale ressaltar que: (i) N inclui todos os nós compartilhados entre os tenants, ou seja, nós que essencialmente contribuem para tornar a aplicação multi-tenant e que (ii) as camadas e N_d servem para apoiar a separação do contexto de execução para uma versão voltada aos requisitos de um tenant específico. Portanto, N_d pode ser visto como um conjunto de nós específicos em relação a uma determinada versão. A título de ilustração a Figura 28 mostra um $PCFG_{MT}$ para uma aplicação SaaS hipotética chamada “App”. Observe que somente as estruturas relacionadas com a definição e usos do identificador do tenant (*ident*) devem ser consideradas, além dos elementos envolvidos com o tratamento de requisições bem como os redirecionamentos que podem levar a novas definições dessa variável. O modelo considera que outros testes já foram realizados considerando as unidades (métodos) da aplicação.

Nesse modelo é importante destacar duas partes principais: o grafo que envolve a definição do identificador do tenant (t_0) e as camadas ($t_1, \dots, t_5$) que incluem as features de cada tenant, as quais geralmente estendem funcionalidades genéricas fornecidas pelo provedor do sistema SaaS. Para o caso em específico do App, as versões são idênticas para tornar mais claro a compreensão da modelagem, por isso existem camadas sobrepostas. Em situações reais em que ocorrem extensões de funcionalidades e mudanças para alinhar as regras de negócio de um tenant é provável a existência de outros grafos e estes não serão os mesmos em outros tenants. Mesmo nessa situação ainda haverá sobreposição, uma vez que parte das operações continuarão sendo compartilhadas entre os tenants. A camada sob o ponto de vista do teste serve para mapear os grafos que fazem uso do identificador *ident* de modo que para outra camada seu valor será diferente. Em outras palavras, em um fluxo de execução de uma versão relativa a “ t_w ”, o identificador deve se manter “ w ” mesmo que ocorram múltiplas redefinições dessa variável provenientes de outros tenants. Esse cenário demonstra que sem um modelo de teste, torna-se difícil analisar os resultados considerando somente a perspectiva do usuário, ou seja, a estrutura interna da aplicação precisa ser mapeada para apoiar a definição de casos de teste relevantes e cenários específicos de execução com a finalidade de exercitar determinados caminhos e explorar a existência de vários tenants.

O grafo t_0 compreende um conjunto de nós $n_i^p \in N_{post}$, onde $p = 0$, i refere-se ao número dos nós de t_0 . Assim, $N_{post} = \{n_1^0, \dots, n_8^0\}$. Adiante tem-se um conjunto de grafos componentes

N_d , onde $d \in D$ e $D = \{t_1, \dots, t_5\}$. Para $d=t_1$, $N_1 = \{n_1^1, \dots, n_{10}^1\}$ de modo que para todo $n_i^j \in N_d$, j representa a camada e i , o número do nó. Assim, $N_{red} = \{n_1^0, n_{10}^1\}$ e $N_{req} = \{n_8^0, n_1^1, n_7^1\}$. Com relação às arestas relativas aos redirecionamentos e requisições em relação a $d=t_1$, tem-se $E_{req} = \{(n_8^0, n_1^1), (n_8^0, n_7^1)\}$ e $E_{red} = \{(n_{10}^1, n_1^0)\}$, assim por diante. Por exemplo, para $d=t_2$: $N_2 = \{n_1^2, \dots, n_{10}^2\}$, $N_{red} = \{n_1^0, n_{10}^2\}$ e $N_{req} = \{n_8^0, n_1^2, n_7^2\}$, $E_{req} = \{(n_8^0, n_1^2), (n_8^0, n_7^2)\}$ e $E_{red} = \{(n_{10}^2, n_1^0)\}$. Como pode ser observado as requisições ou redirecionamentos passam por N_{post} e podem levar a definição do identificador do tenant (nós n_3^0 ou n_7^0). Posteriormente, o fluxo de execução é direcionado aos demais nós de N_d , nos quais ocorrem os usos dessa variável.

Figura 28 – $PCFG_{MT}$ para a aplicação App.

Um caminho π no $PCFG_{MT}$ é dado por uma sequência de nós $\pi = (n_i^g, \dots, n_j^h)$ e $(n_i^g, n_j^h) \in N$. O uso de uma variável *ident*, definida para identificação do tenant, ocorre quando *ident* é referenciada em uma expressão. O símbolo $\prec$ é utilizado para indicar a ordem de execução entre dois nós diferentes. Portanto $n_i^g \prec n_j^h$ indica que n_i^g é executado antes de n_j^h , isso é importante para garantir a execução da *def-uso*. Um caminho completo π inclui (ou cobre) um nó ou uma aresta $(n_i^g, n_k^g) \in N_d$, se ele inclui um subcaminho um nó ou aresta dada. Um caminho π que envolve um **redirecionamento** (*red*) inclui (ou cobre) uma aresta (n_i^d, n_j^p) , onde $n_i^d \in N_d$ e $n_j^p \in N_{post}$, se os sub-caminhos π_1 e π_2 de π , executados respectivamente em d (nós $\in N_d$) e p (nós $\in N_{post}$), incluírem os nós n_i^d e n_j^p e se $n_i^d \prec n_j^p$, ou seja, deve garantir que o redirecionamento foi executado de um nó n_i^d para n_j^p partindo de um contexto de execução dado pela camada d . Similarmente, um caminho π que envolve o tratamento de uma **requisição** (*req*) inclui (ou cobre) uma aresta (n_j^p, n_i^d) , se os sub-caminhos π_1 e π_2 de π , executados respectivamente em p e d , incluírem os nós n_j^p e n_i^d e se $n_j^p \prec n_i^d$, ou seja, deve garantir que a requisição foi executada de um nó n_j^p para n_i^d e que após execução de nós que colaboram com a definição de *ident* parte-se para o fluxo de execução para a versão da aplicação correspondente, dado pela camada d .

No contexto dos sistemas SaaS existem diversas unidades nas quais ocorrem definições e usos subsequentes de muitas variáveis, assim como em uma aplicação convencional. Portanto, várias técnicas de teste podem ser aplicadas. Entretanto, o modelo de teste proposto considera

somente a variável de identificação *ident*, dada sua importância e criticidade dos defeitos vinculados a sua utilização no contexto de sistemas SaaS multi-tenants. Assim são caracterizados três tipos de uso para *ident*:

1. *uso computacional (c-uso)*: ocorre em uma computação, relacionada a um nó n_i^d do $PCFG_{MT}$;
2. *uso predicativo (p-uso)*: ocorre em uma condição (predicado), associada a comandos de fluxo de controle, em arestas (n_i^d, n_j^d) ;
3. *uso redirecional (r-uso)*: ocorre em comandos de redirecionamento e requisições, ou seja, em arestas E_{red} e E_{req} .

Da mesma forma que ocorre com os demais critérios de teste estruturais, um caminho $\pi=(n_1^p, n_2^p, n_j^p, ..., n_1^d, n_k^d)$ é livre de definição em relação a *ident* definida no nó n_1^p e usada no nó n_k^d ou aresta (n_1^p, n_k^d) , se *ident* $\in def(n_1^p)$ e *ident* $\notin def(n_i^p)$, para $i = 2...j$. Partindo dessas definições é possível caracterizar as associações que devem ser requeridas pelos critérios estruturais aplicados a sistemas SaaS multi-tenants. Três tipos de associações podem ser derivadas:

- *associação c-uso* - dada pela tupla (n_j^p, n_i^d) em que *ident* $\in def(n_j^p)$, e n_i^d possui um c-uso de *ident* e existe um caminho livre de definição em relação a *ident* de n_j^p para n_i^d ;
- *associação p-uso* - dada pela tupla $(n_j^p, (n_i^d, n_k^d))$ em que *ident* $\in def(n_j^p)$, e (n_i^d, n_k^d) possui um p-uso de *ident* e existe um caminho livre de definição em relação a *ident* de n_j^p para (n_i^d, n_k^d) ;
- *associação r-uso* - dada pela tupla $(n_j^p, (n_k^p, n_i^d))$, onde n_j^p e $n_k^p \in N_{post}$, *ident* $\in def(n_j^p)$, e (n_k^p, n_i^d) possui um r-uso de *ident* e existe um caminho livre de definição em relação a *ident* de n_j^p para (n_k^p, n_i^d) ;

Essas associações permitem verificar erros desde as interceptações das requisições e dos redirecionamentos até os usos subsequentes da variável *ident*. Desse modo, pode-se identificar desvios nos fluxos de execução que levaram ao uso de um valor indevido para *ident* em relação a *d*. Na Tabela 14 descrevem-se os conjuntos obtidos, os quais são utilizados para derivação dos requisitos de teste e ilustrar essas definições.

Essas informações são utilizadas para apoiar a definição de critérios de teste, os quais são descritos na próxima seção.

6.2.1 Critérios de cobertura

Critérios baseados em fluxo de controle

Tabela 14 – Informações obtidas pelo modelo de teste para a aplicação *App*.

$D = \{t_1, t_2, t_3, t_4, t_5\}$
$N = N_{post} \cup N_d$
$N_{post} = \{1^0, 2^0, 3^0, 4^0, 5^0, 6^0, 7^0, 8^0\}$
$N_d = N_1 \cup N_2 \cup N_3 \cup N_4 \cup N_5$
$N_1 = \{1^1, 2^1, 3^1, 4^1, 5^1, 6^1, 7^1, 8^1, 9^1, 10^1\}$
$N_2 = \{1^2, 2^2, 3^2, 4^2, 5^2, 6^2, 7^2, 8^2, 9^2, 10^2\}$
$N_3 = \{1^3, 2^3, 3^3, 4^3, 5^3, 6^3, 7^3, 8^3, 9^3, 10^3\}$
$N_4 = \{1^4, 2^4, 3^4, 4^4, 5^4, 6^4, 7^4, 8^4, 9^4, 10^4\}$
$N_5 = \{1^5, 2^5, 3^5, 4^5, 5^5, 6^5, 7^5, 8^5, 9^5, 10^5\}$
$N_{red} = \{1^0, 10^1, 10^2, 10^3, 10^4, 10^5\}$
$N_{req} = \{8^0, 1^1, 7^1, 1^2, 7^2, 1^3, 7^3, 1^4, 7^4, 1^5, 7^5\}$
$E = E_d \cup E_{post} \mid E_d = E_1 \cup E_1 \cup E_1 \cup E_4 \cup E_5 \ (d = t_1, ..., t_5)$
$E_1 = \{(1^1, 2^1), (2^1, 3^1), (2^1, 4^1), (3^1, 4^1), (4^1, 5^1), (4^1, 6^1), (5^1, 6^1), (7^1, 8^1), (8^1, 9^1), (9^1, 8^1), (8^1, 10^1)\}$
$E_2 = \{(1^2, 2^2), (2^2, 3^2), (2^2, 4^2), (3^2, 4^2), (4^2, 5^2), (4^2, 6^2), (5^2, 6^2), (7^2, 8^2), (8^2, 9^2), (9^2, 8^2), (8^2, 10^2)\}$
$E_3 = \{(1^3, 2^3), (2^3, 3^3), (2^3, 4^3), (3^3, 4^3), (4^3, 5^3), (4^3, 6^3), (5^3, 6^3), (7^3, 8^3), (8^3, 9^3), (9^3, 8^3), (8^3, 10^3)\}$
$E_4 = \{(1^4, 2^4), (2^4, 3^4), (2^4, 4^4), (3^4, 4^4), (4^4, 5^4), (4^4, 6^4), (5^4, 6^4), (7^4, 8^4), (8^4, 9^4), (9^4, 8^4), (8^4, 10^4)\}$
$E_5 = \{(1^5, 2^5), (2^5, 3^5), (2^5, 4^5), (3^5, 4^5), (4^5, 5^5), (4^5, 6^5), (5^5, 6^5), (7^5, 8^5), (8^5, 9^5), (9^5, 8^5), (8^5, 10^5)\}$
$E_{post} = \{(1^0, 2^0), (1^0, 8^0), (2^0, 3^0), (2^0, 5^0), (3^0, 5^0), (3^0, 4^0), (4^0, 5^0), (5^0, 6^0), (6^0, 7^0), (6^0, 8^0), (7^0, 8^0)\}$
$E_{red} = \{(10^1, 1^0), (10^2, 1^0), (10^3, 1^0), (10^4, 1^0), (10^5, 1^0)\}$
$E_{req} = \{(8^0, 1^1), (8^0, 7^1), (8^0, 1^2), (8^0, 7^2), (8^0, 1^3), (8^0, 7^3), (8^0, 1^4), (8^0, 7^4), (8^0, 1^5), (8^0, 7^5)\}$

Cada GFC pode ser verificado individualmente utilizando os critérios *Todos-Nós* e *Todas-Arestas*. Entretanto, o objetivo é testar as associações presentes em um $PCFG_{MT}$ para uma dada aplicação multi-tenant. Desse modo, os seguintes critérios são definidos:

- *Todos-Nós-req*: requer que todos os nós $n \in N_{req}$ sejam exercitados pelo menos uma vez pelo conjunto de casos de teste;
- *Todos-Nós-red*: requer que todos os nós $n \in N_{red}$ sejam exercitados pelo menos uma vez pelo conjunto de casos de teste;
- *Todos-Nós*: requer que todos os nós do conjunto N sejam exercitados pelo menos uma vez pelo conjunto de casos de teste;
- *Todas-Arestas-req*: requer que as arestas do conjunto N_{req} sejam exercitadas pelo menos uma vez pelo conjunto de casos de teste;
- *Todas-Arestas-red*: requer que as arestas do conjunto N_{red} sejam exercitadas pelo menos uma vez pelo conjunto de casos de teste.

Os critérios *Todos-Nós-req* e *Todos-Nós-red* incluem implicitamente a execução dos nós que recebem o valor de *ident* e devem dar continuidade ao fluxo de execução em uma camada d correspondente a esse valor e assim alcançar os nós pertencentes à N_{req} ou N_{red} , respectivamente. De outro lado, os critérios *Todas-Arestas-red* e *Todas-Arestas-req* possuem o enfoque na associação entre nós de N_d e N_{post} . Ainda é possível assim como em programas convencionais, definir os critérios *Todos os Caminhos* do GFC (para $d = t_1, t_2, ..., t_n$) e *Todos os*

Caminhos do $PCFG_{MT}$, equivalente a executar todas as combinações dos possíveis dos caminhos nos GFC de cada camada d com os caminhos nos GFC que envolvem os nós do conjunto N_{post} , mas esses critérios podem ser considerados impraticáveis.

Crítérios baseados em fluxo de dados

- *Todas-Def-ident*: requer que para cada nó n_i^p em que $ident \in def(n_i^p)$ uma associação definição-uso (c-uso, p-uso ou r-uso) de $ident$ seja exercitada pelo conjunto de casos de teste;
- *Todos-Usos-ident*: requer que todas as associações de usos de $ident$ sejam exercitadas pelo conjunto de casos de teste.

O critério *Todas-Def-ident* visa apoiar a identificação de defeitos em relação a $ident$, de modo que após uma definição e considerando que o fluxo de execução seja livre de definição, o valor da variável se manteve em relação a camada d . Em um cenário multi-tenant são esperadas outras definições simultâneas para essa variável, mas em relação a uma versão específica da aplicação esse valor deve se manter constante, caso contrário um defeito poderá ser identificado. O critério *Todos-Usos-ident* tem o objetivo de apoiar a verificação dos usos de $ident$ em uma camada d , ou seja, se os usos desta variável se mantiveram corretos em uma versão da aplicação correspondente a d . Com base nesse critério devem ser projetados casos de teste que explorem tais usos de $ident$.

A Tabela 15 apresenta exemplos de elementos requeridos para esses critérios, considerando o sistema *App*. Da mesma forma que ocorre em programas sequenciais, a não-executabilidade é inerente a esses critérios de teste. Para ilustrar, considere o critério *Todos-Usos-ident*. A associação $(1^0, (8^0, 1^2))$ é não executável, pois é necessário que ocorra a definição de $ident$ para que a aplicação consiga responder à requisição e gere a versão correspondente. Em outras palavras, uma versão é gerada quando uma requisição ou um redirecionamento é interceptado para ocorrer uma definição de $ident$. Como uma consequência, isso levará ao fluxo de execução correspondente, definido por uma camada d que neste caso é t_2 . Ao analisar essa associação de forma isolada de caminhos anteriores, tem-se uma não-executabilidade. Essa associação depende de uma definição anterior de $ident$ para se tornar executável, isto é, que exista pelo menos um caso de teste capaz de exercitar um caminho envolvendo esta associação.

6.3 Abordagem de teste proposta

O teste paralelo de aplicações web consiste em executar múltiplos casos de teste em diversas combinações. A vantagem dessa estratégia é permitir que testadores e desenvolvedores empreguem recursos para problemas relacionados com a compatibilidade *cross-platform*. Entretanto, para o contexto desta pesquisa a razão de conduzir testes paralelos é simular ações e

Tabela 15 – Elementos requeridos pelos critérios de teste em relação ao sistema App.

Critério	Elementos requeridos
<i>Todos-Nós-req</i>	$8^0, 1^1, 7^1, 1^2, 7^2, 1^3, 7^3, 1^4, 7^4, 1^5, 7^5$
<i>Todos-Nós-red</i>	$1^0, 10^1, 10^2, 10^3, 10^4, 10^5$
<i>Todos-Nós</i>	$1^0, \dots, 8^0, 1^1, \dots, 10^1, 1^2, \dots, 10^2, 1^3, \dots, 10^3, 1^4, \dots, 10^4, 1^5, \dots, 10^5$
<i>Todas-Arestas-req</i>	$(8^0, 1^1), (8^0, 7^1), (8^0, 1^2), (8^0, 7^2), (8^0, 1^3), (8^0, 7^3), (8^0, 1^4), (8^0, 7^4), (8^0, 1^5), (8^0, 7^5)$
<i>Todas-Arestas-red</i>	$(10^1, 1^0), (10^2, 1^0), (10^3, 1^0), (10^4, 1^0), (10^5, 1^0)$
<i>Todas-Def-ident</i>	$(3^0, 7^0), (3^0, 5^0, 6^0)$
<i>Todos-Usos-ident</i>	$(1^1, 2^1), (3^1, 4^1), (1^0, (8^0, 1^1)), \dots, (1^0, (8^0, 1^5)), (7^1, (8^1, 9^1)), (7^1, (8^1, 10^1)), (8^0, (7^1, 8^1)), (8^0, (1^1, 2^1)), (1^2, 2^2), (3^2, 4^2), (7^2, (8^2, 9^2)), (7^2, (8^2, 10^2)), (8^0, (7^2, 8^2)), (8^0, (1^2, 2^2)), (1^3, 2^3), \dots, (8^0, (1^3, 2^3)), (1^4, 2^4), \dots, (8^0, (1^4, 2^4)), (1^5, 2^5), \dots, (8^0, (1^5, 2^5)), (10^1, (1^0, 2^0)), (10^2, (1^0, 2^0)), (10^3, (1^0, 2^0)), (10^4, (1^0, 2^0)), (10^5, (1^0, 2^0))$

requisições de diferentes usuários de cada tenant para uma única instância da aplicação, ou seja, não identificar somente defeitos sob uma perspectiva de teste para programas sequenciais mas além disso, identificar defeitos difíceis de serem reproduzidos ao considerar o cenário real de execução com múltiplas versões. A seguir apresentam-se as etapas definidas para uma abordagem de teste para sistemas multi-tenants e uma visão arquitetural de uma estratégia de teste.

Como uma etapa inicial, admite-se a **1. Identificação de cenários de utilização** a partir da especificação do sistema (KANER, 2003). Esses cenários são úteis para identificação das rotas esperadas pela aplicação multi-tenant, as chamadas *tenant view-routing*. Geralmente são mapeadas com algum método que deve responder a uma requisição. De posse disso, parte-se para a etapa de **2. Rastreamento das definições e usos da variável de identificação dos tenants** e seus recursos. Para isso, deve-se procurar métodos que colaboram com a definição da variável de identificação dos tenants, bem como suas utilizações. Essa variável é vital para qualquer sistema multi-tenant, portanto defeitos em decorrência de definições e usos inapropriados podem levar a sérios problemas, por isso o enfoque desta abordagem é com relação a essa variável. Com base nessas informações deve-se **3. Aplicar o modelo de teste PCFG_{MT}** para representar a dinamicidade das versões disponibilizadas pela solução multi-tenant sob teste, conforme apresentado anteriormente. Com esse modelo **4. Aplicam-se a família de critérios de cobertura propostos** para cobrir as requisições e redirecionamentos que levam às sucessivas definições na variável de identificação dos tenants. Essa etapa envolve o planejamento dos casos de teste, os quais podem ser gerados para cobrir os elementos requeridos por esses critérios e outros como os provenientes da técnica funcional podem ser empregados de forma complementar para obter uma alta cobertura no código. Posteriormente, parte-se para a **5. Execução dos testes automatizados de forma paralela** para as versões designadas aos tenants e por fim, **6. Coletam-se os resultados dessa execução**. Essa última etapa serve para fornecer indicativos para geração de novos casos de teste e a identificação de defeitos em versões isoladas que podem afetar os demais tenants.

A Figura 29 apresenta uma visão arquitetural de alto nível de uma estratégia de teste cujos

princípios são provenientes dos experimentos discutidos no Capítulo 5 mas que foi refinada com a definição do modelo de teste proposto. Essa estratégia envolve um conjunto de técnicas visando manipular a definição de um *workflow* antes de executar os casos de teste de forma paralela. Os principais componentes e seus relacionamentos podem ser divididos em três partes: *Multi-tenant Configuration Dashboard*, *Test Automation API* e *Results*. Somente dois componentes que estão no ***Multi-tenant Configuration Dashboard*** precisam ser fornecidos manualmente pelo testador. Classes com os métodos e elementos de teste, incluindo casos de teste adequados precisam ser projetados para verificar as diferentes versões das aplicações. No *Data for instantiation*, (i) informações sobre os identificadores dos tenants e (ii) o ambiente de teste são requeridos. O primeiro é usado para definir quantas threads serão empregadas para execução dos testes, uma vez que cada thread simula as ações do usuário para uma aplicação particular. No segundo, o testador especifica se os testes devem ser realizados em ambiente local ou remoto, por exemplo, utilizando um TaaS com suporte à automação de testes como o Selenium.

Figura 29 – Uma visão arquitetural da estratégia de teste.

Parameterized test infrastructure foi desenvolvido com extensões no JUnit, como o *Parameterized runner*¹ e Selenium API de modo que os testes podem ser realizados de forma paralela para qualquer número de versões. O ***Test Automation API*** permite executar automaticamente os casos de teste. Para isso, as threads são criadas com métodos de teste e elementos de dados de teste considerando o número de tenants. Entretanto, não se pode garantir que todas as tarefas submetidas serão executadas ao mesmo intervalo de tempo ou antes de algum evento, pois isso depende do escalonamento das threads e isso não pode ser controlado (levando em conta seu

¹ <https://junit.org/junit4/javadoc/4.12/org/junit/runners/Parameterized.html>

comportamento não determinístico). Mesmo que todas as tarefas sejam executadas ao mesmo tempo, isso não implica que serão executadas simultaneamente.

Em relação ao **Results**, é possível monitorar a execução em tempo real de todas as versões sob teste. Por exemplo, quando os testes são realizados localmente ou no SauceLabs, é possível acompanhar as execuções, capturar vídeos e screenshots. Um relatório de avaliação é também gerado para todos os casos de teste para mostrar quais deles revelaram defeitos no cenário multi-tenant. O repositório *History Logs* é atualizado com os testes paralelos, uma vez que podem ser re-executados.

A ideia principal dessa estratégia de teste é simular múltiplas requisições que levam às definições da variável de identificação dos tenants e os usos dessa variável. Embora o teste paralelo e a utilização da técnica funcional de teste permitem identificar certos tipos de defeitos, existem limitações em considerar somente a perspectiva do usuário, como a falta de rastreabilidade no código para identificar as razões pelas quais determinados cenários de utilização levaram a um determinado defeito. Para contornar essa dificuldade, a família de critérios de cobertura proposta somada a utilização de um mecanismo de instrumentação do código contribuem com a definição de casos de teste ainda mais relevantes para explorar a estrutura interna de sistemas SaaS.

6.4 Considerações finais

Na proposição de critérios de teste sempre haverá um risco de deixar de fora algum caso de teste que poderia revelar um defeito, um vez que por mais que sejam utilizadas as melhores práticas de desenvolvimento e revisão, não se pode garantir a ausência de defeitos. Nesse sentido, não existe um procedimento que indique se existe ou não algum dado de teste que faça com que um caminho do programa seja executado, isso deve-se ao princípio de indecidibilidade. Entretanto, a medida que novos modelos surgem, novas estratégia de teste ou extensões às existentes precisam ser definidas para auxiliar na avaliação da qualidade do produto de software em questão e estabelecer um nível de confiabilidade.

As aplicações desenvolvidas para nuvem são altamente dinâmicas e escaláveis, multi-tenancy é o principal mecanismo que permite compartilhar uma única instância da aplicação de forma que recursos computacionais possam ser compartilhados e ao mesmo tempo ser capaz de isolar virtualmente as versões projetadas para cada tenant. Cada organização consegue aplicar alterações partindo de objetos padronizados, criar novas estruturas, alterar elementos de interface e regras de negócio. Essa capacidade impõe certas preocupações quanto à garantia da qualidade de uma solução multi-tenant. Por exemplo, não existem razões para uma mudança no fluxo de execução entre versões designadas para tenants diferentes. No entanto, um engano durante o desenvolvimento ou a falta de um gerenciamento adequado do isolamento de sessões ou de estruturas que deveriam permanecer isoladas podem resultar em defeitos críticos. Diante disso,

considerar somente a visão do usuário nos testes não oferece um nível de confiabilidade aceitável, portanto, o enfoque à estrutura interna da aplicação e a realização dos testes em cenários que simulem situações reais podem aumentar as chances de encontrar defeitos que não tenham sido identificados em fases prévias do teste. Além disso, considerar versões isoladas da aplicação pode não revelar defeitos em caminhos raramente exercitados ou quando múltiplas versões são verificadas simultaneamente. No Capítulo 7 apresenta-se uma avaliação da aplicabilidade da família de critérios proposta. Duas versões instrumentadas do *MtShop* e uma do *iCardapio* foram submetidas aos testes paralelos e automatizados para evidenciar a probabilidade de satisfação dos critérios, análise de cobertura e identificação de defeitos.

AVALIAÇÃO DA APLICABILIDADE DOS CRITÉRIOS PROPOSTOS

7.1 Considerações iniciais

As aplicações multi-tenants possuem diversas características e estratégias para permitir um alto nível de customização, geralmente essas abordagens estão relacionadas com a forma de estruturar e manipular *schemas* e banco de dados, gerenciamento de metadados para composição de versões específicas aos tenants, processamento e isolamento de dados e mecanismos para identificar os recursos de um dado tenant. Por exemplo, um conjunto de tabelas e objetos internos à aplicação são utilizados para gerenciar metadados que correspondem aos objetos dos schemas específicos de cada tenant (XUE; QINGZHONG; LANJU, 2011). Isso permite que sejam criadas novas tabelas estendendo as fornecidas inicialmente e que novas classes de objetos sejam projetadas de acordo com as necessidades de uma organização em particular (BOBROWSKI, 2016). Partindo dessas características e sob uma perspectiva de teste de software existem muitas preocupações para atingir uma aplicação de alta qualidade. Assim, é pouco provável uma estratégia que seja suficientemente completa para cobrir todos os elementos essenciais de uma solução SaaS multi-tenant (GAO; BAI; TSAI, 2011; CHANA; CHAWLA, 2013)

Os estudos preliminares envolvendo a técnica funcional foram exploratórios e demonstraram que verificar uma aplicação multi-tenant simulando as ações dos usuários e suas requisições é uma alternativa válida para identificação de problemas que levam aos desvios nos fluxos de execução. Porém, trata-se de uma abordagem limitada pois não oferece um nível de confiabilidade em termos de rastreabilidade, principalmente em relação às definições e os usos subsequentes da variável de identificação dos tenants. Conforme discutido anteriormente, ao mesmo tempo que uma nova requisição parte de um tenant e a aplicação precisa fornecer os resultados cabíveis, outros tenants executam operações assíncronas que utilizam estruturas compartilhadas e depen-

dendo da forma como o gerenciamento e isolamento de recursos foi desenvolvido, podem existir defeitos que afetam os fluxos de execução de cada versão. Assim, o enfoque na estrutura interna da aplicação e a realização dos testes em cenários que simulem situações reais podem aumentar as chances de encontrar defeitos que não tenham sido identificados em fases prévias do teste. Nesse sentido, além das contribuições mencionadas em capítulos anteriores para a atividade de teste de sistemas SaaS multi-tenant, é importante destacar que a família de critérios de teste proposta é direcionada exclusivamente para o escopo da identificação dos tenants considerando as interceptações das requisições, usos da variável que os identifica e os redirecionamentos.

Neste capítulo apresentam-se a utilização do modelo de teste $PCFG_{MT}$ e avaliação da família de critérios de cobertura proposta e da estratégia de teste considerando o *MtShop* e o *iCardapio*, uma vez que elas apresentam características diferentes, cujas complexidades foram apresentadas nas Tabelas 1 e 9, respectivamente. Para isso, foram consideradas três versões: duas a partir do *MtShop* e uma do *iCardapio*. Com relação ao primeiro sistema, denominam-se *MtShopFails* com defeitos de natureza concorrente, identificados em estudo prévios e *MtShopSafe* com esses defeitos resolvidos adotando uma série de práticas e alterações. Essas versões de avaliação foram instrumentadas off-line¹ com o apoio da *JaCoCo*² e depois submetidas aos testes automatizados com a finalidade de investigar a probabilidade de satisfação dos critérios, cobertura arquitetural e os defeitos revelados com o apoio dessa estratégia de teste.

7.2 Estudos de Caso

Para utilizar o modelo $PCFG_{MT}$ e a família de critérios de cobertura propostos é importante identificar todas as partes no código em que ocorrem as definições e usos do identificador dos tenants para posteriormente definir a aplicação em termos de grafos e extrair os elementos requeridos para o teste com base nos critérios de cobertura. De posse disso, parte-se para o planejamento, elaboração, execução dos casos de teste e por fim, análise dos resultados dos testes.

Com relação às definições e usos da variável utilizada para identificar cada tenant e seus recursos, recomenda-se a localização dos métodos que colaboram com a interceptação das requisições, geralmente trata-se de uma classe que estende alguma provida pelo framework adotado. No caso de aplicações desenvolvidas com *Spring Boot*, estende-se a classe *HandlerInterceptorAdapter*. Os usos geralmente estão em classes que implementam a camada *Controller* em um modelo MVC. Aplicações desenvolvidas em outras linguagens seguem o mesmo princípio ou parte-se para o instanciamento de um *middleware* com a mesma finalidade. Um exemplo disso é o framework *Django-tenants*³ para linguagem Python que permite o *Django*⁴, um framework

¹ <https://www.jacoco.org/jacoco/trunk/doc/offline.html>

² <https://www.jacoco.org/>

³ <https://github.com/tomturner/django-tenants>

⁴ <https://www.djangoproject.com/>

para desenvolvimento web, apoiar o desenvolvimento de sistemas SaaS para ter vários tenants utilizando schemas via *PostgreSQL*⁵. Tenants são definidos por meio do nome do host e o identificador é armazenado em um schema público.

Após a identificação dos métodos ligados a definição do tenant corrente a partir de uma dada requisição, ocorrem diversas operações para combinar as funcionalidades do tenant correspondente e conectar com o banco de dados ou um *schema* adequado. Muitas rotas que conduzem o usuário para as páginas web são mapeadas com métodos que fazem o uso do identificador. Determinados redirecionamentos na aplicação precisam incluir o valor dessa variável (implicando em novas definições) para manter os fluxos de execução restritos a uma versão da aplicação multi-tenant. Uma forma de identificar essas rotas é identificar todos os cenários de utilização a partir da especificação do sistema (KANER, 2003) e assim definir um ou mais casos de testes para cada cenário. A seguir descrevem-se os estudos de caso referentes às versões das aplicações.

7.2.1 MtShopFails

No caso do *MtShop*, a identificação dos métodos que realizam a definição da variável de identificação e os locais onde ocorrem algum tipo de uso dessa variável foi relativamente simples, o conhecimento do domínio e o fato desse sistema ter sido desenvolvido durante o doutorado auxiliaram nessa etapa. Na versão *MtShopFails*, apenas a classe *MultiTenancyInterceptor* (A) é responsável pela definição da variável, enquanto que 7 classes possuem algum *c-uso* e *r-uso* do identificador: *AdminProduct* (B), *RegisterController* (C), *OrderController* (D), *AdminHome* (E), *ProductController* (F), *HomeController* (G) e *CartController* (H). Os principais trechos de código dessas classes são descritos no Apêndice A. Com relação aos *p-uso*, foram identificados somente em páginas JSP: *cart.jsp*, *viewproduct.jsp* e *header.jsp*, os principais trechos de código dessas páginas estão também neste apêndice.

O código da *header.jsp* é incorporado em todas as páginas web fornecidas aos usuários. A página *cart.jsp* é utilizada nas operações envolvendo o carrinho de compras para todas as versões. Conforme destacado no Código-fonte 14, o *p-uso* da variável de identificação “*tenantid*” deve-se às funcionalidades dinâmicas que podem ser customizadas para cada versão fornecida a partir do *MtShopFails*. O método *getCart()* da classe *CartController* (Código-fonte 13) fornece a página *cart.jsp* (linha 20) a partir de uma requisição, incluindo identificador do tenant (linha 2). Vale ressaltar que embora a estratégia de teste e a família de critérios propostas não sejam direcionadas para a “camada do cliente”, é importante identificar onde ocorrem os usos do identificador mesmo que estejam na camada *View* da aplicação para aplicar alguma técnica de teste adequada. As interações com essas páginas para efeitos de modelagem não serão consideradas no modelo *PCFG_{MT}*. Mesmo assim, tais páginas ou terceiras que às utilizam são verificadas indiretamente

⁵ <https://www.postgresql.org/>

quando considera-se o teste automatizado, pois existem verificações que comparam as interfaces obtidas com as esperadas em relação a URL e elementos da interface.

Na Tabela 16 apresentam-se todos os cenários considerados nos testes para o *MtShopFails* e *MtShopSafe* e uma breve descrição contendo o perfil do usuário (caso necessário) e a página correspondente. Note que somente os cenários VIII e IX exigem casos de teste que podem ser elaborados com a técnica funcional, pois envolvem o registro de clientes e de produtos, enquanto que os demais cenários requerem a condução de testes como os de aceitação. Entretanto, tais testes precisam ser realizados de forma a cobrir as arestas que envolvem requisições e redirecionamentos das versões designadas aos tenants, definições e usos subsequentes da variável de identificação e ainda levar em conta possíveis impactos das versões sendo executadas simultaneamente.

Tabela 16 – Cenários de teste em relação as versões *MtShopFails* e *MtShopSafe*.

Cenário	Descrição (<perfil exigido>)[<página>]
I	Acesso a página que lista todos os produtos [All Products]
II	Acesso a página que lista produtos por categoria [All Products]
III	Acesso a página home [Home]
IV	Acesso a página about [About Us]
V	Acesso a página login [Login]
VI	Acesso a página do carrinho de compras (USER) [Cart]
VII	Acesso a página de detalhes de um Produto [Product Detail]
VIII	Registro de Cliente [Register Customer]
IX	Registro de Produtos (ADMIN) [Product Inventory]
X	Acesso a página administrativa após Login (ADMIN) [Login]
XI	Acesso a página home após Login (USER) [Login]
XII	Acesso a página após logout [Login]
XIII	Incluir itens no carrinho (USER) [Product Detail]
XIV	Excluir itens do carrinho (USER) [Cart]
XV	Fechar e submeter pedido de compra (USER) [Order]
XVI	Excluir produto (ADMIN) [Product Inventory]
XVII	Alterar produto (ADMIN) [Product Inventory]
XVIII	Listar usuários (ADMIN) [Customer Management]

Na Figura 30 ilustra-se o modelo $PCFG_{MT}$ elaborado para o *MtShopFails* levando em conta as abstrações discutidas anteriormente e somente as estruturas das classes que fazem uso do identificador. Essa aplicação é extensível em termos de versões que podem ser oferecidas, no entanto foram configuradas somente três versões e com isso, pode-se admitir que $N_d = \{t_1, t_2, t_3\}$, ou seja, existem fluxos de execução distintos para cada versão em relação ao seu identificador correspondente. N_{post} contém os nós de A e N_d contém os nós provenientes das classes $B - H$. Assim, para os nós de B a H existem três camadas. Por exemplo, B_i^1 , B_i^2 e B_i^3 contém os nós referentes à B para o “tenant 1”, “tenant 2” e “tenant 3”, respectivamente.

Como medida para tornar o modelo mais claro e evitar um emaranhado de associações, as arestas de redirecionamentos e requisições (*r-uso*) estão identificadas porém devem ser compreendidas da seguinte forma:

Figura 30 – $PCFG_{MT}$ para a aplicação *MtShopFails*.

- Todos os nós que recebem uma determinada **requisição**, destacada pela seta tracejada com a descrição *<request>* possuem uma relação com o nó 8 da Classe *MultiTenancyInterceptor* (A_8^0), uma vez que todas as requisições passam primeiramente por essa Classe. Nesses casos, existem arestas mapeadas com uma associação *r-uso* entre A_8^0 e tais nós. Por exemplo, tem-se as seguintes arestas de requisição entre A e B: (A_8^0, B_2^1) , (A_8^0, B_{14}^1) , (A_8^0, B_{24}^1) , (A_8^0, B_1^1) e (A_8^0, B_{13}^1) , e assim por diante para os grafos C-H.
- Todos os nós pelos quais partem setas pontilhadas com a descrição *<redirect>* possuem uma relação com o nó A_1^0 , ou seja, existe um **redirecionamento** que parte do nó com esse mapeamento para A_1^0 . Em tais casos, existem arestas mapeadas com uma associação *r-uso* desses nós para A_1^0 . Por exemplo, tem-se as seguintes arestas de redirecionamento: (B_{12}^1, A_1^0) , (B_{23}^1, A_1^0) , (B_{25}^1, A_1^0) e (H_2^1, A_1^0) , pois somente as Classes *AdmiProduct* e *CartController* possuem nós de redirecionamento.

Vale ressaltar que entre A e as demais classes, existem uma série de operações internas ao framework *Spring Boot*, empregado para apoiar o desenvolvimento dos mecanismos da arquitetura multi-tenant. No entanto, o enfoque do modelo e dos testes se restringem à estrutura interna

do código desenvolvido. Com base nessas classes, foram derivados os conjuntos necessários do modelo de teste, os quais possibilitam a aplicação dos critérios de cobertura propostos. Essas informações são apresentadas na Tabela 17.

Conhecer quais caminhos foram exercitados é fundamental para avaliar os elementos requeridos cobertos em uma determinada execução. Uma opção para obter essa informação é instrumentar o código fonte para produzir o rastreamento da execução. Essa instrumentação pode alterar o comportamento original de uma aplicação. Entretanto, essa interferência não afeta o teste estrutural proposto, pois não impede a extração e a execução futura de todas as possíveis arestas que envolvem as associações. Na Tabela 18 descrevem-se alguns elementos requeridos para os critérios propostos em relação ao *MtShopFails*.

Tabela 17 – Conjuntos obtidos pela aplicação do modelo de teste para à versão *MtShopFails*.

$D = \{t_1, t_2, t_3\}$
$N = N_{post} \cup N_d$
$N_{post} = \{A_1^0, A_2^0, A_3^0, A_4^0, A_5^0, A_6^0, A_7^0, A_8^0\}$
$N_d = N_1 \cup N_2 \cup N_3$
$N_1 = \{B_1^1, \dots, B_{25}^1, C_1^1, \dots, C_{16}^1, D_1^1, \dots, D_8^1, E_1^1, \dots, E_3^1, F_1^1, \dots, F_3^1, G_1^1, \dots, G_9^1, H_1^1, \dots, H_3^1\}$
$N_2 = \{B_1^2, \dots, B_{25}^2, C_1^2, \dots, C_{16}^2, D_1^2, \dots, D_8^2, E_1^2, \dots, E_3^2, F_1^2, \dots, F_3^2, G_1^2, \dots, G_9^2, H_1^2, \dots, H_3^2\}$
$N_3 = \{B_1^3, \dots, B_{25}^3, C_1^3, \dots, C_{16}^3, D_1^3, \dots, D_8^3, E_1^3, \dots, E_3^3, F_1^3, \dots, F_3^3, G_1^3, \dots, G_9^3, H_1^3, \dots, H_3^3\}$
$N_{red} = \{A_1^0, B_{12}^1, B_{23}^1, B_{25}^1, H_2^1, B_{12}^2, B_{23}^2, B_{25}^2, H_2^2, B_{12}^3, B_{23}^3, B_{25}^3, H_2^3\}$
$N_{req} = \{A_8^0, B_1^1, B_2^1, B_{13}^1, B_{14}^1, B_{24}^1, C_1^1, C_2^1, D_1^1, D_5^1, E_1^1, E_2^1, E_3^1, F_1^1, F_2^1, F_3^1, G_1^1, G_2^1, G_8^1, G_9^1, H_1^1, H_3^1, B_1^2, \dots, H_3^2, B_1^3, \dots, H_3^3\}$
$E = E_d \cup E_{post} \mid E_d = E_1 \cup E_2 \cup E_3 \ (d = t_1, \dots, t_3)$
$E_1 = \{(B_2^1, B_3^1), (B_3^1, B_4^1), (B_3^1, B_5^1), (B_5^1, B_6^1), (B_5^1, B_7^1), (B_7^1, B_8^1), (B_7^1, B_9^1), (B_9^1, B_{10}^1), (B_9^1, B_{12}^1), (B_{10}^1, B_{11}^1), (B_{11}^1, B_{12}^1), (B_{10}^1, B_{12}^1), (B_{14}^1, B_{15}^1), (B_{15}^1, B_{16}^1), (B_{15}^1, B_{17}^1), (B_{17}^1, B_{18}^1), (B_{17}^1, B_{20}^1), (B_{18}^1, B_{19}^1), (B_{18}^1, B_{23}^1), (B_{19}^1, B_{23}^1), (B_{20}^1, B_{21}^1), (B_{21}^1, B_{22}^1), (B_{21}^1, B_{23}^1), (B_{22}^1, B_{23}^1), (B_{24}^1, B_{25}^1), (C_2^1, C_3^1), (C_3^1, C_4^1), (C_3^1, C_5^1), (C_5^1, C_6^1), (C_5^1, C_7^1), (C_7^1, C_8^1), (C_8^1, C_9^1), (C_9^1, C_{10}^1), (C_9^1, C_{11}^1), (C_{10}^1, C_{11}^1), (C_{11}^1, C_8^1), (C_8^1, C_{12}^1), (C_{12}^1, C_{13}^1), (C_{13}^1, C_{14}^1), (C_{13}^1, C_{15}^1), (C_{15}^1, C_{12}^1), (C_{14}^1, C_{15}^1), (C_{12}^1, C_{16}^1), (D_1^1, D_2^1), (D_2^1, D_3^1), (D_3^1, D_2^1), (D_2^1, D_4^1), (D_5^1, D_6^1), (D_6^1, D_7^1), (D_6^1, D_8^1), (D_7^1, D_8^1), (G_2^1, G_3^1), (G_3^1, G_4^1), (G_4^1, G_5^1), (G_3^1, G_5^1), (G_5^1, G_6^1), (G_5^1, G_7^1), (G_6^1, G_7^1), (H_1^1, H_2^1)\}$
$E_2 = \{(B_2^2, B_3^2), \dots, (H_1^2, H_2^2)\}$
$E_3 = \{(B_2^3, B_3^3), \dots, (H_1^3, H_2^3)\}$
$E_{post} = \{(A_1^0, A_2^0), (A_2^0, A_3^0), (A_3^0, A_4^0), (A_4^0, A_5^0), (A_5^0, A_6^0), (A_6^0, A_7^0), (A_7^0, A_8^0), (A_8^0, A_1^0)\}$
$E_{red} = \{(B_{12}^1, A_1^0), (B_{23}^1, A_1^0), (B_{25}^1, A_1^0), (H_2^1, A_1^0), (B_{12}^2, A_1^0), (B_{23}^2, A_1^0), (B_{25}^2, A_1^0), (H_2^2, A_1^0), (B_{12}^3, A_1^0), (B_{23}^3, A_1^0), (B_{25}^3, A_1^0), (H_2^3, A_1^0)\}$
$E_{req} = \{(A_8^0, B_1^1), (A_8^0, B_2^1), (A_8^0, B_{13}^1), (A_8^0, B_{14}^1), (A_8^0, B_{24}^1), (A_8^0, C_1^1), (A_8^0, C_2^1), (A_8^0, D_1^1), (A_8^0, D_5^1), (A_8^0, E_1^1), (A_8^0, E_2^1), (A_8^0, E_3^1), (A_8^0, F_1^1), (A_8^0, F_2^1), (A_8^0, F_3^1), (A_8^0, G_1^1), (A_8^0, G_2^1), (A_8^0, G_8^1), (A_8^0, G_9^1), (A_8^0, H_1^1), (A_8^0, H_3^1), (A_8^0, B_1^2), \dots, (A_8^0, B_{13}^2), (A_8^0, B_{14}^2), (A_8^0, B_{24}^2), (A_8^0, C_1^2), \dots, (A_8^0, H_3^2)\}$

Execução dos testes automatizados com a versão *MtShopFails*

Para verificar as três versões do *MtShopFails* sob a perspectiva da utilização da família de critérios proposta, os casos de testes previamente elaborados no segundo estudo discutido no Capítulo 5 foram reexecutados. A diferença dessa nova execução está em alterações na aplicação em virtude da JaCoCo para instrumentar o código e monitorar as execuções paralelas da aplicação. A JaCoCo é uma ferramenta de qualidade de software Java de código aberto para medir a cobertura de código. Isso permite visualizar quais linhas do código foram testadas.

Tabela 18 – Elementos requeridos pelos critérios de teste em relação a versão *MtShopFails*.

Critério	Elementos requeridos	Total
<i>Todos-Nós-req</i>	$A_8^0, B_1^1, B_2^1, B_{13}^1, B_{14}^1, B_{24}^1, C_1^1, C_2^1, D_1^1, D_5^1, E_1^1, E_2^1, E_3^1, F_1^1, F_2^1, F_3^1, G_1^1, G_2^1, G_8^1, G_9^1, H_1^1, H_3^1, B_1^2, B_2^2, B_{13}^2, B_{14}^2, B_{24}^2, C_1^2, C_2^2, D_1^2, D_5^2, E_1^2, E_2^2, E_3^2, F_1^2, F_2^2, F_3^2, G_1^2, G_2^2, G_8^2, G_9^2, H_1^2, H_3^2, B_1^3, B_2^3, B_{13}^3, B_{14}^3, B_{24}^3, C_1^3, C_2^3, D_1^3, D_5^3, E_1^3, E_2^3, E_3^3, F_1^3, F_2^3, F_3^3, G_1^3, G_2^3, G_8^3, G_9^3, H_1^3, H_3^3$	65
<i>Todos-Nós-red</i>	$A_1^0, B_{12}^1, B_{23}^1, B_{25}^1, H_2^1, B_{12}^2, B_{23}^2, B_{25}^2, H_2^2, B_{12}^3, B_{23}^3, B_{25}^3, H_2^3$	13
<i>Todos-Nós</i>	$A_1^0, ..., A_8^0, B_1^1, ..., B_{25}^1, C_1^1, ..., C_{16}^1, D_1^1, ..., D_8^1, E_1^1, ..., E_3^1, F_1^1, ..., F_3^1, G_1^1, ..., G_9^1, H_1^1, ..., H_3^1, B_1^2, ..., B_{25}^2, C_1^2, ..., C_{16}^2, D_1^2, ..., D_8^2, E_1^2, ..., E_3^2, F_1^2, ..., F_3^2, G_1^2, ..., G_9^2, H_1^2, ..., H_3^2, B_1^3, ..., B_{25}^3, C_1^3, ..., C_{16}^3, D_1^3, ..., D_8^3, E_1^3, ..., E_3^3, F_1^3, ..., F_3^3, G_1^3, ..., G_9^3, H_1^3, ..., H_3^3$	209
<i>Todas-Arestas-req</i>	$(A_8^0, B_1^1), (A_8^0, B_2^1), (A_8^0, B_{13}^1), (A_8^0, B_{14}^1), (A_8^0, B_{24}^1), (A_8^0, C_1^1), (A_8^0, C_2^1), (A_8^0, D_1^1), (A_8^0, D_5^1), (A_8^0, E_1^1), (A_8^0, E_2^1), (A_8^0, E_3^1), (A_8^0, F_1^1), (A_8^0, F_2^1), (A_8^0, F_3^1), (A_8^0, G_1^1), (A_8^0, G_2^1), (A_8^0, G_8^1), (A_8^0, G_9^1), (A_8^0, H_1^1), (A_8^0, H_3^1), (A_8^0, B_1^2), ..., (A_8^0, H_3^2), (A_8^0, B_1^3), ..., (A_8^0, H_3^3)$	63
<i>Todas-Arestas-red</i>	$(B_{12}^1, A_1^0), (B_{23}^1, A_1^0), (B_{25}^1, A_1^0), (H_2^1, A_1^0), (B_{12}^2, A_1^0), (B_{23}^2, A_1^0), (B_{25}^2, A_1^0), (H_2^2, A_1^0), (B_{12}^3, A_1^0), (B_{23}^3, A_1^0), (B_{25}^3, A_1^0), (H_2^3, A_1^0)$	12
<i>Todas-Def-ident</i>	$(A_7^0, A_8^0), (A_7^0, (A_8^0, B_1^1)), (A_7^0, (A_8^0, B_2^1)), (A_7^0, (A_8^0, B_{13}^1)), (A_7^0, (A_8^0, B_{14}^1)), (A_7^0, (A_8^0, B_{24}^1)), (A_7^0, (A_8^0, C_1^1)), (A_7^0, (A_8^0, C_2^1)), (A_7^0, (A_8^0, D_1^1)), (A_7^0, (A_8^0, D_5^1)), (A_7^0, (A_8^0, E_1^1)), (A_7^0, (A_8^0, E_2^1)), (A_7^0, (A_8^0, E_3^1)), (A_7^0, (A_8^0, F_1^1)), (A_7^0, (A_8^0, F_2^1)), (A_7^0, (A_8^0, F_3^1)), (A_7^0, (A_8^0, G_1^1)), (A_7^0, (A_8^0, G_2^1)), (A_7^0, (A_8^0, G_8^1)), (A_7^0, (A_8^0, G_9^1)), (A_7^0, (A_8^0, H_1^1)), (A_7^0, (A_8^0, H_3^1)), (A_7^0, (A_8^0, B_1^2)), ..., (A_7^0, (A_8^0, H_3^2)), (A_7^0, (A_8^0, B_1^3)), ..., (A_7^0, (A_8^0, H_3^3))$	64
<i>Todos-Usos-ident</i>	$(A_1^0, (A_2^0, A_8^0)), (B_1^1, B_8^1), (D_2^1, D_4^1), (G_5^1, G_7^1), (G_6^1, G_7^1), (B_7^2, B_8^2), ..., (G_6^2, G_7^2), (B_3^3, B_8^3), ..., (G_6^3, G_7^3), (A_7^0, (B_{12}^1, A_1^0)), (A_7^0, (B_{23}^1, A_1^0)), (A_7^0, (B_{25}^1, A_1^0)), (A_7^0, (H_2^1, A_1^0)), (A_7^0, (B_{12}^2, A_1^0)), ..., (A_7^0, (H_2^2, A_1^0)), (A_7^0, (B_{12}^3, A_1^0)), ..., (A_7^0, (H_2^3, A_1^0)), (A_2^0, (A_8^0, B_1^1)), (A_7^0, (A_8^0, B_1^1)), (A_2^0, (A_8^0, B_2^1)), (A_7^0, (A_8^0, B_2^1)), (A_2^0, (A_8^0, B_{13}^1)), (A_7^0, (A_8^0, B_{13}^1)), (A_2^0, (A_8^0, B_{14}^1)), (A_7^0, (A_8^0, B_{14}^1)), (A_2^0, (A_8^0, B_{24}^1)), (A_7^0, (A_8^0, B_{24}^1)), (A_2^0, (A_8^0, C_1^1)), (A_7^0, (A_8^0, C_1^1)), (A_2^0, (A_8^0, C_2^1)), (A_7^0, (A_8^0, C_2^1)), (A_2^0, (A_8^0, D_1^1)), (A_7^0, (A_8^0, D_1^1)), (A_2^0, (A_8^0, D_5^1)), (A_7^0, (A_8^0, D_5^1)), (A_2^0, (A_8^0, E_1^1)), (A_7^0, (A_8^0, E_1^1)), (A_2^0, (A_8^0, E_2^1)), (A_7^0, (A_8^0, E_2^1)), (A_2^0, (A_8^0, E_3^1)), (A_7^0, (A_8^0, E_3^1)), (A_2^0, (A_8^0, F_1^1)), (A_7^0, (A_8^0, F_1^1)), (A_2^0, (A_8^0, F_2^1)), (A_7^0, (A_8^0, F_2^1)), (A_2^0, (A_8^0, F_3^1)), (A_7^0, (A_8^0, F_3^1)), (A_2^0, (A_8^0, G_1^1)), (A_7^0, (A_8^0, G_1^1)), (A_2^0, (A_8^0, G_2^1)), (A_7^0, (A_8^0, G_2^1)), (A_2^0, (A_8^0, G_8^1)), (A_7^0, (A_8^0, G_8^1)), (A_2^0, (A_8^0, G_9^1)), (A_7^0, (A_8^0, G_9^1)), (A_2^0, (A_8^0, H_1^1)), (A_7^0, (A_8^0, H_1^1)), (A_2^0, (A_8^0, H_3^1)), (A_7^0, (A_8^0, H_3^1)), (A_2^0, (A_8^0, B_1^2)), ..., (A_7^0, (A_8^0, H_3^2)), (A_2^0, (A_8^0, B_1^3)), ..., (A_7^0, (A_8^0, H_3^3))$	145

Juntamente com a cobertura, a JaCoCo também relata a complexidade de cada método e informa quanto da complexidade de um método ainda não foi testado.

A ideia nesse processo é verificar o nível de cobertura atingida com a abordagem de teste. Vale ressaltar que a adoção da família de critérios proposta é voltada para apoiar a definição dos elementos a serem exercitados pelos testes. Um dos princípios de qualquer critério de teste é apoiar a construção de um conjunto de casos de teste que seja relevante para identificação de defeitos e além disso, permitir o estabelecimento de um ponto de parada para os testes que permita obter um nível de confiabilidade do software (DELAMARO; MALDONADO; JINO, 2007). Com base nessa premissa, se os casos de testes fossem elaborados seguindo estritamente os critérios estabelecidos, os dados de teste se resumiriam no valor do identificador para cada tenant, ou seja, todos os testes para uma versão “W”, projetada para o “tenant w”, teriam que incluir o identificador correspondente para essa versão e a partir disso bastaria verificar todos os usos subsequentes e as definições que ocorrem partindo dos demais tenants.

Para cobrir os elementos requeridos para cada critério é necessário elaborar casos de teste capazes exercitá-los, ou seja, com a inserção dos dados de teste na aplicação deve-se observar se uma dada aresta ou nó foi coberto e se a saída obtida está em conformidade com a saída esperada. Entretanto, para percorrer os caminhos de execução desejados notou-se a necessidade de que a inserção de dados ocorresse nas páginas web e que as operações sob a visão do usuário fossem acionadas, uma vez que isso levará como uma consequência a execução dos caminhos definidos no modelo $PCFG_{MT}$. Com isso em mente, a abordagem de testes automatizados e paralelos aliada às definições do modelo $PCFG_{MT}$ permitem definir os fluxos de execução a serem verificados em relação às versões, partindo das definições da variável de identificação que ocorrem de forma sucessiva mediante às novas requisições e redirecionamentos.

Ao empregar os critérios da técnica funcional EPC e BVA, 184 casos de testes foram elaborados com base nas especificações das classes *Customer* (144) e *Product* (40). No Apêndice D apresentam-se os casos de teste e a relação de defeitos identificados com a apoio dos testes automatizados. A reexecução dos testes automatizados com os casos de teste previamente estabelecidos e o modelo de teste permitiram avaliar quais caminhos foram cobertos e quais usos foram exercitados. A rastreabilidade em função do identificador do tenant permite coletar dados e observar a satisfação dos critérios, principalmente em relação às requisições e redirecionamentos que podem levar a novas definições da variável de identificação. Na Tabela 19 descrevem-se a relação dos critérios, a quantidade de elementos requeridos e a cobertura atingida com a execução dos testes para o *MtShopFails*.

Tabela 19 – Cobertura alcançada com os critérios de teste em relação a versão *MtShopFails*.

Critério	Total	Cobertura
<i>Todos-Nós-req</i>	65	95.4%
<i>Todos-Nós-red</i>	13	100%
<i>Todos-Nós</i>	209	82%
<i>Todas-Arestas-req</i>	63	95.2%
<i>Todas-Arestas-red</i>	12	100%
<i>Todas-Def-ident</i>	64	100%
<i>Todos-Usos-ident</i>	145	95%

A execução dos testes automatizados para as três versões fornecidas pelo *MtShopFails* levaram a identificação de vários defeitos. Na Tabela 20 descrevem-se o número de ocorrências para cada um deles. Os defeitos em relação ao *RC/MI* foram identificados da seguinte forma: 47 foram no cadastro de *Product* e 10 no cadastro de *Customer*. De modo que, 46 vezes ocorreram $(A_8^0, (B_2^3, B_{12}^2))$ quando o correto seria $(A_8^0, (B_2^3, B_{12}^3))$, ou seja, o caminho (B_2^3, B_{12}^2) está incorreto em função da versão designada ao “tenant 3”; houve também uma ocorrência de $(A_8^0, (B_2^1, B_{12}^2))$, ou seja, desvio no fluxo de execução da versão do “tenant 1”; 9 vezes ocorreram (A_8^0, C_2^1) quando esperava-se (A_8^0, C_2^2) e por fim, uma ocorrência de (A_8^0, C_2^3) quando esperava-se (A_8^0, C_2^2) . Esses defeitos permitem que outro tenant acesse partes não previstas ou gerem saídas incorretas.

Com relação a cobertura os resultados foram satisfatórios, entretanto alguns nós e arestas

Tabela 20 – Defeitos identificados na versão *MtShopFails*.

Defeitos	Ocorrências
<i>Missing Command Error</i> (MCE)	23
<i>Control Flow Error</i> (CFE)	40
<i>Functionality Error</i> (FE)	20
<i>Implementation Error</i> (IE)	157
<i>Suspension Error</i> (SUE)	86
<i>Race Condition and Memory Inconsistency for tenant identification</i> (RC/MI)	57

não foram cobertas por serem não executáveis ou os casos de teste não cobriram estruturas que envolviam exceções. Por exemplo, o nós G_8^1 , G_8^2 e G_8^3 não foram cobertos pois os testes não exercitaram a requisição correspondente em que o acesso é bloqueado em caso de acesso a rotas indevidas. Assim, as arestas como (A_8^0, G_8^1) e $(A_7^0, (A_8^0, G_8^1))$ consequentemente não foram exercitadas. Os critérios de cobertura propostos fornecem diretrizes para rastrear e identificar defeitos potenciais nas partes do código que possuem natureza concorrente, como o SUE e o RC/MI. Além disso, o modelo $PCFG_{MT}$ e a aplicação dos critérios orientam a observação de quais fluxos de execução de uma determinada versão podem sofrer impactos pela falta de um controle mais efetivo no isolamento dos tenants gerando diversos defeitos, inclusive aqueles que são tipicamente sequenciais.

7.2.2 *MtShopSafe*

Com a identificação de defeitos de natureza concorrente no *MtShop* com apoio dos testes funcionais e o rastreamento dos mesmos com os critérios de cobertura, aplicaram-se diversas alterações na aplicação para solucioná-los. Essas alterações envolveram principalmente a **política de autenticação** e o **isolamento dos fluxos de execução e sessões por tenant**. Ao efetuar o login em uma determinada versão específica de um tenant, o identificador do tenant correspondente é enviado na requisição com os dados informados pelo usuário e nas tabelas que armazenam os dados de autenticação foi necessário incluir um campo para o identificador. Isso permite restringir os recursos da aplicação em função dessa informação partindo do usuário. Alguns desses princípios são provenientes do modelo *Stateless*⁶ - um modelo de desenvolvimento de aplicações em que cada requisição é independente. Em aplicações web que utilizam esse modelo tem-se um programa cliente que faz uma solicitação a um servidor de aplicativos, que envia dados de volta ao cliente. O servidor trata todas as conexões do cliente igualmente e não salva informações de solicitações ou sessões anteriores. Com relação ao isolamento dos fluxos de execução, o objetivo foi evitar o não determinismo nas sessões cruzadas dos tenants. Nesse sentido foi implementada uma classe chamada *TenantContextHolder* que utiliza uma *ThreadLocal* para restringir a definição e o valor do tenant corrente para cada versão, conforme comentado no Capítulo 3, além de alterações na interceptação das requisições.

Da mesma forma que na versão *MtShopFails*, apenas a classe *MultiTenancyInterceptor*

⁶ <https://nordicapis.com/defining-stateful-vs-stateless-web-services/>

(A) é responsável pela definição da variável. Com relação ao *c-uso* e *r-uso* do identificador, notou-se em 9 classes: *AdminProduct* (B), *CustomAuthenticationToken* (C), *OrderController* (D), *CustomUserDetails* (E), *CartController* (F), *HomeController* (H), *AdminHome* (I), *ProductController* (J) e *RegisterController* (L). Com relação aos *p-uso*, foram identificados nas mesmas páginas JSP utilizadas na versão *MtShopFails* (*cart.jsp*, *viewproduct.jsp* e *header.jsp*) e na classe *CustomAuthenticationFilter* (G). No Apêndice B descrevem-se os principais trechos de código dessas classes.

Similar ao estudo de caso anterior, na Figura 31 ilustra-se o modelo $PCFG_{MT}$ para o *MtShopSafe* levando em conta as abstrações definidas para o modelo e somente as estruturas das classes que fazem uso do identificador. Assim como na versão *MtShopFails* foram consideradas somente três versões, visto que o *MtShop* fornece três aplicações para domínios de produtos distintos e com isso, pode-se admitir também que $N_d = \{t_1, t_2, t_3\}$. N_{post} contém os nós de A e N_d contém os nós provenientes das classes B – L. Na Tabela 21 descrevem-se os principais elementos derivados do modelo $PCFG_{MT}$ e na Tabela 22 descrevem-se os elementos requeridos para cada critério de teste proposto considerando a versão *MtShopSafe*.

Figura 31 – $PCFG_{MT}$ para a aplicação *MtShopSafe*.

Execução dos testes automatizados com a versão *MtShopSafe*

Na Tabela 23 apresentam-se o total de elementos requeridos segundo a família de critérios de teste proposta e a cobertura alcançada ao executar os testes automatizados para o

Tabela 21 – Conjuntos obtidos pela aplicação do modelo de teste para a versão *MtShopSafe*.

$D = \{t_1, t_2, t_3\}$
$N = N_{post} \cup N_d$
$N_{post} = A_1^0, A_2^0, A_3^0, A_4^0, A_5^0, A_6^0, A_7^0, A_8^0, A_9^0, A_{10}^0, A_{11}^0$
$N_d = N_1 \cup N_2 \cup N_3$
$N_1 = B_1^1, \dots, B_{29}^1, C_1^1, C_2^1, D_1^1, \dots, D_8^1, E_1^1, E_2^1, F_1^1, \dots, F_3^1, G_1^1, \dots, G_8^1, H_1^1, \dots, H_8^1, I_1^1, I_2^1, I_3^1, J_1^1, J_2^1, J_3^1, L_1^1, \dots, L_{15}^1$
$N_2 = B_1^2, \dots, B_{29}^2, C_1^2, C_2^2, D_1^2, \dots, D_8^2, E_1^2, E_2^2, F_1^2, \dots, F_3^2, G_1^2, \dots, G_8^2, H_1^2, \dots, H_8^2, I_1^2, I_2^2, I_3^2, J_1^2, J_2^2, J_3^2, L_1^2, \dots, L_{15}^2$
$N_3 = B_1^3, \dots, B_{29}^3, C_1^3, C_2^3, D_1^3, \dots, D_8^3, E_1^3, E_2^3, F_1^3, \dots, F_3^3, G_1^3, \dots, G_8^3, H_1^3, \dots, H_8^3, I_1^3, I_2^3, I_3^3, J_1^3, J_2^3, J_3^3, L_1^3, \dots, L_{15}^3$
$N_{red} = A_1^0, B_{12}^1, B_{27}^1, B_{29}^1, F_1^1, B_{12}^2, F_2^2, B_{12}^3, F_3^3$
$N_{req} = A_{11}^0, B_1^1, B_2^1, B_{13}^1, B_{14}^1, B_{28}^1, D_1^1, D_5^1, F_1^1, F_3^1, H_1^1, H_2^1, H_8^1, I_1^1, I_2^1, I_3^1, J_1^1, J_2^1, J_3^1, L_1^1, L_2^1, B_1^2, \dots, L_2^2, B_1^3, \dots, L_2^3$
$E_1 = \{(B_2^1, B_3^1), (B_3^1, B_4^1), (B_3^1, B_5^1), (B_5^1, B_6^1), (B_5^1, B_7^1), (B_7^1, B_8^1), (B_7^1, B_9^1), (B_9^1, B_{10}^1), (B_9^1, B_{12}^1), (B_{10}^1, B_{11}^1), (B_{10}^1, B_{12}^1), (B_{11}^1, B_{12}^1), (B_{14}^1, B_{15}^1), (B_{15}^1, B_{16}^1), (B_{15}^1, B_{17}^1), (B_{17}^1, B_{18}^1), (B_{17}^1, B_{19}^1), (B_{19}^1, B_{21}^1), (B_{19}^1, B_{20}^1), (B_{21}^1, B_{24}^1), (B_{21}^1, B_{22}^1), (B_{22}^1, B_{23}^1), (B_{22}^1, B_{27}^1), (B_{23}^1, B_{27}^1), (B_{24}^1, B_{25}^1), (B_{25}^1, B_{26}^1), (B_{25}^1, B_{27}^1), (B_{28}^1, B_{29}^1), (D_1^1, D_2^1), (D_2^1, D_4^1), (D_2^1, D_3^1), (D_3^1, D_4^1), (D_5^1, D_6^1), (D_6^1, D_7^1), (D_6^1, D_8^1), (D_7^1, D_8^1), (F_1^1, F_2^1), (G_1^1, G_2^1), (G_2^1, G_3^1), (G_2^1, G_4^1), (G_3^1, G_4^1), (G_4^1, G_6^1), (G_4^1, G_5^1), (G_5^1, G_6^1), (G_6^1, G_7^1), (G_6^1, G_8^1), (G_7^1, G_8^1), (H_2^1, H_3^1), (H_3^1, H_4^1), (H_3^1, H_5^1), (H_4^1, H_5^1), (H_5^1, H_6^1), (H_5^1, H_7^1), (H_6^1, H_7^1), (L_2^1, L_3^1), (L_3^1, L_4^1), (L_3^1, L_5^1), (L_5^1, L_6^1), (L_5^1, L_7^1), (L_7^1, L_8^1), (L_7^1, L_9^1), (L_9^1, L_{10}^1), (L_9^1, L_{11}^1), (L_{11}^1, L_{12}^1), (L_{11}^1, L_{13}^1), (L_{13}^1, L_{14}^1), (L_{13}^1, L_{15}^1)\}$
$E_2 = \{(B_2^2, B_3^2), \dots, (L_{13}^2, L_{15}^2)\}$
$E_3 = \{(B_2^3, B_3^3), \dots, (L_{13}^3, L_{15}^3)\}$
$E_{post} = \{(A_1^0, A_2^0), (A_2^0, A_{12}^0), (A_2^0, A_3^0), (A_3^0, A_4^0), (A_4^0, A_5^0), (A_5^0, A_6^0), (A_6^0, A_8^0), (A_6^0, A_7^0), (A_7^0, A_8^0), (A_8^0, A_9^0), (A_4^0, A_9^0), (A_9^0, A_{10}^0), (A_9^0, A_{11}^0), (A_{10}^0, A_{11}^0)\}$
$E_{red} = \{(B_{12}^1, A_1^0), (B_{27}^1, A_1^0), (B_{29}^1, A_1^0), (F_2^1, A_1^0), (B_{12}^2, A_1^0), \dots, (F_2^2, A_1^0), (B_{12}^3, A_1^0), \dots, (F_2^3, A_1^0)\}$
$E_{req} = \{(A_{11}^0, B_1^1), (A_{11}^0, B_2^1), (A_{11}^0, B_{13}^1), (A_{11}^0, B_{14}^1), (A_{11}^0, B_{28}^1), (A_{11}^0, D_1^1), (A_{11}^0, D_5^1), (A_{11}^0, F_1^1), (A_{11}^0, F_3^1), (A_{11}^0, H_1^1), (A_{11}^0, H_2^1), (A_{11}^0, H_8^1), (A_{11}^0, I_1^1), (A_{11}^0, I_2^1), (A_{11}^0, I_3^1), (A_{11}^0, J_1^1), (A_{11}^0, J_2^1), (A_{11}^0, J_3^1), (A_{11}^0, L_1^1), (A_{11}^0, L_2^1), (A_{11}^0, B_1^2), \dots, (A_{11}^0, L_2^2), (A_{11}^0, B_1^3), \dots, (A_{11}^0, L_2^3)\}$

MtShopSafe. A execução dos testes automatizados das três versões fornecidas pelo *MtShopSafe* levaram a identificação de vários defeitos, uma vez que nem todos os defeitos foram solucionados. Pelos mesmos motivos que no estudo de caso anterior não foi alcançada a cobertura de 100% para todos os elementos requeridos pelos critérios. Para o critério *Todos-Nós*, $B_{11}^1, B_{23}^1, D_6^1, D_7^1, H_3^1, H_4^1, G_2^1, \dots, G_7^1, B_{11}^1, \dots, G_7^2$ e $B_{11}^1, \dots, G_7^3$ não foram executados e para o critério *Todos-Usos-ident*, as arestas (G_6^1, G_7^1) , (G_6^2, G_7^2) e (G_6^3, G_7^3) não foram cobertas pelos testes.

Na Tabela 24 descrevem-se o número de ocorrências para cada defeito identificado no *MtShopSafe*. Para o caso dos defeitos do tipo *Suspension Error* (SUE) e *Race Condition and Memory Inconsistency for tenant identification* (RC/MI) não houveram ocorrências, visto que os problemas com o isolamento de recursos e sessões cruzadas foram solucionados. Com base nesse segundo estudo, os critérios podem ser úteis na condução de testes de regressão para verificar se as versões fornecidas por uma aplicação multi-tenant continuam válidas após diversas evoluções ao longo do ciclo de vida. Em geral, os critérios contribuem para identificação tanto de defeitos em estruturas compartilhadas entre os tenants quanto para funcionalidades específicas. Apesar das versões consideradas serem idênticas em termos de *GFCs*, o modelo de teste *PCFG_{MT}* e os critérios podem ser utilizados para apoiar a verificação de versões distintas.

Tabela 22 – Elementos requeridos pelos critérios de teste em relação a versão *MtShopSafe*.

Critério	Elementos requeridos	Total
<i>Todos-Nós-req</i>	$A_{11}^0, B_1^1, B_2^1, B_{13}^1, B_{14}^1, B_{28}^1, D_1^1, D_5^1, F_1^1, F_3^1, H_1^1, H_2^1, H_8^1, I_1^1, I_2^1, I_3^1, J_1^1, J_2^1, J_3^1, L_1^1, L_2^1, B_1^2, \dots, L_2^3, B_1^3, \dots, L_2^3$	61
<i>Todos-Nós-red</i>	$A_{11}^0, B_{12}^1, B_{27}^1, B_{29}^1, F_2^1, B_{12}^2, \dots, F_2^2, B_{12}^3, \dots, F_2^3$	13
<i>Todos-Nós</i>	$A_1^0, \dots, A_{11}^0, B_1^1, \dots, B_{29}^1, C_1^1, C_2^1, D_1^1, \dots, D_8^1, E_1^1, E_2^1, F_1^1, \dots, F_3^1, G_1^1, \dots, G_8^1, H_1^1, \dots, H_8^1, I_1^1, \dots, I_3^1, J_1^1, \dots, J_3^1, L_1^1, \dots, L_{15}^1, B_1^2, \dots, B_{29}^2, C_1^2, C_2^2, D_1^2, \dots, D_8^2, E_2^2, E_2^3, F_1^2, \dots, F_3^2, G_1^2, \dots, G_8^2, H_1^2, \dots, H_8^2, I_1^2, \dots, I_3^2, J_1^2, \dots, J_3^2, L_2^2, \dots, L_{15}^2, B_1^3, \dots, B_{29}^3, C_1^3, C_2^3, D_1^3, \dots, D_8^3, E_1^3, E_2^3, F_1^3, \dots, F_3^3, G_1^3, \dots, G_8^3, H_1^3, \dots, H_8^3, I_1^3, \dots, I_3^3, J_1^3, \dots, J_3^3, L_3^3, \dots, L_{15}^3$	254
<i>Todas-Arestas-req</i>	$(A_{11}^0, B_1^1), (A_{11}^0, B_2^1), (A_{11}^0, B_{13}^1), (A_{11}^0, B_{14}^1), (A_{11}^0, B_{28}^1), (A_{11}^0, D_1^1), (A_{11}^0, D_5^1), (A_{11}^0, F_1^1), (A_{11}^0, F_3^1), (A_{11}^0, H_1^1), (A_{11}^0, H_2^1), (A_{11}^0, H_8^1), (A_{11}^0, I_1^1), (A_{11}^0, I_2^1), (A_{11}^0, I_3^1), (A_{11}^0, J_1^1), (A_{11}^0, J_2^1), (A_{11}^0, J_3^1), (A_{11}^0, L_1^1), (A_{11}^0, L_2^1), (A_{11}^0, B_1^2), \dots, (A_{11}^0, L_2^2), (A_{11}^0, B_1^3), \dots, (A_{11}^0, L_2^3)$	60
<i>Todas-Arestas-red</i>	$(B_{12}^1, A_1^0), (B_{27}^1, A_1^0), (B_{29}^1, A_1^0), (F_2^1, A_1^0), (B_{12}^2, A_1^0), \dots, (F_2^2, A_1^0), (B_{12}^3, A_1^0), \dots, (F_2^3, A_1^0)$	12
<i>Todas-Def-ident</i>	$(A_8^0, (A_0^0, A_{11}^0)), (A_{10}^0, (A_0^0, A_{11}^0)), (A_8^0, (A_0^0, B_1^1)), (A_8^0, (A_0^0, B_2^1)), (A_8^0, (A_0^0, B_{13}^1)), (A_8^0, (A_0^0, B_{14}^1)), (A_8^0, (A_0^0, B_{28}^1)), (A_8^0, (A_0^0, D_1^1)), (A_8^0, (A_0^0, D_5^1)), (A_8^0, (A_0^0, F_1^1)), (A_8^0, (A_0^0, F_3^1)), (A_8^0, (A_0^0, H_1^1)), (A_8^0, (A_0^0, H_2^1)), (A_8^0, (A_0^0, H_8^1)), (A_8^0, (A_0^0, I_1^1)), (A_8^0, (A_0^0, I_2^1)), (A_8^0, (A_0^0, I_3^1)), (A_8^0, (A_0^0, J_1^1)), (A_8^0, (A_0^0, J_2^1)), (A_8^0, (A_0^0, J_3^1)), (A_8^0, (A_0^0, L_1^1)), (A_8^0, (A_0^0, L_2^1)), (A_{10}^0, (A_0^0, B_1^1)), (A_{10}^0, (A_0^0, B_2^1)), (A_{10}^0, (A_0^0, B_{13}^1)), (A_{10}^0, (A_0^0, B_{14}^1)), (A_{10}^0, (A_0^0, B_{28}^1)), (A_{10}^0, (A_0^0, D_1^1)), (A_{10}^0, (A_0^0, D_5^1)), (A_{10}^0, (A_0^0, F_1^1)), (A_{10}^0, (A_0^0, F_3^1)), (A_{10}^0, (A_0^0, H_1^1)), (A_{10}^0, (A_0^0, H_2^1)), (A_{10}^0, (A_0^0, H_8^1)), (A_{10}^0, (A_0^0, I_1^1)), (A_{10}^0, (A_0^0, I_2^1)), (A_{10}^0, (A_0^0, I_3^1)), (A_{10}^0, (A_0^0, J_1^1)), (A_{10}^0, (A_0^0, J_2^1)), (A_{10}^0, (A_0^0, J_3^1)), (A_{10}^0, (A_0^0, L_1^1)), (A_{10}^0, (A_0^0, L_2^1)), (A_8^0, (A_0^0, B_1^2)), \dots, (A_{10}^0, (A_0^0, L_2^2)), (A_8^0, (A_0^0, B_1^3)), \dots, (A_{10}^0, (A_0^0, L_2^3))$	122
<i>Todos-Usos-ident</i>	$(A_1^0, (A_0^0, A_{11}^0)), (A_1^0, (A_0^0, A_{11}^0)), (B_7^1, B_8^1), (B_{19}^1, B_{20}^1), (D_2^1, D_4^1), (G_6^1, G_7^1), (G_6^1, G_8^1), (H_5^1, H_7^1), (H_6^1, H_7^1), (A_9^0, (A_1^1, B_1^1)), (A_{10}^0, (A_1^1, B_1^1)), (A_9^0, (A_1^1, B_{13}^1)), (A_{10}^0, (A_1^1, B_{13}^1)), (B_2^1, B_3^1), (B_{14}^1, B_{15}^1), (A_9^0, (A_1^1, B_{28}^1)), (A_{10}^0, (A_1^1, B_{28}^1)), (D_1^1, D_2^1), (D_5^1, D_6^1), (F_1^1, F_2^1), (A_9^0, (A_1^1, F_3^1)), (A_{10}^0, (A_1^1, F_3^1)), (A_9^0, (A_1^1, H_1^1)), (A_{10}^0, (A_1^1, H_1^1)), (A_9^0, (A_1^1, H_8^1)), (A_{10}^0, (A_1^1, H_8^1)), (A_9^0, (A_1^1, I_1^1)), (A_{10}^0, (A_1^1, I_1^1)), (A_9^0, (A_1^1, I_3^1)), (A_{10}^0, (A_1^1, I_3^1)), (A_9^0, (A_1^1, J_1^1)), (A_{10}^0, (A_1^1, J_1^1)), (A_9^0, (A_1^1, J_2^1)), (A_{10}^0, (A_1^1, J_2^1)), (A_9^0, (A_1^1, J_3^1)), (A_{10}^0, (A_1^1, J_3^1)), (L_2^1, L_3^1), (A_9^0, (A_1^1, L_1^1)), (A_{10}^0, (A_1^1, L_1^1)), (B_{12}^1, (A_1^0, A_6^0)), (B_{12}^1, (A_1^0, A_9^0)), (B_{27}^1, (A_1^0, A_6^0)), (B_{27}^1, (A_1^0, A_9^0)), (B_{29}^1, (A_1^0, A_6^0)), (B_{29}^1, (A_1^0, A_9^0)), (F_1^1, (A_1^0, A_6^0)), (F_2^1, (A_1^0, A_9^0)), (B_7^2, B_8^2), \dots, (F_2^2, (A_1^0, A_9^0)), (B_7^3, B_8^3), \dots, (F_2^3, (A_1^0, A_9^0))$	125

Tabela 23 – Cobertura alcançada com os critérios de teste em relação a versão *MtShopSafe*.

Critério	Total	Cobertura
<i>Todos-Nós-req</i>	61	100%
<i>Todos-Nós-red</i>	13	100%
<i>Todos-Nós</i>	254	85.82%
<i>Todas-Arestas-req</i>	60	100%
<i>Todas-Arestas-red</i>	12	100%
<i>Todas-Def-ident</i>	122	100%
<i>Todos-Usos-ident</i>	125	97.6%

7.2.3 iCardapio

O *iCardapio* possui uma arquitetura de dados diferente do *MtShop*, uma única base de dados é compartilhada entre os tenants, não há gerenciamento de autenticação com base nos dados armazenados e não houve preocupação em desenvolver estratégias para isolamento dos recursos entre os tenants. Conforme discutido no Capítulo 5, notou-se a existência de desvios nos fluxos de execução quando mais de uma versão é verificada simultaneamente. Entretanto,

Tabela 24 – Defeitos identificados na versão *MtShopSafe*.

Defeitos	Ocorrências
<i>Missing Command Error</i> (MCE)	10
<i>Control Flow Error</i> (CFE)	17
<i>Functionality Error</i> (FE)	18
<i>Implementation Error</i> (IE)	140
<i>Suspension Error</i> (SUE)	0
<i>Race Condition and Memory Inconsistency for tenant identification</i> (RC/MI)	0

somente com os testes funcionais não foi possível rastrear estruturalmente em quais elementos ocorrem os desvios. Na Tabela 25 apresentam-se os cenários de teste considerados incluindo uma breve descrição, perfil de usuário exigido e a página na qual são aplicados os testes. Somente os cenários *II* e *VI* requerem casos de teste para verificar as classes que definem um *Tenant/Restaurant* e *Product*.

Tabela 25 – Cenários de teste em relação ao *iCardapio*.

Cenário	Descrição (<perfil exigido>)[<página>]
<i>I</i>	Acesso a página para criar novo <i>Tenant/Restaurant</i> [Home]
<i>II</i>	Inserir novo <i>Tenant/Restaurant</i> [Home]
<i>III</i>	Acesso a página login [Login]
<i>IV</i>	Acesso a página para inserir novo <i>Product</i> (ADMIN)[Adicionar Produto]
<i>V</i>	Acesso a página para excluir um <i>Product</i> (ADMIN)[Home]
<i>VI</i>	Inserir novo <i>Product</i> (ADMIN)[Adicionar Produto]
<i>VII</i>	Excluir um <i>Product</i> (ADMIN)[Home]

No Apêndice C apontam-se os principais trechos de código envolvidos com a definição do tenant corrente partindo das requisições e os usos desse valor para diversas finalidades. A identificação do tenant corrente é usada para armazenar os registros de produtos com a chave do restaurante correto, ou seja, cada restaurante é um tenant cujo ID é usado para a relação com os registros de produtos. Isso fica mais claro na classe *MultiTenantSimpleJpaRepository* (Código-fonte 34), especificamente no método *setCurrentTenant()* (linhas 17-19) que é chamado pela maioria dos métodos nessa classe, os quais realizam operações com a base dados compartilhada. No *MtShop* essa definição é usada para selecionar os metadados a serem empregados para conexão com a base de dados correta, uma vez que a aplicação adota uma arquitetura de dados isolados por tenant.

Para garantir que esses elementos sejam cobertos por casos de testes mais direcionados, admite-se o modelo *PCFG_{MT}* e a aplicação da família de critérios, assim como nos estudos de caso anteriores. A definição do tenant envolvem várias classes, mas somente os usos explícitos da variável de identificação foram mapeados. Na classe *ICardapioTenantResolver* (A) o identificador é extraído do subdomínio e depois que as operações na classe *MultiTenantSimpleJpaRepository* (D) são executadas, a versão correspondente do *iCardapio* é fornecida para o usuário. Para isso, a classe *RestaurantController* (B) atende as requisições dos usuários provendo as páginas relativas para cada requisição. Na classe *Restaurant* há um campo chamado “*subdomain*”, trata-se do

identificador de cada tenant. No Apêndice C foram incluídas outras estruturas que colaboram para identificação dos tenants e a classe *Product* com o atributo *tenantId* para restringir os objetos ao tenant correspondente.

Na Figura 32 ilustra-se o modelo $PCFG_{MT}$ para o *iCardapio* considerando somente as estruturas das classes que colaboram com a definição ou tenham algum uso do identificador. Foram considerados também três tenants para essa aplicação com os identificadores “tenant 6”, “tenant 7” e “tenant 8”, portanto pode-se admitir que $N_d = \{t_6, t_7, t_8\}$. No *iCardapio* a definição de novos tenants pode ser realizada sem alterar o código da aplicação, diferente do *MtShop*, entretanto para manter a equivalência com as análises envolvendo o *MtShop* manteve-se o número de tenants. N_{post} contém os nós de A e N_d contém os nós provenientes das classes B, C e D. Assim, para os nós de B, C e D existem três camadas que representam os fluxos de execução para cada tenant.

Figura 32 – $PCFG_{MT}$ para a aplicação *iCardapio*.

Na Tabela 26 descrevem-se os principais elementos derivados do modelo $PCFG_{MT}$ e na Tabela 27 descrevem-se os elementos requeridos para cada critério de teste proposto considerando a versão *iCardapio*.

Execução dos testes automatizados com a versão *iCardapio*

Com a aplicação dos critérios da técnica funcional EPC e BVA, 86 casos de teste foram elaborados considerando as especificações das classes *Restaurant* (54) e *Product* (32). A execução dos testes automatizados das três versões fornecidas pelo *iCardapio* levaram a identificação de vários defeitos, conforme apresentado na Tabela 28. No Apêndice E apresentam-se os casos de teste e os tipos de defeitos identificados com a apoio dos testes automatizados.

As ocorrências de defeitos do tipo RC/MI referem-se aos desvios no fluxo de execução envolvendo *ICardapioTenantResolver* e *RestaurantController*, em que houve a execução da aresta (A_4^0, B_{12}^7) quando deveria ter sido a aresta (A_4^0, B_{12}^6) , ou seja, com base nos critérios foi possível identificar defeitos envolvendo a versão designada ao “tenant 6”. Na Tabela 29

Tabela 26 – Conjuntos obtidos pela aplicação do modelo de teste para a versão *iCardapio*.

$D = \{t_6, t_7, t_8\}$
$N = N_{post} \cup N_d$
$N_{post} = A_1^0, A_2^0, A_3^0, A_4^0$
$N_d = N_6 \cup N_7 \cup N_8$
$N_6 = B_1^6, B_2^6, B_3^6, B_4^6, B_5^6, B_6^6, B_7^6, B_8^6, B_9^6, B_{10}^6, B_{11}^6, B_{12}^6, B_{13}^6, B_{14}^6, C_1^6, D_1^6$
$N_7 = B_1^7, \dots, D_1^7$
$N_8 = B_1^8, \dots, D_1^8$
$N_{red} = A_1^0, B_{12}^6, B_{14}^6, B_{12}^7, B_{14}^7, B_{12}^8, B_{14}^8$
$N_{req} = A_4^0, B_1^6, B_2^6, B_3^6, B_4^6, B_{12}^6, B_{13}^6, B_1^7, \dots, B_{13}^7, B_1^8, \dots, B_{13}^8$
$E_6 = \{(B_5^6, B_6^6), (B_5^6, B_7^6), (B_7^6, B_8^6), (B_7^6, B_9^6), (B_9^6, B_{10}^6), (B_{10}^6, B_{11}^6), (B_9^6, B_{11}^6), (B_8^6, B_{11}^6), (B_{13}^6, B_{14}^6)\}$
$E_7 = \{(B_5^7, B_6^7), \dots, (B_{13}^7, B_{14}^7)\}$
$E_8 = \{(B_5^8, B_6^8), \dots, (B_{13}^8, B_{14}^8)\}$
$E_{post} = \{(A_1^0, A_2^0), (A_2^0, A_3^0), (A_2^0, A_4^0), (A_3^0, A_4^0)\}$
$E_{red} = \{(B_{12}^6, A_1^0), (B_{14}^6, A_1^0), (B_{12}^7, A_1^0), (B_{14}^7, A_1^0), (B_{12}^8, A_1^0), (B_{14}^8, A_1^0)\}$
$E_{req} = \{(A_4^0, B_1^6), (A_4^0, B_2^6), (A_4^0, B_3^6), (A_4^0, B_4^6), (A_4^0, B_5^6), (A_4^0, B_{12}^6), (A_4^0, B_{13}^6), (A_4^0, B_1^7), \dots, (A_4^0, B_{13}^7), (A_4^0, B_1^8), \dots, (A_4^0, B_{13}^8)\}$

Tabela 27 – Elementos requeridos pelos critérios de teste em relação a versão *iCardapio*.

Critério	Elementos requeridos	Total
<i>Todos-Nós-req</i>	$A_4^0, B_1^6, B_2^6, B_3^6, B_4^6, B_5^6, B_{12}^6, B_{13}^6, B_1^7, \dots, B_{13}^7, B_1^8, \dots, B_{13}^8$	22
<i>Todos-Nós-red</i>	$A_1^0, B_{12}^6, B_{14}^6, B_{12}^7, B_{14}^7, B_{12}^8, B_{14}^8$	7
<i>Todos-Nós</i>	$A_1^0, \dots, A_4^0, B_1^6, \dots, B_{14}^6, C_1^6, D_1^6, B_1^7, \dots, B_{14}^7, C_1^7, D_1^7, B_1^8, \dots, B_{14}^8, C_1^8, D_1^8$	52
<i>Todas-Arestas-req</i>	$(A_4^0, B_1^6), (A_4^0, B_2^6), (A_4^0, B_3^6), (A_4^0, B_4^6), (A_4^0, B_5^6), (A_4^0, B_{12}^6), (A_4^0, B_{13}^6), (A_4^0, B_1^7), \dots, (A_4^0, B_{13}^7), (A_4^0, B_1^8), \dots, (A_4^0, B_{13}^8)$	21
<i>Todas-Arestas-red</i>	$(B_{12}^6, A_1^0), (B_{14}^6, A_1^0), (B_{12}^7, A_1^0), (B_{14}^7, A_1^0), (B_{12}^8, A_1^0), (B_{14}^8, A_1^0)$	6
<i>Todas-Def-ident</i>	$(A_2^0, A_4^0), (A_3^0, A_4^0), (A_2^0, (A_4^0, B_1^6)), (A_3^0, (A_4^0, B_1^6)), (A_2^0, (A_4^0, B_2^6)), (A_3^0, (A_4^0, B_2^6)), (A_2^0, (A_4^0, B_3^6)), (A_3^0, (A_4^0, B_3^6)), (A_2^0, (A_4^0, B_4^6)), (A_3^0, (A_4^0, B_4^6)), (A_2^0, (A_4^0, B_5^6)), (A_3^0, (A_4^0, B_5^6)), (A_2^0, (A_4^0, B_{12}^6)), (A_3^0, (A_4^0, B_{12}^6)), (A_2^0, (A_4^0, B_{13}^6)), (A_3^0, (A_4^0, B_{13}^6)), (A_2^0, (A_4^0, B_1^7)), \dots, (A_3^0, (A_4^0, B_{13}^7)), (A_2^0, (A_4^0, B_1^8)), \dots, (A_3^0, (A_4^0, B_{13}^8))$	38
<i>Todos-Usos-ident</i>	$(A_1^0, (A_2^0, A_4^0)), (A_1^0, (A_3^0, A_4^0)), (A_2^0, (A_4^0, B_1^6)), (A_3^0, (A_4^0, B_1^6)), (A_2^0, (A_4^0, B_2^6)), (A_3^0, (A_4^0, B_2^6)), (A_2^0, (A_4^0, B_3^6)), (A_3^0, (A_4^0, B_3^6)), (A_2^0, (A_4^0, B_4^6)), (A_3^0, (A_4^0, B_4^6)), (A_2^0, (A_4^0, B_5^6)), (A_3^0, (A_4^0, B_5^6)), (A_2^0, (A_4^0, B_{12}^6)), (A_3^0, (A_4^0, B_{12}^6)), (A_2^0, (A_4^0, B_{13}^6)), (A_3^0, (A_4^0, B_{13}^6)), (C_1^6, C_2^6), (D_1^6, D_2^6), (A_2^0, (A_4^0, B_1^7)), \dots, (D_1^7, D_2^7), (A_2^0, (A_4^0, B_1^8)), \dots, (D_1^8, D_2^8)$	44

Tabela 28 – Defeitos identificados na versão *iCardapio*.

Defeitos	Ocorrências
<i>Missing Command Error (MCE)</i>	22
<i>Control Flow Error (CFE)</i>	0
<i>Functionality Error (FE)</i>	0
<i>Implementation Error (IE)</i>	104
<i>Suspension Error (SUE)</i>	0
<i>Race Condition and Memory Inconsistency for tenant identification (RC/MI)</i>	20

descrevem-se o total de elementos a serem cobertos pelos testes com base na família de critérios de teste proposta para o *iCardapio* e a cobertura alcançada com a execução dos testes paralelos.

Crítérios de teste podem ser definidos para dois propósitos principais: (i) apoiar a geração de casos de teste e (ii) apoiar a avaliação de casos de teste. No primeiro, a ideia é construir casos de teste com base nos critérios. Isso poderia ter sido feito ao empregar os critérios propostos. No

Tabela 29 – Cobertura alcançada com os critérios de teste em relação a versão *iCardapio*.

Critério	Total	Cobertura
<i>Todos-Nós-req</i>	22	100%
<i>Todos-Nós-red</i>	7	100%
<i>Todos-Nós</i>	52	92.31%
<i>Todas-Arestas-req</i>	21	100%
<i>Todas-Arestas-red</i>	6	100%
<i>Todas-Def-ident</i>	38	100%
<i>Todos-Usos-ident</i>	44	100%

segundo, a ideia é avaliar se um conjunto de teste é eficaz para testar uma aplicação segundo os requisitos dos critérios e isso foi realizado considerando as três versões instrumentadas das aplicações: (a) os cenários de uso das aplicações foram identificados; (b) posteriormente, os métodos que colaboram com a definição da variável de identificação dos tenants e seus usos computacionais, predicativos e redirecionais também foram identificados; (c) o modelo *PCFG_{MT}* foi aplicado assim como os critérios de teste propostos e a extração de elementos requeridos; (d) admitiu-se os casos de teste definidos previamente com apoio da técnica funcional de teste; (e) houve a execução dos testes automatizados de forma paralela para as versões designadas aos tenants com tais casos de teste e por fim, (f) houve a coleta dos resultados dessa execução.

Embora a utilização dos casos de teste provenientes da utilização da técnica funcional tenham coberto os critérios propostos, em relação às saídas obtidas com a execução dos testes paralelos, tais critérios de teste oferecem a visão do que foi coberto e podem ser empregados para melhorar essa cobertura, caso necessário. Nestes estudos de caso não houve a necessidade, provavelmente por que a utilização do teste funcional foi suficiente mas não existem garantias que em outras aplicações isso será mantido. Uma proposta futura é empregar casos de teste gerados de maneira ad-hoc e avaliar a cobertura, possivelmente a cobertura será mais baixa em relação aos dados reportados nestes estudos de caso e isso poderá exigir a criação de novos casos de teste. Em geral, os critérios propostos servem para avaliar a cobertura em relação às requisições, redirecionamentos e definições da variável de identificação do tenant em relação às versões disponibilizadas por uma solução multi-tenant. Os critérios propostos são úteis por conta do estabelecimento de um nível de confiabilidade em relação aos defeitos que levam aos desvios nos fluxos de execução. Os defeitos do tipo *RC/MI* foram identificados com o teste funcional mas em relação a estrutura não havia a rastreabilidade, nesse sentido um modelo de teste foi proposto para representar a dinamicidade das versões em termos de grafos com a finalidade de identificar os pontos potenciais para defeitos que precisariam ser cobertos pelos testes.

Uma das questões que motivam atingir a cobertura de um código sob teste é a necessidade de garantir que as partes do código mais bem testadas sejam aquelas com maior probabilidade de conter defeitos. A ideia é concentrar os testes em partes mais críticas, onde os defeitos são mais propensos a levar um resultado ruim aos clientes, por isso é importante desenvolver testes que não ignorem outras partes relevantes do código. A JaCoCo foi importante para obter métricas

de cobertura de código. Essa ferramenta foi utilizada para gerar as versões instrumentadas das aplicações, depois disso foi necessário implantá-las localmente em um servidor *Apache Tomcat*⁷. Os dados de execução dos testes automatizados ficaram armazenados no arquivo *jacoco.exec*, que é criado no mesmo diretório onde a aplicação se encontra. Para gerar o relatório de cobertura dos testes, bastou executar os comandos *mvn jacoco:restore-instrumented-classes* e *mvn jacoco:report*. Esse relatório foi analisado considerando os elementos requeridos para cada critério de teste proposto, gerando os dados de cobertura apresentados. Definir testes para as partes complexas do código é importante e mesmo que seja atingido 100% de cobertura, defeitos ainda podem existir no código. Uma alternativa válida é refatorar determinadas estruturas para torná-las menos complexas e aplicar critérios de teste explorando os mecanismos essenciais que tornam uma aplicação multi-tenant.

7.3 Considerações finais

Os serviços baseados na computação em nuvem estão cada vez mais presentes nas organizações, seja para melhorar seu *time to market*, competitividade e reduzir os custos operacionais. Serviços como plataformas de desenvolvimento de aplicações para nuvem são atraentes pois permitem que as empresas acessem rapidamente recursos de software sob demanda, evitam custos e a complexidade associada a compra, instalação, configuração e manutenção contínua de um data center e hardware dedicado. Do ponto de vista dos fornecedores de sistemas, o gerenciamento de múltiplas versões de aplicações que compartilham funcionalidades em locais dedicados muitas vezes é ineficiente. Assim, a adoção do modelo SaaS e a arquitetura multi-tenancy pode ser vista como uma alternativa. Embora muitos benefícios possam ser alcançados com essa arquitetura, existem muitos desafios técnicos que levam aos desenvolvedores implementarem determinados mecanismos que podem se diferenciar significativamente e isso pode abrir margens para defeitos potenciais. Dentre as preocupações com a qualidade dessas aplicações, deve-se garantir que em um cenário dinâmico de múltiplas organizações compartilhando uma única instância não ocorram desvios nos fluxos de execução. Portanto, pesquisas na área de teste de software nesse campo são fundamentais para apoiar a melhoria da qualidade desses sistemas.

Os estudos de caso ilustram a utilização dessas definições e servem para avaliar a abordagem de teste e a família de critérios propostas. Para isso, adotou-se um sistema multi-tenant desenvolvido ao longo das pesquisas, um e-commerce chamado *MtShop* e o *iCardapio*, um sistema simples mas que ilustra a utilização de uma base de dados totalmente compartilhada entre os tenants. Essas aplicações foram modificadas por conta da instrumentação no código para rastrear os caminhos de execução a serem cobertos. Nesses estudos de caso foram coletadas informações sobre a cobertura das arestas e nós em que ocorrem três tipos de uso da variável de identificação dos tenants: uso computacional, predicativo e o redirecional. Esse último refere-se ao uso do identificador para requisições e redirecionamentos que colaboram para novas definições

⁷ <http://tomcat.apache.org>

dessa variável. Além de investigar a cobertura, investigou-se a probabilidade de satisfação dos critérios e os defeitos revelados com o apoio dessa estratégia de teste. Os resultados desses estudos de caso foram promissores e a inovação gerada neste projeto concentra-se no fato da importância do tipo de teste para o contexto de aplicações desenvolvidas para a nuvem, dada a criticidade dos defeitos que podem ser identificados. Além disso, a maioria dos trabalhos na área de teste para esse tipo de aplicação são voltadas para requisitos não funcionais.

No próximo capítulo, apresenta-se uma sumarização das contribuições originadas com este trabalho, discutem-se as limitações da pesquisa, as lições aprendidas e trabalhos futuros.

CONCLUSÕES

A evolução da computação em nuvem vem promovendo uma revolução nas últimas décadas, tornando aplicações, infraestruturas e plataformas disponíveis como serviços sob demanda na Internet. SaaS é um dos modelos de entrega de serviços mais adotados neste segmento. SaaS é um modelo de aplicações residentes na infraestrutura da nuvem fornecidas como um serviço aos consumidores. A arquitetura multi-tenancy é um dos principais fatores que diferem os sistemas SaaS de aplicações web convencionais, pois permite que uma única instância da aplicação seja compartilhada entre múltiplos *tenants*, atendendo seus requisitos específicos por meio de customizações e compartilhamento de recursos computacionais. Com a crescente adoção desse modelo, estudos sobre a atividade de teste neste contexto têm sido intensificados. No entanto, existe uma carência por diretrizes sobre quais técnicas e critérios utilizar considerando diferentes objetivos para esse contexto. Assim, é necessário estimar a aplicabilidade, efetividade e custo dessas técnicas e critérios no campo de sistemas SaaS. Além disso, muitos autores argumentam que o teste tradicional de software não pode ser aplicado satisfatoriamente para testar aplicações no ambiente de nuvem. A razão disso deve-se às características implícitas dessas aplicações, como a alta capacidade de customização, ambiente dinâmico e arquitetura multi-tenancy. Mesmo assim, muitas similaridades presentes em aplicações convencionais podem ser encontradas em aplicações na nuvem, como: linguagens de programação, processamento paralelo e acesso às aplicações por meio de um browser.

Esta pesquisa apresentou uma abordagem de teste para sistemas SaaS multi-tenants que explora possíveis desvios nos fluxos de execução devido ao comportamento não determinístico, um desafio conhecido no campo de programação concorrente. Defende-se a tese de que essa abordagem é aplicável e auxilia na identificação de defeitos devido a falta de um isolamento adequado das sessões por tenant, afetando os frequentes tratamentos de requisições e redirecionamentos que ocorrem após a interceptação e definição de uma variável de identificação dos tenants, vital para esse modelo de aplicações.

Como apresentado no Capítulo 1, o objetivo geral desta tese foi desenvolver essa abordagem de teste para tais sistemas. Em razão do domínio ser uma área muito ampla, conforme reportado no Capítulo 4 com um Mapeamento Sistemático da Literatura, definiu-se uma classificação preliminar de defeitos para tornar possível a definição do escopo das pesquisas. A pesquisa teve início com a condução de estudos sobre os sistemas multi-tenants, considerando o background do aluno de doutorado, visando adequar o que era conhecido, sobre o teste de software para utilizar no contexto de sistemas desenvolvidos para nuvem. Assim, um tempo foi dedicado para analisar códigos de exemplos para possibilitar o mapeamento de conceitos e similaridades. Tanto o estudo desses exemplos quanto o levantamento bibliográfico serviram para o desenvolvimento de um sistema e-commerce multi-tenant, o *MtShop*. Essa aplicação encontra-se em processo de registro justo ao INPI (Instituto Nacional da Propriedade Industrial) via Agência USP de Inovação. Como a ideia inicial do projeto consistia em explorar a técnica de teste funcional, foram realizados dois estudos de caso que foram apresentados no Capítulo 5. No primeiro atentou-se em explorar quantitativamente os defeitos com apoio de testes automatizados com execução paralela e no segundo, o objetivo foi replicar o anterior levando em conta o *MtShop*, a execução de máquinas virtuais isoladas e uma análise exploratória qualitativa dos defeitos.

Embora os resultados desses experimentos tenham sido relevantes do ponto de vista da simulação do ambiente real em que ocorrem múltiplas requisições simultâneas, a rastreabilidade dos defeitos com base em uma visão estrutural mostrou-se necessária. Com base nessa premissa, foram definidos um modelo de testes chamado $PCFG_{MT}$ e um conjunto de critérios que exploram certas particularidades desse tipo de aplicação em função da identificação dos tenants e seus recursos. Com essa representação tornou-se possível explorar o teste estrutural e extrair os elementos requeridos para os testes, conforme discutido no Capítulo 6. Para avaliar a aplicabilidade da abordagem proposta foram aplicados três estudos de caso exploratórios, utilizando os mesmos sistemas utilizados nos estudos preliminares com a técnica de teste funcional. A diferença foi a aplicação dos critérios propostos, inclusão de novos casos de teste e a execução de versões instrumentadas para avaliar a satisfação dos critérios, a cobertura do código atingida e a capacidade de identificação de defeitos. Os detalhes dessa avaliação são apresentados no Capítulo 7.

Com os estudos de caso observou-se que a abordagem de teste proposta é promissora e aplicável e pode contribuir com a identificação de defeitos e auxiliar no desenvolvimento de aplicações com mais qualidade para nuvem. O tempo investido na condução desses estudos foi extenso devido à construção dos grafos, que foi realizada manualmente com a análise dos códigos-fonte das aplicações. Apesar disso, foi possível aplicar todos os passos da abordagem, desde a (i) definição dos cenários de teste; (ii) identificação dos métodos que colaboram com a definição da variável de identificação dos tenants e os usos computacionais, predicativos e redirecionais dessa variável; (iii) utilização do modelo $PCFG_{MT}$; (iv) aplicação dos critérios de teste propostos, bem como a extração dos elementos requeridos; (v) execução dos testes

automatizados de forma paralela para as versões designadas aos tenants e por fim, a (vi) coleta dos resultados dessa execução. Com isso, considera-se ter obtido evidências de que a abordagem é aplicável pois apoia a identificação de defeitos partindo das informações extraídas do modelo que de fato são essenciais para evitar a ocorrência de desvios nos fluxos de execução.

8.1 Contribuições e limitações da pesquisa

Esta tese apresenta várias contribuições para o teste de software no domínio de aplicações desenvolvidas para nuvem. A primeira contribuição deve-se ao mapeamento das oportunidades de pesquisas e a elaboração de uma taxonomia de defeitos partindo das características dos sistemas multi-tenants. Posteriormente, houve o desenvolvimento de uma solução SaaS para fins acadêmicos e evidências empíricas sobre a capacidade dos testes de aceitação ao serem automatizados e executados em paralelo para as versões específicas dos tenants.

A principal contribuição foi a definição de uma abordagem de teste. Esse estudo serviu de base para a proposição do modelo de teste e um conjunto de critérios de teste estruturais. O modelo de teste visa validar no código as operações que envolvem as requisições e redirecionamentos partindo das versões oferecidas por um sistema multi-tenant. Isso serve de base para a geração dos elementos requeridos, gerados a partir dos critérios estruturais. Estudos de caso exploratórios, utilizando sistemas multi-tenants foram conduzidos para avaliar a aplicabilidade da abordagem de teste proposta.

Algumas limitações deste trabalho foram identificadas e são listadas a seguir:

1. Os estudos de caso exploratórios foram conduzidos utilizando sistemas desenvolvidos em ambiente acadêmico e isso implica em uma ameaça à validade dos resultados, do ponto de vista da generalização. Apesar disso, um aspecto importante é que um dos sistemas não foi desenvolvido pelo aluno, o que é importante para evitar um viés durante a atividade de teste. Houve a preocupação em seguir a documentação e diretrizes de desenvolvimento recentes para obter uma solução mais próxima possível das reais, uma vez que não foram encontradas aplicações completas com os princípios mais importantes de multi-tenancy para a condução das pesquisas;
3. Foram utilizados somente dois sistemas para validação da abordagem de teste, uma vez que não havia outros sistemas disponíveis. Um número maior de sistemas com tamanhos e características diferenciadas possibilitaria uma generalização dos resultados do estudo. Para isso, uma ferramenta que automatize a abordagem se faz necessária;
4. O modelo de teste proposto considera apenas um tipo de uso diferente dos usos de variáveis previamente considerados em testes de programas convencionais. Essa limitação não permitiu testar outros tipos de comunicação. Estender a abordagem para abarcar os demais tipos de comunicação poderia revelar outros tipos de defeitos.

5. Geralmente as aplicações desenvolvidas para o ambiente de nuvem adotam algum framework que apoia os mecanismos mais importantes como: interceptação de requisições, utilização de múltiplas bases de dados, autenticação e persistência com diferentes níveis de compartilhamento. Portanto, parte da aplicação é vista como uma “caixa-preta” e muitas vezes considerado como suficientemente testado, mas parte disso é verdade. Um exemplo ilustrativo conhecido pelos desenvolvedores é a dificuldade em utilizar ou ser provavelmente um defeito no *Hibernate* quanto ao isolamento da sessões por tenant¹. Assim, uma limitação dos estudos foi não cobrir também defeitos em frameworks, uma vez que esses defeitos levam ao desenvolvimento de estratégias próprias.

8.2 Lições aprendidas

Durante a condução desta pesquisa algumas atividades não foram executadas como inicialmente planejadas e alguns desafios não previstos precisaram ser superados. No início da pesquisa, um dos maiores desafios do projeto foi com relação ao acesso aos sistemas multi-tenants. Esses sistemas geralmente são proprietários, cujo o código não é *open source* ou disponibilizado em repositórios como o *gitHub*. Após incessantes buscas por aplicações suficientemente completas para a proposta dos estudos, foram encontrados somente frameworks e partes isoladas de aplicações. Assim, a solução foi desenvolver uma aplicação seguindo práticas recentes de desenvolvimento e as documentações de frameworks. Um desafio que exigiu muito esforço, mas que sem a visão prática dos fundamentos desse tipo de aplicação talvez os resultados não teriam sido alcançados. As perspectivas de desenvolvedor e de testador foram colocadas à prova para gerar reflexões e definições que percorreram o andamento das pesquisas.

A necessidade de executar várias versões dos sistemas multi-tenants em paralelo com os testes automatizados foi outro desafio, o qual exigiu conhecimento para implementar as classes de teste com o *Selenium*. A dificuldade em executar o *Selenium Grid* implicou na implementação extensões para o JUnit para permitir a execução paralela e parametrizada dos testes desenvolvidos com o *Selenium*.

A instrumentação off-line do *MtShop* e o *iCardapio* com o *JaCoCo* não foi trivial, a documentação para utilização com *Maven*² foi considerada insuficiente e isso levou a necessidade de buscar apoio em fóruns de discussão.

8.3 Oportunidades futuras

Há diversas pesquisas que podem surgir a partir dos estudos conduzidos durante este doutorado e apresentados nesta tese. Um estudo mais imediato seria a aplicação da abordagem

¹ <https://hibernate.atlassian.net/browse/HHH-9766>

² <https://maven.apache.org/>

de teste em outros estudos de caso. Esta tarefa é fundamental para que a abordagem de teste seja efetivamente utilizada pela comunidade.

Uma variedade de estudos poderiam ser conduzidos com diversos objetivos, como: avaliação da efetividade da abordagem em revelar defeitos, que poderia ser medida pelo número de defeitos semeados e revelados; custo em revelar defeitos, medido pelo número de dados de teste utilizados; desempenho do sistema, medido pelo número de componentes que o software contém e os efetivamente usados para realizar determinadas tarefas, dentre outros. Uma possibilidade para a semeadura de defeitos seria, por exemplo, utilizar operadores de mutação.

Outro trabalho futuro poderia ser implementar a geração automatizada de dados de teste por meio de uma abordagem bio-inspirada, em conformidade com uma ferramenta que apoie a execução do teste. Um trabalho nessa direção foi publicado recentemente no qual o aluno participou como co-autor, trata-se de um abordagem utilizada em programas concorrentes para cobrir estruturas que envolvem passagem por mensagem e variável compartilhada (VILELA *et al.*, 2019).

Uma possibilidade é explorar outras técnicas de teste, como o teste baseados em modelos para considerar as evoluções frequentes nas versões destinadas aos tenants e as integrações. A investigação de mecanismos de predição de defeitos pode ser uma linha de pesquisa com resultados interessantes ao aplicar algoritmos de aprendizado de máquina para identificar os trechos de código com maior probabilidade de defeitos, levando em conta as definições e os usos subsequentes da variável de identificação identificados no modelo de teste proposto nesta tese.

8.4 Publicações

Nesta seção são apresentadas as publicações dos autores desta pesquisa durante o período de doutorado, incluindo um artigo completo submetido em periódico e os artigos completos publicados em anais de congressos (full papers).

8.4.1 Trabalho submetido em periódico

- **PINTO, Victor Hugo S. C.; VILELA, R. F.; OLIVEIRA, R. R.; SOUZA, S. R. S. . *Investigating the impact of the automated testing for multitenant SaaS systems*. *Software: Practice and Experience*, 2019.** Esse artigo refere-se ao segundo estudo envolvendo a técnica funcional de teste em que foram consideradas duas aplicações: *iCardapio* e o *MtShop*. Adotou-se a execução paralela dos testes automatizados em máquinas virtuais isoladas com o apoio de um TaaS.

8.4.2 Trabalhos completos publicados em anais de congressos

- **PINTO, Victor Hugo S. C.; SOUZA, S. R. S.; SOUZA, P. S. L. . A Preliminary Fault Taxonomy for Multi-tenant SaaS systems.** In: *19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, 2019, Lanarca, Cyprus. Proceedings of 19th Annual IEEE/ACM International Symposium in Cluster, Cloud, and Grid Computing (CCGrid 2019), 2019. p. 178-187.* Essa publicação divulga a taxonomia de defeitos proposta para sistemas SaaS multi-tenants, uma contribuição importante para definição de estratégias de teste mais direcionadas aos desafios de desenvolver tais aplicações com qualidade.
- **PINTO, Victor Hugo S. C.; OLIVEIRA, R. R.; VILELA, R. F.; SOUZA, S. R. S. . Evaluating the User Acceptance Testing for Multi-tenant Cloud Applications.** In: *8th International Conference on Cloud Computing and Services Science, 2018, Funchal. Proceedings of the 8th International Conference on Cloud Computing and Services Science. Portugal: SCITEPRESS - Science and Technology Publications, 2018. p. 47-56.* Essa publicação divulga um experimento conduzido neste trabalho envolvendo testes funcionais. A ideia principal foi investigar o impacto do número de tenants em relação ao número de defeitos quando as versões fornecidas pelo sistema SaaS são executadas simultaneamente por meio de testes automatizados.
- **PINTO, Victor Hugo S. C.; LUZ, H. J. F. ; OLIVEIRA, R. R.; SOUZA, P. S. L.; SOUZA, S. R. S. . A Systematic Mapping Study on the Multi-tenant Architecture of SaaS Systems.** In: *28th International Conference on Software Engineering & Knowledge Engineering (SEKE), 2016, San Francisco, CA. Proceedings of 28th International Conference on Software Engineering & Knowledge Engineering. Pittsburgh, USA: KSI Research Inc. and Knowledge Systems Institute Graduate School, 2016. p. 396-401.* Essa publicação divulga o mapeamento sistemático que foi fundamental para obter um panorama da área, identificar as tendências e oportunidades de pesquisa e principalmente apoiar a definição dos elementos fundamentais que tornam uma aplicação multi-tenant.
- **PINTO, Victor Hugo S. C.; SOUZA, P. S. L.; SOUZA, S. R. S. .Estudo e Definição de uma Estratégia Sistemática de Teste para Aplicações na Nuvem.** In: *WTDSOft (V Workshop de Teses e Dissertações do CBSOft), 2015, Belo Horizonte, MG. Anais do V Workshop de Teses e Dissertações do CBSOft (WTDSOft). Porto Alegre: SBC, 2015. v. 01.* Essa publicação divulga a proposta inicial de uma abordagem de teste para sistemas multi-tenants. Os direcionamentos recorrentes da discussão desse trabalho serviram como base para definição de várias etapas deste doutorado.

8.4.2.1 Outros trabalhos completos publicados em anais de congressos

- VILELA, R. F.; PINTO, Victor Hugo S. C.; COLANZI, T. E.; SOUZA, R. S. S. . **Bio-inspired optimization of test data generation for concurrent programs**. In: *11th Symposium on Search-Based Software Engineering, 2019, Tallinn Estonia. Proceedings of 11th Symposium on Search-Based Software Engineering (SSBSE 2019)*, 2019.

REFERÊNCIAS

- ABUBAKR, T. **Cloud app vs. web app: Understanding the differences**. 2010. Disponível em: <http://www.techrepublic.com/blog/the-enterprise-cloud/cloud-app-vs-web-app-understanding-the-differences/>. Citado na página 59.
- AHMAD, A. A.-S.; ANDRAS, P. Measuring and testing the scalability of cloud-based software services. In: IEEE. [S.l.], 2018. (2018 Fifth International Symposium on Innovation in Information and Communication Technology (ISIICT)), p. 1–8. Citado na página 43.
- AKERELE, O.; RAMACHANDRAN, M.; DIXON, M. Testing in the cloud: Strategies, risks and benefits. In: MAHMOOD, Z.; SAEED, S. (Ed.). Springer London, 2013, (Software Engineering Frameworks for the Cloud Computing Paradigm). p. 165–185. Disponível em: http://dx.doi.org/10.1007/978-1-4471-5031-2_8. Citado na página 45.
- ALKHATIB, H.; FARABOSCHI, P.; FRACHTENBERG, E.; KASAHARA, H.; LANGE, D.; LAPLANTE, P.; MERCHANT, A.; MILOJICIC, D.; SCHWAN, K. “**IEEE CS 2022 report (draft)**”. [S.l.], 2014. Citado nas páginas 87, 89 e 91.
- ALMORSY, M.; GRUNDY, J.; IBRAHIM, A. S. Collaboration-based cloud computing security management framework. In: **IEEE Int. Conf. on Cloud Computing**. [S.l.: s.n.], 2011. p. 364–371. Citado na página 88.
- _____. Tossma: A tenant-oriented saas security management architecture. In: **IEEE 5th Int. Conf. on Cloud Computing**. [S.l.: s.n.], 2012. p. 981–988. Citado na página 88.
- ARMBRUST, M.; FOX, A.; GRIFFITH, R.; JOSEPH, A. D.; KATZ, R.; KONWINSKI, A.; LEE, G.; PATTERSON, D.; RABKIN, A.; STOICA, I.; ZAHARIA, M. A view of cloud computing. **Commun. ACM**, ACM, New York, NY, USA, v. 53, n. 4, p. 50–58, 2010. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/1721654.1721672>. Citado nas páginas 27, 28 e 50.
- ARORA, V.; BHATIA, R.; SINGH, M. A systematic review of approaches for testing concurrent programs. **Concurr. Comput. : Pract. Exper.**, John Wiley and Sons Ltd., Chichester, UK, v. 28, n. 5, p. 1572–1611, abr. 2016. Citado na página 115.
- BADGER, L.; GRANCE, T.; R., P.-C.; VOAS, e. J. **Cloud Computing Synopsis and Recommendations**. [S.l.], 2012. Citado nas páginas 48, 49 e 50.
- BAI, X.; LI, M.; CHEN, B.; TSAI, W.-T.; GAO, J. Cloud testing tools. In: IEEE. [S.l.], 2011. (Service Oriented System Engineering (SOSE), 2011 IEEE 6th International Symposium on), p. 1–12. Citado nas páginas 45 e 116.
- BARBOSA Élen F.; MALDONADO, J. C.; VINCENZI, A. M. R.; DELAMARO, M. E.; SOUZA, S. do Rocio Senger de; JINO, M. Introdução ao teste de software. In: **Minicurso apresentado ao XIV Simpósio Brasileiro de Engenharia de Software (SBES2000)**. João Pessoa, PB: [s.n.], 2000. p. 330–378. Citado nas páginas 29, 36 e 39.

- BEIZER, B. **Software testing techniques**. 2. ed. New York, NY, USA: Van Nostrand Reinhold Co., 1990. ISBN 0-442-20672-0. Citado na página 37.
- BEZEMER, C.-P.; ZAIDMAN, A. Multi-tenant saas applications: Maintenance dream or nightmare? In: **Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)**. New York, NY, USA: ACM, 2010. (IWPSE-EVOL '10), p. 88–92. ISBN 978-1-4503-0128-2. Disponível em: <<http://doi.acm.org/10.1145/1862372.1862393>>. Citado na página 52.
- BOBROWSKI, S. **The Force.com Multitenant Architecture**. 2016. Disponível em: <https://developer.salesforce.com/page/Multi_Tenant_Architecture>. Acesso em: 05/07/2019. Citado nas páginas 28 e 145.
- CHANA, I.; CHAWLA, P. Testing perspectives for cloud-based applications. In: **Software Engineering Frameworks for the Cloud Computing Paradigm**. [S.l.]: Springer, 2013. p. 145–164. Citado nas páginas 28 e 145.
- CHONG, F.; CARRARO, G. Architecture strategies for catching the long tail. In: MSDN LIBRARY. [S.l.], 2006. (Microsoft Corporation), p. 9–10. Citado nas páginas 50, 52, 53, 54, 55 e 61.
- CHONG, F.; CARRARO, G.; WOLTER, R. Multi-tenant data architecture. **MSDN Library, Microsoft Corporation**, p. 14–30, 2006. Citado na página 55.
- COPELAND, L. **A practitioner's guide to software test design**. [S.l.]: Artech House, 2004. Citado na página 38.
- DELAMARO, M. E.; MALDONADO, J. C. Proteum-A tool for the assessment of test adequacy for C programs. In: **Proceedings of the Conference on Performability in Computing Systems (PCS 96)**. New Brunswick, NJ: [s.n.], 1996. p. 79–95. Citado na página 42.
- DELAMARO, M. E.; MALDONADO, J. C.; JINO, M. **Introdução ao Teste de Software**. [S.l.]: Elsevier, 2007. v. 394. (Campus, v. 394). Citado nas páginas 36, 37, 38, 39, 40 e 151.
- DILLON, T.; WU, C.; CHANG, E. Cloud computing: issues and challenges. In: IEEE. **Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on**. [S.l.], 2010. p. 27–33. Citado na página 50.
- DOMENICO, M. D. Multilayer modeling and analysis of human brain networks. **Giga Science**, Oxford University Press, v. 6, n. 5, p. gix004, 2017. Citado na página 132.
- DOMINGO, E. J.; NINO, J. T.; LEMOS, A. L.; LEMOS, M. L.; PALACIOS, R. C.; BERBÍS, J. M. G. Cloudio: a cloud computing-oriented multi-tenant architecture for business information systems. In: IEEE. [S.l.], 2010. (2010 IEEE 3rd International Conference on Cloud Computing), p. 532–533. Citado na página 56.
- GAMMA, E. **Design patterns: elements of reusable object-oriented software**. [S.l.]: Pearson Education India, 1995. Citado na página 94.
- GAO, J.; BAI, X.; TSAI, W.-T. Cloud testing-issues, challenges, needs and practice. **Software Engineering: An International Journal**, v. 1, n. 1, p. 9–23, 2011. Citado nas páginas 45, 75, 77, 79, 80, 90 e 145.

- GAO, J.; BAI, X.; TSAI, W.-T.; UEHARA, T. Testing as a Service (TaaS) on Clouds. In: . [S.l.: s.n.], 2013. (Service Oriented System Engineering (SOSE), 2013 IEEE 7th International Symposium on), p. 212–223. Citado nas páginas 45, 76, 80 e 81.
- GARG, D.; DATTA, A. Parallel execution of prioritized test cases for regression testing of web applications. In: AUSTRALIAN COMPUTER SOCIETY, INC. [S.l.], 2013. (Proceedings of the Thirty-Sixth Australasian Computer Science Conference-Volume 135), p. 61–68. Citado na página 45.
- GHAG, S. **Software Validations of Application Deployed on Windows Azure**. Infosys Whi-tepaper, 2012. Disponível em: <www.infosys.com/cloud/>. Citado na página 81.
- GHUMMAN, W. A.; LASSIG, J. Verification requirements for secure and reliable cloud compu-ting. In: IEEE. **Cloud and Green Computing (CGC), 2013 Third International Conference on**. [S.l.], 2013. p. 143–150. Citado na página 75.
- GOTTLIEB, A.; ALMASI, G. **Highly parallel computing**. [S.l.]: Benjamin/Cummings Redwood City, CA, 1989. Citado na página 29.
- HE, Q.; HAN, J.; YANG, Y.; GRUNDY, J.; JIN, H. Qos-driven service selection for multi-tenant saas. In: **Int. Conf. on Cloud Computing**. [S.l.: s.n.], 2012. p. 566–573. Citado na página 88.
- HECHT, M. S. **Flow analysis of computer programs**. [S.l.]: Elsevier Science Inc., 1977. Citado na página 41.
- HOFFMAN, D. Using oracles in test automation. In: . [S.l.: s.n.], 2001. (Proceedings of Pacific Northwest Software Quality Conference), p. 90–117. Citado na página 35.
- HU, K.; LEI, L.; TSAI, W.-T. Multi-tenant verification-as-a-service (vaas) in a cloud. **Simulation Modelling Practice and Theory**, Elsevier, v. 60, p. 122–143, 2016. Citado na página 92.
- IEEE. IEEE standard glossary of software engineering terminology. In: **Standard 610.12**. [S.l.]: IEEE Computer Society Press, 1990. Citado na página 36.
- INCKI, K.; ARI, I.; SÖZER, H. A survey of software testing in the cloud. In: IEEE. **Software Se-curity and Reliability Companion (SERE-C), 2012 IEEE Sixth International Conference on**. [S.l.], 2012. p. 18–23. Citado na página 75.
- JOHNSEN, E. B.; OWE, O. An asynchronous communication model for distributed concurrent objects. **Software & Systems Modeling**, Springer, v. 6, n. 1, p. 39–58, 2007. Citado na página 91.
- JOVANOVIĆ, I. Software testing methods and techniques. **Transactions on Internet Research**, v. 26, 2009. Citado na página 29.
- KABBEDIJK, J.; BEZEMER, C.-P.; JANSEN, S.; ZAIDMAN, A. Defining multi-tenancy: A systematic mapping study on the academic and the industrial perspective. **Journal of Systems and Software**, v. 100, n. 0, p. 139 – 148, 2015. ISSN 0164-1212. Disponível em: <<http://www.sciencedirect.com/science/article/pii/S0164121214002313>>. Citado nas páginas 28, 53 e 54.
- KALE, S. S.; BORHADE, R. H. Development of multitenant saas framework at single instance and with zero effort multitenancy. In: IEEE. **Advances in Computing, Communications and Informatics (ICACCI)**. [S.l.], 2013. p. 834–839. Citado na página 86.

- KANER, C. **An introduction to scenario testing**. [S.l.], 2003. Citado nas páginas 140 e 147.
- KANG, S.; MYUNG, J.; YEON, J.; HA, S.-w.; CHO, T.; CHUNG, J.-m.; LEE, S.-g. A general maturity model and reference architecture for saas service. In: **Database Systems for Advanced Apps**. [S.l.: s.n.], 2010. p. 337–346. Citado na página 82.
- KITCHENHAM, B. **Procedures for Performing Systematic Reviews**. [S.l.], 2004. Citado na página 81.
- KREBS, R.; MOMM, C.; KOUNEV, S. Architectural concerns in multi-tenant saas applications. In: . Porto, Portugal: [s.n.], 2012. (Proceedings of the 2nd International Conference on Cloud Computing and Services Science (CLOSER), v. 12), p. 426–431. Citado na página 56.
- KREBS, R.; WERT, A.; KOUNEV, S. Multi-tenancy performance benchmark for web application platforms. In: **Int. Conf. on Web Eng.** [S.l.: s.n.], 2013. p. 424–438. Citado na página 87.
- LAPLANTE, P.; ZHANG, J.; VOAS, J. *et al.* What's in a name? distinguishing between saas and soa. **It Professional**, IEEE, v. 10, n. 3, p. 46–50, 2008. Citado na página 60.
- LEWIS, B.; BERG, D. J. **Threads Primer: A Guide to Multithreaded Programming**. Upper Saddle River, NJ, USA: Prentice Hall Press, 1995. Citado na página 100.
- LINKMAN, S.; VINCENZI, A. M. R.; MALDONADO, J. An evaluation of systematic functional testing using mutation testing. In: THE IEE. **7th International Conference on Empirical Assessment in Software Engineering (EASE)**. [S.l.], 2013. Citado na página 39.
- LNĚNÍČKA, M.; ČAPEK, J. *et al.* Classification and evaluation of cloud-based testing tools: The case study of web applications' security testing. **Acta Informatica Pragensia**, Acta Informatica Pragensia, v. 7, n. 1, p. 40–57, 2018. Citado na página 43.
- MAENHAUT, P.-J.; MOENS, H.; DECAT, M.; BOGAERTS, J.; LAGAISSÉ, B.; JOOSEN, W.; ONGENAE, V.; TURCK, F. D. Characterizing the performance of tenant data management in multi-tenant cloud authorization systems. In: **Network Operations and Management Symposium (NOMS)**, IEEE. [S.l.: s.n.], 2014. p. 1–8. Citado na página 88.
- MAHMOOD, Z.; SAEED, S. **Software engineering frameworks for the cloud computing paradigm**. [S.l.]: Springer, 2013. Citado nas páginas 50, 58, 59, 76, 77, 80, 81 e 90.
- MAKKI, M.; LANDUYT, D. V.; JOOSEN, W. Automated workflow regression testing for multi-tenant saas: integrated support in self-service configuration dashboard. In: ACM. [S.l.], 2016. (Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation), p. 70–73. Citado na página 92.
- MALDONADO, J. C. **Critérios Potenciais Usos: Uma Contribuição ao Teste Estrutural de Software**. Tese (Doutorado) — DCA/FEEC/UNICAMP, Campinas, SP, Brasil, 1991. Citado na página 41.
- MANDUCA, A. M.; MUNSON, E. V.; FORTES, R. P.; PIMENTEL, M. G. C. A nonintrusive approach for implementing single database, multitenant services from web applications. In: **29th ACM Symposium on Applied Computing**. [S.l.: s.n.], 2014. p. 751–756. Citado nas páginas 86, 105 e 106.
- MARINOS, A.; BRISCOE, G. Community cloud computing. In: **Cloud Computing**. [S.l.]: Springer, 2009. p. 472–484. Citado na página 51.

MCCABE, T. J. A complexity measure. **Software Engineering, IEEE Transactions on**, IEEE, n. 4, p. 308–320, 1976. Citado na página 40.

MELL, P.; GRANCE, T. **The nist definition of cloud computing - Recommendations of the National Institute of Standards and Technology**. [S.l.], 2011. Disponível em: <<http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf>>. Citado nas páginas 27, 28, 49 e 50.

MOON, H. J.; HACIGUMUS, H.; CHI, Y.; HSIUNG, W.-P. Swat: A lightweight load balancing method for multitenant databases. In: **16th Int. Conf. on Extending Database Technology**. [S.l.: s.n.], 2013. p. 65–76. Citado na página 87.

MORAKOS, P.; MELIONES, A. Design and implementation of a cloud saas framework for multi-tenant applications. In: **5th Int. Conf. on Information, Intelligence, Systems and Applications**. [S.l.: s.n.], 2014. p. 273–278. Citado na página 86.

MYERS, G. J.; SANDLER, C. **The Art of Software Testing**. [S.l.]: John Wiley & Sons, 2004. ISBN 0471469122. Citado nas páginas 29, 35, 36, 37, 38 e 40.

NAM, T.; YEOM, K. Ontology model to support multi-tenancy in software as a service environment. In: **Int. Conf. on Future Internet of Things and Cloud**. [S.l.: s.n.], 2014. p. 146–151. Citado na página 86.

OSELIO, B.; KULESZA, A.; HERO, A. O. Multi-layer graph analytics for social networks. In: **IEEE. 2013 5th IEEE International Workshop on Computational Advances in Multi-Sensor Adaptive Processing (CAMSAP)**. [S.l.], 2013. p. 284–287. Citado na página 132.

OSTRAND, T. J.; BALCER, M. J. The category-partition method for specifying and generating functional tests. **Commun. ACM**, ACM, New York, NY, USA, v. 31, n. 6, p. 676–686, jun. 1988. Citado na página 35.

OUTSYSTEMS. **How to Build a Multi-tenant Application**. 2019. Disponível em: <https://success.outsystems.com/Support/Enterprise_Customers/Maintenance_and_Operations/How_to_Build_a_Multi-tenant_Application>. Acesso em: 03/06/2019. Citado na página 57.

PARVEEN, T.; TILLEY, S. When to migrate software testing to the cloud? In: **IEEE. Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on**. [S.l.], 2010. p. 424–427. Citado na página 77.

PATIKIRIKORALA, T.; KUMARA, I.; COLMAN, A.; HAN, J.; WANG, L.; WEERASIRI, D.; RANASINGHE, W. Dynamic performance management in multi-tenanted business process servers using nonlinear control. In: **Int. Conf. on Service-Oriented Computing**. [S.l.: s.n.], 2012. p. 206–221. Citado na página 87.

PERREY, R.; LYCETT, M. Service-oriented architecture. In: **IEEE. Applications and the Internet Workshops, 2003. Proceedings. 2003 Symposium on**. [S.l.], 2003. p. 116–119. Citado na página 51.

PINTO, V. H. S. C.; LUZ, H. J. F.; OLIVEIRA, R. R.; SOUZA, P. S. L.; SOUZA, S. R. S. A systematic mapping study on the multi-tenant architecture of saas systems. In: **28th International Conference on Software Engineering & Knowledge Engineering (SEKE)**. [S.l.: s.n.], 2016. p. 396–401. Citado nas páginas 28, 29, 31, 81, 90 e 105.

- PINTO, V. H. S. C.; OLIVEIRA, R. R.; VILELA, R. F.; SOUZA, S. R. S. Evaluating the User Acceptance Testing for Multi-tenant cloud applications. In: **Proceedings of the 8th International Conference on Cloud Computing and Services Science (CLOSER)**. [S.l.: s.n.], 2018. p. 47–56. Citado nas páginas 31 e 105.
- PINTO, V. H. S. C.; SOUZA, S. R. S.; SOUZA, P. S. L. A Preliminary Fault Taxonomy for Multi-tenant SaaS systems. In: **19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2019)**. [S.l.: s.n.], 2019a. p. 178–187. Citado nas páginas 31, 91 e 96.
- PINTO, V. H. S. C.; VILELA, R. F.; OLIVEIRA, R. R.; SOUZA, S. R. S. Investigating the impact of the automated testing for multi-tenant SaaS systems. In: *Journal of Software: Practice and Experience* (2019), (To appear). 2019b. Citado na página 31.
- PRESSMAN, R. S. **Engenharia de Software - Uma Abordagem Profissional**. Sétima edição. [S.l.]: McGraw Hill, 2010. Citado nas páginas 37, 38 e 40.
- PRIYADHARSHINI, V.; MALATHI, A. **Survey on software testing techniques in cloud computing**. [S.l.], 2014. Citado na página 91.
- PUSULURI, N. R. **Software Testing Concepts And Tools**. [S.l.]: Dreamtech Press, 2006. Citado na página 43.
- RAMACHANDRAN, L.; NARENDRA, N. C.; PONNALAGU, K. Dynamic provisioning in multi-tenant service clouds. **Service Oriented Computing and Applications**, Springer, v. 6, n. 4, p. 283–302, 2012. Citado na página 87.
- RAPPS, S.; WEYUKER, E. J. Data flow analysis techniques for test data selection. In: IEEE COMPUTER SOCIETY PRESS. **Proceedings of the 6th international conference on Software engineering**. [S.l.], 1982. p. 272–278. Citado na página 41.
- _____. Selecting software test data using data flow information. **Software Engineering, IEEE Transactions on**, IEEE, n. 4, p. 367–375, 1985. Citado na página 41.
- RIMAL, B. P.; CHOI, E.; LUMB, I. A taxonomy and survey of cloud computing systems. In: IEEE. **INC, IMS and IDC, 2009. NCM'09. Fifth International Joint Conference on**. [S.l.], 2009. p. 44–51. Citado nas páginas 28 e 50.
- RIUNGU, L. M.; TAIPALE, O.; SMOLANDER, K. Software testing as an online service: Observations from practice. In: IEEE. **Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on**. [S.l.], 2010. p. 418–423. Citado na página 76.
- ROPER, M. **Software Testing**. [S.l.]: McGraw-Hill Book Company Europe, 1994. Citado na página 38.
- RU, J.; GRUNDY, J.; KEUNG, J. Software engineering for multi-tenancy computing challenges and implications. In: **Int. Work. on Innovative Soft. Dev. Methodologies and Practices**. [S.l.: s.n.], 2014. p. 1–10. Citado na página 82.
- RU, J.; KEUNG, J. An empirical investigation on the simulation of priority and shortest-job-first scheduling for cloud-based software systems. In: IEEE. [S.l.], 2013. (2013 22nd Australian Software Engineering Conference), p. 78–87. Citado nas páginas 27, 47 e 94.

SALEH, E.; SIANIPAR, J.; TAKOUNA, I.; MEINEL, C. Secplace: A security-aware placement model for multi-tenant SaaS environments. In: **Int. Conf. on Ubiquitous Intell. and Comp.** [S.l.: s.n.], 2014. p. 596–602. Citado na página 88.

SALESFORCE. **The Force.com Multitenant Architecture - Understanding the Design of Salesforce.com's Internet Application Development Platform**. [S.l.], 2008. 1-14 p. Citado nas páginas 52, 54, 59, 91 e 98.

SARASWATHI, M.; BHUVANESWARI, T. Multitenant saas model of cloud computing: Issues and solutions. In: IEEE. **Communication and Network Technologies (ICCNT), 2014 International Conference on**. [S.l.], 2014. p. 27–32. Citado nas páginas 75 e 88.

SARMANHO, F. S.; SOUZA, P. S.; SOUZA, S. R.; SIMÃO, A. S. Structural testing for semaphore-based multithread programs. In: SPRINGER. **International Conference on Computational Science**. [S.l.], 2008. p. 337–346. Citado na página 100.

SELENIUM. “**Docs: Sauce Connect**”. 2017. Disponível em: <<https://docs.saucelabs.com/reference/sauce-connect/>>. Citado na página 118.

_____. **Test Automation for Web Applications**. 2019. Disponível em: <https://www.seleniumhq.org/docs/01_introducing_selenium.jsp>. Acesso em: 15/06/2019. Citado na página 44.

SENGUPTA, B.; ROYCHOUDHURY, A. Engineering multi-tenant software-as-a-service systems. In: . New York, NY, USA: ACM, 2011. (Proceedings of the 3rd International Workshop on Principles of Engineering Service-Oriented Systems), p. 15–21. ISBN 978-1-4503-0591-4. Disponível em: <<http://doi.acm.org/10.1145/1985394.1985397>>. Citado nas páginas 82, 86 e 93.

SIMÃO, A.; PETRENKO, A. Generating checking sequences for partial reduced finite state machines. In: **Proceedings of the 20th IFIP TC 6/WG 6.1 International conference on Testing of Software and Communicating Systems: 8th International Workshop, TestCom '08 / FATES '08**. Berlin, Heidelberg: Springer-Verlag, 2008. Citado na página 40.

SOUZA, P. L. de; SAWABE, E. T.; SIMÃO, A. da S.; VERGILIO, S. R.; SOUZA, S. d. R. S. de. Valipvm-a graphical tool for structural testing of pvm programs. In: SPRINGER. **European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting**. [S.l.], 2008. p. 257–264. Citado na página 29.

SOUZA, P. S.; SOUZA, S. R.; ZALUSKA, E. Structural testing for message-passing concurrent programs: an extended test model. **Concurrency and Computation**, 2014. Citado na página 95.

SOUZA, P. S.; SOUZA, S. S.; ROCHA, M. G.; PRADO, R. R.; BATISTA, R. N. Data flow testing in concurrent programs with message passing and shared memory paradigms. **Procedia Computer Science**, Elsevier, v. 18, p. 149–158, 2013. Citado na página 29.

STITC. **Software Testing in the Cloud (STITC)**. 2013. Disponível em: <<http://www.stitc.org/>>. Citado na página 76.

SUMALATHA, M.; PARTHIBAN, M. SaaS multitenant performance testing over social networks. **International Journal of Enterprise Network Management**, Inderscience Publishers (IEL), v. 9, n. 3-4, p. 234–250, 2018. Citado na página 92.

- SUN, H.; ZHAO, T.; TANG, Y.; LIU, X. A qos-aware load balancing policy in multi-tenancy environment. In: **8th Int. Symposium on Service Oriented System Engineering**. [S.l.: s.n.], 2014. p. 140–147. Citado na página 87.
- SURESHKUMAR, D.; KANNAN, R. J.; PURNIEMAA, P. Multi tenancy deploy model and issues in saas: A survey. **International Journal of Computer Science & Applications (TIJCSA)**, v. 2, n. 08, 2013. Citado nas páginas 58 e 90.
- TAI, S.; NIMIS, J.; LENK, A.; KLEMS, M. Cloud service engineering. In: **ACM. Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2**. [S.l.], 2010. p. 475–476. Citado nas páginas 27 e 48.
- TRUYEN, E.; CARDOZO, N.; WALRAVEN, S.; VALLEJOS, J.; BAINOMUGISHA, E.; GÜNTHER, S.; D'HONDT, T.; JOOSEN, W. Context-oriented programming for customizable SaaS applications. In: **27th ACM symposium on applied computing**. [S.l.: s.n.], 2012. p. 418–425. Citado nas páginas 85 e 86.
- TSAI, W.-T.; LI, Q.; COLBOURN, C. J.; BAI, X. Adaptive fault detection for testing tenant applications in multi-tenancy saas systems. In: **IEEE**. [S.l.], 2013. (Cloud Engineering (IC2E), 2013 IEEE International Conference on), p. 183–192. Citado nas páginas 82, 87, 88, 91 e 93.
- TSAI, W.-T.; SHAO, Q.; HUANG, Y.; BAI, X. Towards a scalable and robust multi-tenancy SaaS. In: **Asia-Pacific Symp. on Internetware**. [S.l.: s.n.], 2010. p. 8. Citado na página 88.
- TSAI, W.-T.; SUN, X. SaaS multi-tenant application customization. In: **IEEE 7th Int. Symposium on Service Oriented System Engineering**. [S.l.: s.n.], 2013. p. 1–12. Citado na página 87.
- VAQUERO, L. M.; RODERO-MERINO, L.; CACERES, J.; LINDNER, M. A break in the clouds. **ACM SIGCOMM Computer Communication Review**, v. 39, n. 1, p. 50, 2008. ISSN 01464833. Disponível em: <<http://dl.acm.org/citation.cfm?id=1496091.1496100>>. Citado nas páginas 27 e 48.
- VASHISTHA, A.; AHMED, P. Saas multi-tenancy isolation testing challenges and issues. **International Journal of Soft Computing and Engineering**, 2012. Citado nas páginas 58 e 90.
- VERGÍLIO, S. R.; SOUZA, S. d. R. S. d.; SOUZA, P. S. L. d. Coverage testing criteria for message-passing parallel programs. **Digest Papers**, 2005. Citado nas páginas 29 e 132.
- VIJAYANATHAN, N.; SHUBHA, B. **The Challenges Associated with SaaS Testing**. Infosys Whitepaper, 2011. Disponível em: <<http://www.infosys.com/IT-services/independent-validation-testing-services/white-papers/Documents/saas-testing.pdf>>. Citado nas páginas 58, 59, 60 e 76.
- VILELA, R. F.; PINTO, V. H. S. C.; COLANZI, T. E.; SOUZA, S. R. S. Bio-inspired optimization of test data generation for concurrent programs. In: **In: 11th Symposium on Search-Based Software Engineering (SSBSE 2019)**. [S.l.: s.n.], 2019. Citado na página 167.
- VINCENZI, A. M. R.; MALDONADO, J. C.; WONG, W. E.; DELAMARO, M. E. Coverage testing of java programs and components. **Science of Computer Programming**, Elsevier, Amsterdam, Netherlands, 2005. ISSN 0167-6423. Citado na página 42.

- WALRAVEN, S.; LANDUYT, D. V.; TRUYEN, E.; HANDEKYN, K.; JOOSEN, W. Efficient customization of multi-tenant software-as-a-service applications with service lines. **JSS**, Elsevier, v. 91, p. 48–62, 2014. Citado nas páginas 85 e 86.
- WALRAVEN, S.; TRUYEN, E.; JOOSEN, W. A middleware layer for flexible and cost-efficient multi-tenant applications. In: **Int. Conf. on Middleware**. [S.l.]: Springer, 2011. p. 370–389. Citado na página 87.
- WEISSMAN, C. D.; BOBROWSKI, S. The design of the force. com multitenant internet application development platform. In: ACM. [S.l.], 2009. (Proceedings of the 2009 ACM SIGMOD International Conference on Management of data), p. 889–896. Citado na página 56.
- WOOD, K.; ANDERSON, M. Understanding the complexity surrounding multitenancy in cloud computing. In: **IEEE 8th Int. Conf. on e-Business Engineering**. [S.l.: s.n.], 2011. p. 119–124. Citado na página 88.
- WU, L. L.; KAISER, G. E. Constructing subtle concurrency bugs using synchronization-centric second-order mutation operators. In: . [S.l.: s.n.], 2011. (Proceedings of the 23rd International Conference on Software Engineering & Knowledge Engineering (SEKE)), p. 244–249. Citado na página 95.
- XUE, W.; QINGZHONG, L.; LANJU, K. Multiple sparse tables based on pivot table for multi-tenant data storage in saas. In: IEEE. **2011 IEEE International Conference on Information and Automation**. [S.l.], 2011. p. 634–637. Citado na página 145.
- YAISH, H.; GOYAL, M. Multi-tenant database access control. In: **16th Int. Conf. on Computational Science and Engineering**. [S.l.: s.n.], 2013. p. 870–877. Citado na página 89.
- YANG, C.-S.; POLLOCK, L. L. The Challenges in Automated Testing of Multithreaded Programs. In: **Proceedings of 14th International Conference on Testing Computer Software**. [S.l.: s.n.], 1997. p. 157–166. Citado na página 100.
- YANG, C.-S. D.; SOUTER, A. L.; POLLOCK, L. L. All-du-path coverage for parallel programs. **SIGSOFT Softw. Eng. Notes**, ACM, New York, NY, USA, v. 23, n. 2, p. 153–162, mar. 1998. ISSN 0163-5948. Disponível em: <<http://doi.acm.org/10.1145/271775.271804>>. Citado na página 132.
- ZHANG, K.; LI, Q.; SHI, Y. Data privacy preservation during schema evolution for multi-tenancy applications in cloud computing. In: **Web Information Systems and Mining**. [S.l.]: Springer, 2011. p. 376–383. Citado na página 88.

TRECHOS DOS CÓDIGOS-FONTE DO *MTSHOPFAILS*

Código-fonte 6 – Trecho de código da classe *MultiTenancyInterceptor.java*

```

1  public class MultiTenancyInterceptor extends HandlerInterceptorAdapter {

3      @Override
      /*1*/ public boolean preHandle(HttpServletRequest req, HttpServletResponse res,
          Object handler) throws Exception {
5      /*1*/ Map<String, Object> pathVars = (Map<String, Object>) req.getAttribute(
          HandlerMapping.URI_TEMPLATE_VARIABLES_ATTRIBUTE);

7      /*1*/ Authentication authentication = SecurityContextHolder.getContext().
          getAuthentication();

9      /*2*/ if (pathVars.containsKey("tenantid")) {
          /*3*/ if(!pathVars.get("tenantid").toString().equals(CurrentTenant.get()) &&
              authentication != null && authentication.getDetails() != null){
11     /*4*/ SecurityContextHolder.clearContext();
          /*4*/ HttpSession session = req.getSession(false);
13     /*5*/ if (session != null) {
          /*6*/ session.invalidate();
15     /*6*/ }
          /*4*/ }

17     /*7*/ req.setAttribute("CURRENT_TENANT_IDENTIFIER", pathVars.get("tenantid"));
          /*7*/ }

19     /*8*/ return true;
          /*8*/ }

21 }

```

Código-fonte 7 – Trecho de código da classe *AdminProduct.java*

```

1  @Controller
    @RequestMapping("/{tenantid}/admin")
3  public class AdminProduct {

5      @Autowired
          private ProductService productService;

```

```

7
    @Autowired
9     private ShopService shopService;

11
    @RequestMapping("/product/addProduct")
13     /*1*/ public String addProduct(@PathVariable String tenantid, Model model){
    /*1*/ Product product = new Product();
15     /*1*/ Shop shop = shopService.getAllData();

17     /*1*/ product.setProductCategory(shop.getCategory1());
    /*1*/ product.setProductCondition("new");
19     /*1*/ product.setProductStatus("active");
    /*1*/ product.setUrlPicture("sem");

21     /*1*/ model.addAttribute("shop", shop);
23     /*1*/ model.addAttribute("product", product);
    /*1*/ model.addAttribute("tenantid", tenantid);

25     /*1*/ return "addProduct";
27     /*1*/ }

29     @RequestMapping(value = "/product/addProduct", method = RequestMethod.POST
    )
    /*2*/ public String addProductPost(@PathVariable String tenantid, @Valid
        @ModelAttribute("product") Product product, BindingResult results,
        HttpServletRequest request, Model model){
31     /*2*/ model.addAttribute("tenantid", tenantid);
    /*3*/ if (String.valueOf(product.getProductPrice()).equals("0.0") || String.
        valueOf(product.getProductPrice()).equals("")){
33     /*4*/ model.addAttribute("errorPrice", "The product price must not be 0.");
    /*4*/ Shop shop = shopService.getAllData();
35     /*4*/ model.addAttribute("shop", shop);
    /*4*/ model.addAttribute("product", product);
37     /*4*/ return "addProduct";
    /*4*/ }

39     /*5*/ if (String.valueOf(product.getUnitInStock()).equals("")){
41     /*6*/ model.addAttribute("errorUnitInStock", "The product unit must not be null
        .");
    /*6*/ Shop shop = shopService.getAllData();
43     /*6*/ model.addAttribute("shop", shop);
    /*6*/ model.addAttribute("product", product);
45     /*6*/ return "addProduct";
    /*6*/ }

47     /*7*/ if(results.hasErrors()){
49     /*8*/ Shop shop = shopService.getAllData();
    /*8*/ model.addAttribute("shop", shop);
51     /*8*/ model.addAttribute("product", product);
    /*8*/ model.addAttribute("tenantid", tenantid);
53     /*8*/ return "addProduct";
    /*8*/ }

55     /*9*/ MultipartFile productImage = product.getProductImage();
    /*9*/ Map uploadResult = null;

57     /*9*/ if(productImage!=null && !productImage.isEmpty()){

```

```

59  /*10*/    try {
        /*10*/    uploadResult = SingletonCloudinary.getCloudinary().uploader().upload(
            productImage.getBytes(), ObjectUtils.asMap("png", "auto"));
61  /*10*/    product.setUrlPicture("v"+String.valueOf(uploadResult.get("version"))+"
        /"+String.valueOf(uploadResult.get("public_id")));
        /*11*/    } catch (Exception e) {
63  /*11*/        e.printStackTrace();
        /*11*/        throw new RuntimeException("Product image save failed",e);
65  /*11*/    }
        /*9*/    }
67  /*12*/    productService.addProduct(product);
        /*12*/    return "redirect:/{tenantid}/admin/productInventory";
69  /*12*/ }

71      @RequestMapping("/product/editProduct/{id}")
        /*13*/ public String editProduct(@PathVariable String tenantid, @PathVariable("
            id") int id, Model model){
73  /*13*/    Product product = productService.getProductById(id);
        /*13*/    Shop shop = shopService.getAllData();
75  /*13*/    model.addAttribute("shop", shop);
        /*13*/    model.addAttribute("product", product);
77  /*13*/    model.addAttribute("tenantid", tenantid);
        /*13*/    return "editProduct";
79  /*13*/ }

81      @RequestMapping(value = "/product/editProduct", method = RequestMethod.
        POST)
        /*14*/ public String editProductPost(@PathVariable String tenantid,
            @ModelAttribute("product") Product product, BindingResult results,
            HttpServletRequest request, Model model) throws IOException{
83  /*14*/    model.addAttribute("tenantid", tenantid);

85  /*15*/    if(results.hasErrors()){
        /*16*/    return "editProduct";
87  /*16*/ }

89  /*17*/    MultipartFile productImage = product.getProductImage();
        /*17*/    Map uploadResult = null;
91
        /*17*/    if(productImage!=null && !productImage.isEmpty()){
93  /*18*/    try {
        /*18*/    uploadResult = SingletonCloudinary.getCloudinary().uploader().upload(
            productImage.getBytes(), ObjectUtils.asMap("png", "auto"));
95  /*18*/    product.setUrlPicture("v"+String.valueOf(uploadResult.get("version"))+"
        /"+String.valueOf(uploadResult.get("public_id")));

97  /*19*/    } catch (Exception e) {
        /*19*/    e.printStackTrace();
99  /*19*/    throw new RuntimeException("Product image save failed",e); }
        /*20*/    } else {
101  /*20*/        Product prod = productService.getProductById(product.getProductId());
        /*21*/        if (prod.getUrlPicture()!=null){
103  /*22*/            product.setUrlPicture(prod.getUrlPicture());
        /*22*/        }
105  /*23*/    }

107  /*23*/    productService.editProduct(product);

```

```

109  /*23*/ return "redirect://{tenantid}/admin/productInventory";
    /*23*/ }

111
    @RequestMapping(value = "/product/deleteProduct/{id}")
113  /*24*/ public String deleteProduct(@PathVariable String tenantid, @PathVariable(
        "id") int id, Model model, HttpServletRequest request){
    /*24*/ model.addAttribute("tenantid", tenantid);
115  /*24*/ Product product = productService.getProductById(id);
    /*24*/ productService.deleteProduct(product);
117  /*25*/ return "redirect://{tenantid}/admin/productInventory";
    /*25*/ }
119 }

```

Código-fonte 8 – Trecho de código da classe *RegisterController.java*

```

1  @Controller
    @RequestMapping("/{tenantid}")
3  public class RegisterController {

5      @Autowired
        private CustomerService customerService;

7
        @RequestMapping("/register")
9  /*1*/ public String registerCustomer(@PathVariable String tenantid, Model model)
    {
    /*1*/ Customer customer = new Customer();
11  /*1*/ BillingAddress billingAddress = new BillingAddress();
    /*1*/ ShippingAddress shippingAddress = new ShippingAddress();
13  /*1*/ customer.setBillingAddress(billingAddress);
    /*1*/ customer.setShippingAddress(shippingAddress);
15  /*1*/ model.addAttribute("customer", customer);
    /*1*/ model.addAttribute("tenantid", tenantid);
17  /*1*/ return "registerCustomer";
    /*1*/ }

19
        @RequestMapping(value = "/register", method= RequestMethod.POST)
21  /*2*/ public String registerCustomerPost(@PathVariable String tenantid, @Valid
        @ModelAttribute("customer") Customer customer, BindingResult results, Model
        model){

23  /*2*/ model.addAttribute("tenantid", tenantid);

25  /*3*/ if(results.hasErrors()){
    /*4*/ return "registerCustomer";
27  /*4*/ }

29  /*5*/ if (customer.getCustomerPassword().length()>10 || customer.
        getCustomerPassword().length()<5){
    /*6*/ model.addAttribute("passwordMsgcharacter", " The password must be between 5
        and 10 characters");
31  /*6*/ model.addAttribute("customer", customer);
    /*6*/ return "registerCustomer";
33  /*6*/ }

35  /*7*/ List<Customer> customerList = customerService.getAllCustomers();

```

```

37  /*8*/ for(int i=0; i<customerList.size(); i++){
    /*9*/  if(customer.getEmail().equals(customerList.get(i).
        getEmail())){
39  /*10*/    model.addAttribute("emailMsg"," Email already exists!");
    /*10*/    return "registerCustomer";
41  /*10*/  }
    /*11*/ }
43
    /*12*/ for(int i=0; i<customerList.size(); i++){
45  /*13*/  if(customer.getUsername().equals(customerList.get(i).
        getUsername())){
    /*14*/    model.addAttribute("usernameMsg"," Username already exists!");
47  /*14*/    return "registerCustomer";
    /*14*/  }
49  /*15*/ }

51  /*16*/ customerService.addCustomer(customer);
    /*16*/ customer.setEnabled(true);
53  /*16*/ return "registerCustomerSucess";
    /*16*/ }
55 }

```

Código-fonte 9 – Trecho de código da classe *OrderController.java*

```

1  @Controller
    @RequestMapping("/{tenantid}")
3  public class OrderController {

5      @Autowired
        private CartService cartService;

7

        @Autowired
9        private CartItemService cartItemService;

11       @Autowired
            private CustomerOrderService customerOrderService;
13

            @RequestMapping("/order/{cartId}")
15  /*1*/ public String createOrder(@PathVariable String tenantid, @PathVariable("
            cartId") int cartId, Model model) {
        /*1*/ CustomerOrder customerOrder = new CustomerOrder();
17  /*1*/ Cart cart=cartService.getCartById(cartId);
        /*1*/ double grandTotalTemp = 0.0;
19

        /*2*/ for (int i=0; i<cart.getCartItems().size(); i++) {
21  /*3*/  grandTotalTemp = cart.getCartItems().get(i).getTotalPrice();
        /*3*/ }
23

        /*4*/ cart.setGrandTotal(grandTotalTemp);
25  /*4*/ customerOrder.setCart(cart);

27  /*4*/ Customer customer = cart.getCustomer();
        /*4*/ customerOrder.setCustomer(customer);
29  /*4*/ customerOrder.setBillingAddress(customer.getBillingAddress());
        /*4*/ customerOrder.setShippingAddress(customer.getShippingAddress());
31

        /*4*/ model.addAttribute("tenantid", tenantid);

```

```

33  /*4*/ model.addAttribute("order", customerOrder);
    /*4*/ return "orderConfirmation";
35  /*4*/ }

37      @RequestMapping(value = "/order/{cartId}", method = RequestMethod.POST)
    /*5*/ public String submitOrderPost(@PathVariable String tenantId, @Valid
        @ModelAttribute("order") CustomerOrder order, BindingResult results,
        HttpServletRequest request, Model model){
39  /*5*/ model.addAttribute("tenantId", tenantId);

41  /*6*/ if(results.hasErrors()){
    /*7*/ return "orderConfirmation";
43  /*7*/ }

45  /*8*/ Cart cart=cartService.getCartById(order.getCart().getCartId());
    /*8*/ cartItemService.removeAllCartItems(cart);
47  /*8*/ customerOrderService.addCustomerOrder(order);
    /*8*/ return "thankCustomer";
49  /*8*/ }
    }

```

Código-fonte 10 – Trecho de código da classe *AdminHome.java*

```

@Controller
2  @RequestMapping("/{tenantId}/admin")
    public class AdminHome {

4

        @Autowired
6        private ProductService productService;

8        @Autowired
        private CustomerService customerService;
10

        @RequestMapping
12  /*1*/ public String adminPage(@PathVariable String tenantId, Model model){
    /*1*/ model.addAttribute("tenantId", tenantId);
14  /*1*/ return "admin";
    /*1*/ }

16

        @RequestMapping("/productInventory")
18  /*2*/ public String productInventory(@PathVariable String tenantId, Model model)
        {
            /*2*/ List<Product> products = productService.getProductList();
20  /*2*/ model.addAttribute("products", products);
            /*2*/ model.addAttribute("tenantId", tenantId);
22  /*2*/ return "productInventory";
            /*2*/ }

24

        @RequestMapping("/customer")
26  /*3*/ public String customerManagement(@PathVariable String tenantId, Model
            model){
            /*3*/ List<Customer> customerList = customerService.getAllCustomers();
28  /*3*/ model.addAttribute("customerList", customerList);
            /*3*/ model.addAttribute("tenantId", tenantId);
30  /*3*/ return "customerManagement";
            /*3*/ }

32  }

```

Código-fonte 11 – Trecho de código da classe *ProductController.java*

```

@Controller
2 @RequestMapping("/{tenantid}/product")
  public class ProductController {
4
      @Autowired
6      ProductService productService;

8      @RequestMapping("/productList/all")
      /*1*/ public String getProducts(@PathVariable String tenantid, Model model){
10 /*1*/ List<Product> products = productService.getProductList();
      /*1*/ model.addAttribute("products", products);
12 /*1*/ model.addAttribute("tenantid", tenantid);
      /*1*/ return "productList";
14 /*1*/ }

16      @RequestMapping("/viewProduct/{id}")
      /*2*/ public String viewProduct(@PathVariable String tenantid, @PathVariable int
          id, Model model) throws IOException {
18 /*2*/ Product product = productService.getProductById(id);
      /*2*/ model.addAttribute("product", product);
20 /*2*/ model.addAttribute("tenantid", tenantid);
      /*2*/ return "viewProduct";
22 /*2*/ }

24      @RequestMapping("/productList")
      /*3*/ public String getProductByCategory(@PathVariable String tenantid,
          @RequestParam("searchCondition") String searchCondition, Model model){
26 /*3*/ List<Product> products = productService.getProductList();
      /*3*/ model.addAttribute("searchCondition", searchCondition);
28 /*3*/ model.addAttribute("products", products);
      /*3*/ model.addAttribute("tenantid", tenantid);
30 /*3*/ return "productList";
      /*3*/ }
32 }

```

Código-fonte 12 – Trecho de código da classe *HomeController.java*

```

@Controller
2 @RequestMapping("/{tenantid}")
  public class HomeController {
4
      @Autowired
6      private ShopService shopService;

8      @RequestMapping
      /*1*/ public String home(@PathVariable String tenantid, Model model){
10 /*1*/ Shop shop = shopService.getAllData();
      /*1*/ model.addAttribute("shop", shop);
12 /*1*/ model.addAttribute("tenantid", tenantid);
      /*1*/ CurrentTenant.set(tenantid);
14 /*1*/ return "home";
      /*1*/ }
16

```

```

    @RequestMapping("/login")
18  /*2*/ public String login(@PathVariable("tenantid") String tenantid,
    @RequestParam(value = "error", required = false) String error, @RequestParam
    (value = "logout", required = false) String logout, Model model,
    HttpServletRequest request){

20  /*3*/ if (error!=null){
    /*4*/ model.addAttribute("error", " Invalid username and password");
22  /*4*/ }
    /*5*/ if (logout!=null){
24  /*6*/ model.addAttribute("msg", " You have logout sucessfully");
    /*6*/ }

26
    /*7*/ model.addAttribute("tenantid", tenantid);
28  /*7*/ CurrentTenant.set(tenantid);
    /*7*/ return "login";
30  /*7*/ }
    (...)

32  @RequestMapping("/noaccess")
    /*8*/ public String accessDenied(@PathVariable String tenantid, Model model){
34  /*8*/ model.addAttribute("tenantid", tenantid);
    /*8*/ return "noaccess";
36  /*8*/ }

38  @RequestMapping("/about")
    /*9*/ public String about(@PathVariable String tenantid, Model model){
40  /*9*/ model.addAttribute("tenantid", tenantid);
    /*9*/ Shop shop = shopService.getAllData();
42  /*9*/ model.addAttribute("shop", shop);
    /*9*/ return "about";
44  /*9*/ }
}

```

Código-fonte 13 – Trecho de código da classe *CartController.java*

```

1  @Controller
    @RequestMapping("/{tenantid}/customer/cart")
3  public class CartController {

5      @Autowired
        private CustomerService customerService;

7

        @RequestMapping
9  /*1*/ public String getCart(@PathVariable String tenantid,
        @AuthenticationPrincipal CustomUserDetails activeUser, Model model){
    /*1*/ Customer customer = customerService.getCustomerByUsername(activeUser.
        getUsername());
11  /*1*/ int cartId = customer.getCart().getCartId();
    /*1*/ model.addAttribute("tenantid", tenantid);
13  /*2*/ return "redirect:{tenantid}/customer/cart/"+cartId;
    /*2*/ }

15

        @RequestMapping("/{cartId}")
17  /*3*/ public String getCartRedirect(@PathVariable String tenantid, @PathVariable
        (value ="cartId") int cartId, Model model){
    /*3*/ model.addAttribute("cartId", cartId);
19  /*3*/ model.addAttribute("tenantid", tenantid);

```

```

    /*3*/  return "cart";
21 /*3*/ }
    }

```

Código-fonte 14 – Trecho de código da página *cart.jsp*

```

(...)
2 /*1*/ <c:if test="${tenantid == 'tenant_1'}">
  /*2*/ <section class="container" ng-app="cartApp">
4 /*2*/ <div ng-controller = "cartCtrl" ng-init="initCartId('${cartId}')">
  /*2*/ </c:if>
6 /*3*/ <c:if test="${tenantid == 'tenant_2'}">
  /*4*/ <section class="container" ng-app="cartApp2">
8 /*4*/ <div ng-controller = "cartCtrl2" ng-init="initCartId('${cartId}')">
  /*4*/ </c:if>
10 /*5*/ <c:if test="${tenantid == 'tenant_3'}">
  /*6*/ <section class="container" ng-app="cartApp3">
12 /*6*/ <div ng-controller = "cartCtrl3" ng-init="initCartId('${cartId}')">
  /*6*/ </c:if>
14 /*7*/ (...)
  /*8*/ <c:if test="${tenantid == 'tenant_1'}">
16 /*9*/ <script src="<c:url value="/js/controller1.js" /> "></script>
  /*9*/ </c:if>
18 /*10*/ <c:if test="${tenantid == 'tenant_2'}">
  /*11*/ <script src="<c:url value="/js/controller2.js" /> "></script>
20 /*11*/ </c:if>
  /*12*/ <c:if test="${tenantid == 'tenant_3'}">
22 /*13*/ <script src="<c:url value="/js/controller3.js" /> "></script>
  /*13*/ </c:if>
24 /*14*/ (...)

```

Código-fonte 15 – Trecho de código da página *viewproduct.jsp*

```

(...)
2 /*1*/ <c:if test="${tenantid == 'tenant_1'}">
  /*2*/ <section class="container" ng-app="cartApp">
4 /*2*/ <div ng-controller = "cartCtrl" ng-init="initCartId('${cartId}')">
  /*2*/ </c:if>
6 /*3*/ <c:if test="${tenantid == 'tenant_2'}">
  /*4*/ <section class="container" ng-app="cartApp2">
8 /*4*/ <div ng-controller = "cartCtrl2" ng-init="initCartId('${cartId}')">
  /*4*/ </c:if>
10 /*5*/ <c:if test="${tenantid == 'tenant_3'}">
  /*6*/ <section class="container" ng-app="cartApp3">
12 /*6*/ <div ng-controller = "cartCtrl3" ng-init="initCartId('${cartId}')">
  /*6*/ </c:if>
14 /*7*/ (...)
  /*8*/ <c:if test="${tenantid == 'tenant_1'}">
16 /*9*/ <script src="<c:url value="/js/controller1.js" /> "></script>
  /*9*/ </c:if>
18 /*10*/ <c:if test="${tenantid == 'tenant_2'}">
  /*11*/ <script src="<c:url value="/js/controller2.js" /> "></script>
20 /*11*/ </c:if>
  /*12*/ <c:if test="${tenantid == 'tenant_3'}">
22 /*13*/ <script src="<c:url value="/js/controller3.js" /> "></script>
  /*13*/ </c:if>
24 /*14*/ (...)

```

Código-fonte 16 – Trecho de código da página *header.jsp*

```
(...)  
2  /*1*/ <c:if test="${tenantid == 'tenant_1'}">  
   /*2*/ <link href="<c:url value="/css/bootstrap.min.t1.css" />" rel="stylesheet"  
       >  
4  /*2*/ </c:if>  
   /*3*/ <c:if test="${tenantid == 'tenant_2'}">  
6  /*4*/ <link href="<c:url value="/css/bootstrap.min.t2.css" />" rel="stylesheet"  
       >  
   /*4*/ </c:if>  
8  /*5*/ <c:if test="${tenantid == 'tenant_3'}">  
   /*6*/ <link href="<c:url value="/css/bootstrap.min.t3.css" />" rel="stylesheet"  
       >  
10 /*6*/ </c:if>  
   /*7*/ (...)
```

TRECHOS DOS CÓDIGOS-FONTE DO *MTSHOPSAFE*

Código-fonte 17 – Trecho de código da classe *AdminHome.java*

```
1  @Controller
   @RequestMapping("/{tenantid}/admin")
3  public class AdminHome {

5      @Autowired
       private ProductService productService;

7

       @Autowired
9       private CustomerService customerService;

11     @RequestMapping
       /*1*/ public String adminPage(@PathVariable String tenantid, Model model){
13     /*1*/ model.addAttribute("tenantid", tenantid);
       /*1*/ return "admin";
15     /*1*/ }

17     @RequestMapping("/{productInventory}")
       /*2*/ public String productInventory(@PathVariable String tenantid, Model model)
       {
19     /*2*/ List<Product> products = productService.getProductList();
       /*2*/ model.addAttribute("products", products);
21     /*2*/ model.addAttribute("tenantid", tenantid);
       /*2*/ return "productInventory";
23     /*2*/ }

25     @RequestMapping("/{customer}")
       /*3*/ public String customerManagement(@PathVariable String tenantid, Model
       model){
27     /*3*/ List<Customer> customerList = customerService.getAllCustomers();
       /*3*/ model.addAttribute("customerList", customerList);
29     /*3*/ model.addAttribute("tenantid", tenantid);
       /*3*/ return "customerManagement";
31     /*3*/ }
}
```

Código-fonte 18 – Trecho de código da classe *AdminProduct.java*

```

@Controller
2  @RequestMapping("/{tenantid}/admin")
   public class AdminProduct {
4
       @Autowired
6       private ProductService productService;

       @Autowired
8       private ShopService shopService;
10

12       @RequestMapping("/product/addProduct")
       /*1*/ public String addProduct(@PathVariable String tenantid, Model model){
14       /*1*/ Product product = new Product();
       /*1*/ Shop shop = shopService.getAllData();
16
       /*1*/ product.setProductCategory(shop.getCategory1());
18       /*1*/ product.setProductCondition("new");
       /*1*/ product.setProductStatus("active");
20       /*1*/ product.setUrlPicture("sem");

22       /*1*/ model.addAttribute("shop", shop);
       /*1*/ model.addAttribute("product", product);
24       /*1*/ model.addAttribute("tenantid", tenantid);

26       /*1*/ return "addProduct";
       /*1*/ }
28

       @RequestMapping(value = "/product/addProduct", method = RequestMethod.POST
           )
30       /*2*/ public String addProductPost(@PathVariable String tenantid, @Valid
           @ModelAttribute("product") Product product, BindingResult results,
           HttpServletRequest request, Model model){
       /*2*/ model.addAttribute("tenantid", tenantid);
32       /*3*/ if (String.valueOf(product.getProductPrice()).equals("0.0") || String.
           valueOf(product.getProductPrice()).substring(0,1).equals("-") || String.
           valueOf(product.getProductPrice()).equals("")){
       /*4*/ model.addAttribute("errorPrice","The product price must not be 0 or
           less than 0.");
34       /*4*/ Shop shop = shopService.getAllData();
       /*4*/ model.addAttribute("shop", shop);
36       /*4*/ model.addAttribute("product", product);
       /*4*/ return "addProduct";
38       /*4*/ }

40       /*5*/ if (String.valueOf(product.getUnitInStock()).equals("") ||
       /*6*/ String.valueOf(product.getUnitInStock()).substring(0,1).equals("-")) {
42       /*6*/ model.addAttribute("errorUnitInStock","The product unit must not be
           null or less than 0.");
       /*6*/ Shop shop = shopService.getAllData();
44       /*6*/ model.addAttribute("shop", shop);
       /*6*/ model.addAttribute("product", product);
46       /*6*/ return "addProduct";

```

```

    /**6*/    }
48
    /**7*/    if(results.hasErrors()){
50 /**8*/        Shop shop = shopService.getAllData();
    /**8*/        model.addAttribute("shop", shop);
52 /**8*/        model.addAttribute("product", product);
    /**8*/        model.addAttribute("tenantid", tenantid);
54 /**8*/        return "addProduct";
    /**8*/    }
56 /**9*/    MultipartFile productImage = product.getProductImage();
    /**9*/    Map uploadResult = null;
58
    /**9*/    if(productImage!=null && !productImage.isEmpty()){
60 /**10*/        try {
    /**10*/        uploadResult = SingletonCloudinary.getCloudinary().uploader().upload(
            productImage.getBytes(),ObjectUtils.asMap("png", "auto"));
62 /**10*/        product.setUrlPicture("v"+String.valueOf(uploadResult.get("version"))+"/
            "+String.valueOf(uploadResult.get("public_id")));
    /**11*/        } catch (Exception e) {
64 /**11*/            e.printStackTrace();
    /**11*/            throw new RuntimeException("Product image save failed",e);
66 /**11*/        }
    /**9*/    }
68 /**12*/    productService.addProduct(product);
    /**12*/    return "redirect:/{tenantid}/admin/productInventory";
70 /**12*/ }

72    @RequestMapping("/product/editProduct/{id}")
    /**13*/    public String editProduct(@PathVariable String tenantid, @PathVariable("
        id") int id, Model model){
74 /**13*/    Product product = productService.getProductById(id);
    /**13*/    Shop shop = shopService.getAllData();
76 /**13*/    model.addAttribute("shop", shop);
    /**13*/    model.addAttribute("product", product);
78 /**13*/    model.addAttribute("tenantid", tenantid);
    /**13*/    return "editProduct";
80 /**13*/ }

82    @RequestMapping(value = "/product/editProduct", method = RequestMethod.
        POST)
    /**14*/    public String editProductPost(@PathVariable String tenantid,
        @ModelAttribute("product") Product product, BindingResult results,
        HttpServletRequest request, Model model) throws IOException{
84 /**14*/    model.addAttribute("tenantid", tenantid);

86 /**15*/    if (String.valueOf(product.getProductPrice()).equals("0.0") || String.
        valueOf(product.getProductPrice()).substring(0,1).equals("-") || String.
        valueOf(product.getProductPrice()).equals("")){
    /**16*/        model.addAttribute("errorPrice","The product price must not be 0 or less
            than 0.");
88 /**16*/    Shop shop = shopService.getAllData();
    /**16*/    model.addAttribute("shop", shop);
90 /**16*/    model.addAttribute("product", product);
    /**16*/    return "editProduct";
92 /**16*/ }

94 /**17*/    if (String.valueOf(product.getUnitInStock()).equals("")) || String.valueOf

```

```

        (product.getUnitInStock()).substring(0,1).equals("-")) {
/*18*/  model.addAttribute("errorUnitInStock","The product unit must not be null
        or less than 0.");
96 /*18*/  Shop shop = shopService.getAllData();
/*18*/  model.addAttribute("shop", shop);
98 /*18*/  model.addAttribute("product", product);
/*18*/  return "editProduct";
100 /*18*/ }

102 /*19*/ if(results.hasErrors()){
/*20*/  Shop shop = shopService.getAllData();
104 /*20*/  model.addAttribute("shop", shop);
/*20*/  model.addAttribute("product", product);
106 /*20*/  model.addAttribute("tenantid", tenantid);
/*20*/  return "editProduct";
108 /*20*/ }

110 /*21*/ MultipartFile productImage = product.getProductImage();
/*21*/ Map uploadResult = null;
112
/*21*/ if(productImage!=null && !productImage.isEmpty()){
114 /*22*/ try {
/*22*/  uploadResult = SingletonCloudinary.getCloudinary().uploader().upload(
        productImage.getBytes(), ObjectUtils.asMap("png", "auto"));
116 /*22*/  product.setUrlPicture("v"+String.valueOf(uploadResult.get("version"))+"/
        "+String.valueOf(uploadResult.get("public_id")));
/*23*/  } catch (Exception e) {
118 /*23*/    e.printStackTrace();
/*23*/    throw new RuntimeException("Product image save failed",e); }
120 /*24*/ } else {
/*24*/    Product prod = productService.getProductById(product.getProductId());
122 /*25*/    if (prod.getUrlPicture()!=null){
/*26*/        product.setUrlPicture(prod.getUrlPicture());
124 /*26*/    }
/*27*/ }

126
/*27*/ productService.editProduct(product);
128
/*27*/ return "redirect:/{tenantid}/admin/productInventory";
130 /*27*/ }

132 @RequestMapping(value = "/product/deleteProduct/{id}")
/*28*/ public String deleteProduct(@PathVariable String tenantid, @PathVariable(
        "id") int id, Model model, HttpServletRequest request){
134 /*28*/ model.addAttribute("tenantid", tenantid);
/*28*/ Product product = productService.getProductById(id);
136 /*28*/ productService.deleteProduct(product);
/*29*/ return "redirect:/{tenantid}/admin/productInventory";
138 /*29*/ }
}

```

Código-fonte 19 – Trecho de código da classe *CartController.java*

```

1 @Controller
  @RequestMapping("/{tenantid}/customer/cart")
3 public class CartController {

```

```

5      @Autowired
      private CustomerService customerService;

7

      @RequestMapping
9  /***/ public String getCart(@PathVariable String tenantId,
      @AuthenticationPrincipal CustomUserDetails activeUser, Model model){
      /***/ Customer customer = customerService.getCustomerByUsername(activeUser.
      getUsername());
11 /***/ int cartId = customer.getCart().getCartId();
      /***/ model.addAttribute("tenantId", tenantId);
13 /***/ return "redirect:/{tenantId}/customer/cart/"+cartId;
      /***/ }

15

      @RequestMapping("/{cartId}")
17 /***/ public String getCartRedirect(@PathVariable String tenantId, @PathVariable
      (value = "cartId") int cartId, Model model){
      /***/ model.addAttribute("cartId", cartId);
19 /***/ model.addAttribute("tenantId", tenantId);
      /***/ return "cart";
21 /***/ }
      }

```

Código-fonte 20 – Trecho de código da classe *CustomAuthenticationFilter.java*

```

public class CustomAuthenticationFilter extends
      UsernamePasswordAuthenticationFilter {

2

      public static final String SPRING_SECURITY_FORM_TENANT_NAME_KEY = "tenant";

4

      @Override
6  public Authentication attemptAuthentication(HttpServletRequest request,
      HttpServletResponse response) throws AuthenticationException {
      if (!request.getMethod().equals("POST")) {
8          throw new AuthenticationServiceException("Authentication method not
      supported: " + request.getMethod());
      }

10

      CustomAuthenticationToken authRequest = getAuthRequest(request);
12  String tenant = authRequest.getTenant();
      TenantContextHolder.setTenantId(tenant);
14  setDetails(request, authRequest);
      return this.getAuthenticationManager().authenticate(authRequest); //lento
16  }

18 /***/ private CustomAuthenticationToken getAuthRequest(HttpServletRequest
      request) {
      /***/ String username = obtainUsername(request);
20 /***/ String password = obtainPassword(request);
      /***/ String tenant = obtainTenant(request);

22

      /***/ if (username == null) {
24 /***/ username = "";
      /***/ }

26 /***/ if (password == null) {
      /***/ password = "";
28 /***/ }
      /***/ if (tenant == null) {

```

```

30  /*7*/  tenant = "";
    /*7*/  }
32
    /*8*/  return new CustomAuthenticationToken(username, password, tenant);
34  /*8*/  }

36  private String obtainTenant(HttpServletRequest request) {
    return request.getParameter(
        SPRING_SECURITY_FORM_TENANT_NAME_KEY);
38  }
}

```

Código-fonte 21 – Trecho de código da classe *CustomAuthenticationToken.java*

```

1  public class CustomAuthenticationToken extends
    UsernamePasswordAuthenticationToken {
    private static final long serialVersionUID = 1L;
3
    private String tenant;
5
    /*1*/ public CustomAuthenticationToken(Object principal, Object credentials,
        String tenant) {
7  /*1*/ super(principal, credentials);
    /*1*/ this.tenant = tenant;
9  /*1*/ super.setAuthenticated(false);
    /*1*/ }
11
    /*2*/ public CustomAuthenticationToken(Object principal, Object credentials,
        String tenant, Collection<? extends GrantedAuthority> authorities) {
13 /*2*/ super(principal, credentials, authorities);
    /*2*/ this.tenant = tenant;
15 /*2*/ super.setAuthenticated(true);
    /*2*/ }
17
    public String getTenant() {
19  return this.tenant;
    }
21 }

```

Código-fonte 22 – Trecho de código da classe *CustomUserDetails.java*

```

1  public class CustomUserDetails extends org.springframework.security.core.
   .userdetails.User {
    private static final long serialVersionUID = 1L;
3
    private String tenant;
5
    /*1*/ public CustomUserDetails(String username, String password, Collection<?
        extends GrantedAuthority> authorities, String tenant) {
7  /*1*/ super(username, password, authorities);
    /*1*/ this.tenant = tenant;
9  /*1*/ }

11  public String getTenant() {
    return tenant;
13  }

15  /*2*/ public void setTenant(String tenant) {

```

```

    /***/ this.tenant = tenant;
17 /***/ }
    }

```

Código-fonte 23 – Trecho de código da classe *HomeController.java*

```

@Controller
2 @RequestMapping("/{tenantid}")
    public class HomeController {
4
        @Autowired
6        private ShopService shopService;

8        @RequestMapping
        /***/ public String home(@PathVariable String tenantid, Model model){
10 /***/ Shop shop = shopService.getAllData();
        /***/ model.addAttribute("shop", shop);
12 /***/ model.addAttribute("tenantid", tenantid);
        /***/ return "home";
14 /***/ }

16        @RequestMapping("/login")
        /***/ public String login(@PathVariable("tenantid") String tenantid,
            @RequestParam(value = "error", required = false) String error, @RequestParam
            (value = "logout", required = false) String logout, Model model,
            HttpServletRequest request){
18
        /***/ if (error!=null){
20 /***/ model.addAttribute("error", " Invalid username and password");
        /***/ }
22 /***/ if (logout!=null){
        /***/ model.addAttribute("msg", " You have logout sucessfully");
24 /***/ }

26 /***/ model.addAttribute("tenantid", tenantid);
        /***/ return "login";
28 /***/ }
        (...)

30        @RequestMapping("/about")
        /***/ public String about(@PathVariable String tenantid, Model model){
32 /***/ model.addAttribute("tenantid", tenantid);
        /***/ Shop shop = shopService.getAllData();
34 /***/ model.addAttribute("shop", shop);
        /***/ return "about";
36 /***/ }
    }

```

Código-fonte 24 – Trecho de código da classe *MultiTenancyInterceptor.java*

```

1 public class MultiTenancyInterceptor extends HandlerInterceptorAdapter {

3        @Override
        /***/ public boolean preHandle(HttpServletRequest req, HttpServletResponse res,
            Object handler) throws Exception {
5 /***/ Map<String, Object> pathVars = (Map<String, Object>) req.getAttribute
            HandlerMapping.URI_TEMPLATE_VARIABLES_ATTRIBUTE);

```

```

7  /*1*/ Authentication auth = SecurityContextHolder.getContext().getAuthentication
    ();

9  /*2*/ if (auth != null && !auth.getClass().equals(AnonymousAuthenticationToken.
    class)) {
    /*3*/ CustomUserDetails userDetails = (CustomUserDetails) auth.getPrincipal();
11 /*4*/ if (!userDetails.getTenant().equals(String.valueOf(pathVars.get("tenantid"
    )))) {
    /*5*/ res.sendError(HttpServletResponse.SC_UNAUTHORIZED, "Authentication
        Failed");
13 /*5*/ SecurityContextHolder.clearContext();
    /*5*/ HttpSession session = req.getSession(false);
15 /*6*/ if (session != null) {
    /*7*/ session.invalidate();
17 /*7*/ }
    /*8*/ TenantContextHolder.setTenantId(userDetails.getTenant());
19 /*8*/ }
    /*9*/ if (pathVars.containsKey("tenantid")) {
21 /*10*/ TenantContextHolder.setTenantId(String.valueOf(pathVars.get("tenantid")
    ));
    /*10*/ }
23 /*11*/ return true;
    /*11*/ }
25 /*12*/ }
    }

```

Código-fonte 25 – Trecho de código da classe *OrderController.java*

```

@Controller
2 @RequestMapping("/{tenantid}")
public class OrderController {
4
    @Autowired
6 private CartService cartService;

8 @Autowired
private CartItemService cartItemService;

10 @Autowired
12 private CustomerOrderService customerOrderService;

14 @RequestMapping("/order/{cartId}")
/*1*/ public String createOrder(@PathVariable String tenantid, @PathVariable("
    cartId") int cartId, Model model) {
16 /*1*/ CustomerOrder customerOrder = new CustomerOrder();
    /*1*/ Cart cart=cartService.getCartById(cartId);
18 /*1*/ double grandTotalTemp = 0.0;

20 /*2*/ for (int i=0; i<cart.getCartItems().size(); i++) {
    /*3*/ grandTotalTemp = cart.getCartItems().get(i).getTotalPrice();
22 /*3*/ }

24 /*4*/ cart.setGrandTotal(grandTotalTemp);
    /*4*/ customerOrder.setCart(cart);
26
    /*4*/ Customer customer = cart.getCustomer();
28 /*4*/ customerOrder.setCustomer(customer);

```

```

    /**4*/ customerOrder.setBillingAddress(customer.getBillingAddress());
30 /**4*/ customerOrder.setShippingAddress(customer.getShippingAddress());

32 /**4*/ model.addAttribute("tenantid", tenantid);
    /**4*/ model.addAttribute("order", customerOrder);
34 /**4*/ return "orderConfirmation";
    /**4*/ }

36
    @RequestMapping(value = "/order/{cartId}", method = RequestMethod.POST)
38 /**5*/ public String submitOrderPost(@PathVariable String tenantid, @Valid
    @ModelAttribute("order") CustomerOrder order, BindingResult results,
    HttpServletRequest request, Model model){
    /**5*/ model.addAttribute("tenantid", tenantid);

40
    /**6*/ if(results.hasErrors()){
42 /**7*/ return "orderConfirmation";
    /**7*/ }

44
    /**8*/ Cart cart=cartService.getCartById(order.getCart().getCartId());
46 /**8*/ cartItemService.removeAllCartItems(cart);
    /**8*/ customerOrderService.addCustomerOrder(order);
48 /**8*/ return "thankCustomer";
    /**8*/ }

50 }

```

Código-fonte 26 – Trecho de código da classe *ProductController.java*

```

@Controller
2 @RequestMapping("/{tenantid}/product")
    public class ProductController {

4
        @Autowired
6        ProductService productService;

8
        @RequestMapping("/productList/all")
    /**1*/ public String getProducts(@PathVariable String tenantid, Model model){
10 /**1*/ List<Product> products = productService.getProductList();
    /**1*/ model.addAttribute("products", products);
12 /**1*/ model.addAttribute("tenantid", tenantid);
    /**1*/ return "productList";
14 /**1*/ }

16
        @RequestMapping("/viewProduct/{id}")
    /**2*/ public String viewProduct(@PathVariable String tenantid, @PathVariable int
        id, Model model) throws IOException {
18 /**2*/ Product product = productService.getProductById(id);
    /**2*/ model.addAttribute("product", product);
20 /**2*/ model.addAttribute("tenantid", tenantid);
    /**2*/ return "viewProduct";
22 /**2*/ }

24
        @RequestMapping("/productList")
    /**3*/ public String getProductByCategory(@PathVariable String tenantid,
        @RequestParam("searchCondition") String searchCondition, Model model){
26 /**3*/ List<Product> products = productService.getProductList();
    /**3*/ model.addAttribute("searchCondition", searchCondition);
28 /**3*/ model.addAttribute("products", products);

```

```

/*3*/  model.addAttribute("tenantid", tenantid);
30  /*3*/  return "productList";
/*3*/  }
32  }

```

Código-fonte 27 – Trecho de código da classe *RegisterController.java*

```

@Controller
2  @RequestMapping("/{tenantid}")
  public class RegisterController {
4
      @Autowired
6      private CustomerService customerService;

8      @RequestMapping("/register")
/*1*/  public String registerCustomer(@PathVariable String tenantid, Model model)
      {
10  /*1*/  Customer customer = new Customer();
/*1*/  BillingAddress billingAddress = new BillingAddress();
12  /*1*/  ShippingAddress shippingAddress = new ShippingAddress();
/*1*/  customer.setBillingAddress(billingAddress);
14  /*1*/  customer.setShippingAddress(shippingAddress);
/*1*/  model.addAttribute("customer", customer);
16  /*1*/  model.addAttribute("tenantid", tenantid);
/*1*/  return "registerCustomer";
18  /*1*/  }

20      @RequestMapping(value = "/register", method= RequestMethod.POST)
/*2*/  public String registerCustomerPost(@PathVariable String tenantid, @Valid
      @ModelAttribute("customer") Customer customer, BindingResult results, Model
      model){
22
/*2*/  model.addAttribute("tenantid", tenantid);
24
/*3*/  if (!EmailValidator.getInstance().isValid(customer.getCustomerEmail())){
26  /*4*/  model.addAttribute("emailMsg", " Invalid email");
/*4*/  model.addAttribute("customer", customer);
28  /*4*/  return "registerCustomer";
/*4*/  }
30  /*5*/  if (customer.getCustomerUsername().length()>15) {
/*6*/  model.addAttribute("usernameMsgcharacter", " The username must be at most
      15 characters!");
32  /*6*/  model.addAttribute("customer", customer);
/*6*/  return "registerCustomer";
34  /*6*/  }
/*7*/  if (customer.getCustomerPassword().length()>10 || customer.
      getCustomerPassword().length()<5){
36  /*8*/  model.addAttribute("passwordMsgcharacter", " The password must be between 5
      and 10 characters");
/*8*/  model.addAttribute("customer", customer);
38  /*8*/  return "registerCustomer";
/*8*/  }
40  /*9*/  if (EmailValidator.getInstance().isValid(customer.getCustomerEmail()) &&
/*10*/  customerService.emailAlreadyExists(customer.getCustomerEmail())) {
42  /*10*/  model.addAttribute("emailMsg", " Email already exists!");
/*10*/  model.addAttribute("customer", customer);
44  /*10*/  return "registerCustomer";

```

```
    /*10*/ }
46  /*11*/ if (customerService.usernameAlreadyExists(customer.getCustomerUsername())
    ) {
    /*12*/  model.addAttribute("usernameMsg"," Username already exists!");
48  /*12*/  model.addAttribute("customer", customer);
    /*12*/  return "registerCustomer";
50  /*12*/  }
    /*13*/  if(results.hasErrors()){
52  /*14*/  model.addAttribute("customer", customer);
    /*14*/  return "registerCustomer";
54  /*14*/  }
    /*15*/  customerService.addCustomer(customer);
56  /*15*/  return "registerCustomerSucess";
    /*15*/  }
58  }
```

TRECHOS DOS CÓDIGOS-FONTE DO *ICARDAPIO*

Código-fonte 28 – Trecho de código da classe *ICardapioTenantResolver.java*

```
public class ICardapioTenantResolver implements CurrentTenantResolver<Long> {
2
    private static String MASTER_TENANT_SUBDOMAIN = "default";
4
    @Autowired
6    private RestaurantsRepository restaurantsRepository;

8    /**/ protected Long getCurrentTenantFromSubdomain() {
    /**/ ServletRequestAttributes attr = (ServletRequestAttributes)
        RequestContextHolder.currentRequestAttributes();
10    /**/ String[] url = attr.getRequest().getRequestURL().toString().split("/");
    /**/ String subdomain = url[url.length-1];
12
    /**2*/ if (subdomain.equals("addProduct")){
14    /**3*/ subdomain = url[url.length-2];
    /**3*/ }
16

    /**4*/ Restaurant restaurant = restaurantsRepository.getBySubdomain(subdomain);
18    /**4*/ return restaurant != null ? restaurant.getTenantId() : 0;
    /**4*/ }
20

    @Override
22    public Long getMasterTenantId() {
        return restaurantsRepository.getBySubdomain(MASTER_TENANT_SUBDOMAIN).
            getTenantId();
24    }

26    @Override
    public boolean isMasterTenant() {
28        Long masterTenant = getMasterTenantId();
        Long tenantFromSubdomain = getCurrentTenantFromSubdomain();
30        return masterTenant.equals(tenantFromSubdomain);
    }
32
```

```

@Override
34 public Long getCurrentTenantId() {
    return getCurrentTenantFromSubdomain();
36 }
}

```

Código-fonte 29 – Trecho de código da classe *RestaurantController.java*

```

1 @Controller
  @Transactional(propagation=Propagation.REQUIRED)
3 public class RestaurantController {

5     @Autowired
    private RestaurantFacade facade;

7

    @Autowired
9     private CurrentTenantResolver<Long> tenantResolver;

11    @RequestMapping("favicon.ico")
    @ResponseBody
13    void favicon() {}

15    @RequestMapping("/{tenant}")
    /*1*/ public String home(@PathVariable String tenant, Model model) {
17    /*1*/ Restaurant restaurant = facade.getRestaurant();
    /*1*/ model.addAttribute("restaurant", restaurant);
19    /*1*/ List<Category> categories = facade.getAllCategories();
    /*1*/ model.addAttribute("categories", categories);
21    /*1*/ model.addAttribute("tenant", tenant);
    /*1*/ return "home";
23    /*1*/ }

25    @RequestMapping("/")
    /*2*/ public String landing(@ModelAttribute("tenant") Restaurant tenant, Model
        model) {
27    /*2*/ Restaurant master = facade.getRestaurantById(tenantResolver.
        getMasterTenantId());
    /*2*/ model.addAttribute("restaurant", master);
29    /*2*/ model.addAttribute("tenant", tenant);
    /*2*/ return "landing";
31    /*2*/ }

33    @RequestMapping(value="/{tenant}/login", method = RequestMethod.GET)
    /*3*/ public String login_map(@PathVariable String tenant, Model model) {
35    /*3*/ model.addAttribute("subdomain", tenant);
    /*3*/ model.addAttribute("error", "false");
37    /*3*/ model.addAttribute("tenant", tenant);
    /*3*/ return "login";
39    /*3*/ }

41    @RequestMapping(value="/{tenant}/loginError", method = RequestMethod.GET)
    /*4*/ public String loginError(@PathVariable String tenant, Model model) {
43    /*4*/ model.addAttribute("error", "true");
    /*4*/ model.addAttribute("tenant", tenant);
45    /*4*/ return "login";
    /*4*/ }
47

```

```

        @RequestMapping(value="/{tenant}/logout", method = RequestMethod.GET)
49  /*5*/ public void logout(@PathVariable String tenant, HttpServletRequest request
    , HttpServletResponse response) {
    /*5*/ model.addAttribute("tenant", tenant);
51  /*5*/ LogoutSuccessHandlerVh efetuaLogout = new LogoutSuccessHandlerVh();
    /*5*/ try {
53  /*6*/     efetuaLogout.onLogoutSuccess(request, response, SecurityContextHolder.
        getContext().getAuthentication());
    /*6*/ }
55  /*7*/ catch (IOException e) {
        // TODO Auto-generated catch block
57  /*8*/     e.printStackTrace();
    /*8*/ }
59  /*9*/ catch (ServletException e) {
        // TODO Auto-generated catch block
61  /*10*/     e.printStackTrace();
    /*10*/ }
63  /*11*/ }

        @PreAuthorize("hasRole('admin')")
        @RequestMapping(value = "{tenant}/addProduct", method = RequestMethod.
            POST)
67  /*12*/ public String addProduct(@PathVariable String tenant, @ModelAttribute("
    product") Product product, Model model, final RedirectAttributes
    redirectAttributes) {
    /*12*/ product = facade.addProduct(product);
69  /*12*/ model.addAttribute("product", product);
    /*12*/ redirectAttributes.addFlashAttribute("message", "Produto adicionado com
        sucesso");
71  /*12*/ return "redirect:/" + tenant;
    /*12*/ }

73
        @PreAuthorize("hasRole('admin')")
75  @RequestMapping(value = "{tenant}/removeProduct", method = RequestMethod.
            GET)
    /*13*/ public String removeProduct(@PathVariable String tenant, @ModelAttribute(
        "categoryId") Long categoryId, @ModelAttribute("productId") Long productId,
        final RedirectAttributes redirectAttributes) {
77  /*13*/ facade.removeProduct(categoryId, productId);
    /*13*/ model.addAttribute("tenant", tenant);
79  /*13*/ redirectAttributes.addFlashAttribute("message", "Produto removido com
        sucesso");
    /*14*/ return "redirect:/" + tenant;
81  /*14*/ }

83  (...)

85  }

```

Código-fonte 30 – Trecho de código da interface *CurrentTenantResolver.java*

```

1  public interface CurrentTenantResolver<T extends Serializable> {

3      T getCurrentTenantId();

5      T getMasterTenantId();

```

```
7  boolean isMasterTenant();
9  }
```

Código-fonte 31 – Trecho de código da classe *Product.java*

```
1  @Entity
   @Multitenant
3  @TenantDiscriminatorColumn(name="TENANT_ID", discriminatorType=
    DiscriminatorType.INTEGER, contextProperty=PersistenceUnitProperties.
    MULTITENANT_PROPERTY_DEFAULT)
   public class Product implements Serializable {
5
   private static final long serialVersionUID = 1L;
7
   @Id
   @GeneratedValue(strategy=GenerationType.IDENTITY)
   private Long id;
11
   @Column(name="TENANT_ID", insertable=false, updatable=false)
13  private Long tenantId;

15  @ManyToOne
   private Category category;
17
   private String name;
19
   private String description;
21
   private BigDecimal price;
23
   public Product() {
25     super();
   }
27
   public Long getId() {
29     return this.id;
   }
31
   public void setId(Long id) {
33     this.id = id;
   }
35
   public Long getTenantId() {
37     return tenantId;
   }
39
   public Category getCategory() {
41     return this.category;
   }
43
   public void setCategory(Category category) {
45     this.category = category;
   }
47
   public String getName() {
49     return this.name;
```

```

    }
51
    public void setName(String name) {
53        this.name = name;
    }
55
    public String getDescription() {
57        return this.description;
    }
59
    public void setDescription(String description) {
61        this.description = description;
    }
63
    public BigDecimal getPrice() {
65        return this.price;
    }
67
    public void setPrice(BigDecimal price) {
69        this.price = price;
    }
71
    @Override
73    public int hashCode() {
        return new HashCodeBuilder(5, 7)
75        .append(tenantId)
        .append(category)
77        .append(name)
        .toHashCode();
79    }

81    @Override
    public boolean equals(Object obj) {
83        if (obj == null) {
            return false;
85        }
        if (obj == this) {
87            return true;
        }
89        if (obj.getClass() != getClass()) {
            return false;
91        }

93        Product rhs = (Product) obj;
        return new EqualsBuilder()
95        .appendSuper(super.equals(obj))
        .append(tenantId, rhs.getTenantId())
97        .append(category, rhs.getCategory())
        .append(name, rhs.getName())
99        .isEquals();
    }
101 }

```

Código-fonte 32 – Trecho de código da classe *Restaurant.java*

```

1    @Entity
    public class Restaurant implements Serializable {

```

```
3
    private static final long serialVersionUID = 1L;
5
    @Id
7    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Long id;
9
    private String name;
11
    private String subdomain;
13
    private String slogan;
15
    private String phone;
17
    private String address;
19
    private String city;
21
    public Restaurant() {
23        super();
    }
25
    public Long getId() {
27        return id;
    }
29
    public void setId(Long id) {
31        this.id = id;
    }
33
    @Transient
35    public Long getTenantId() {
        return this.id;
37    }
39
    public String getName() {
        return this.name;
41    }
43
    public void setName(String name) {
        this.name = name;
45    }
47
    public String getSubdomain() {
        return subdomain;
49    }
51
    /*1*/ public void setSubdomain(String subdomain) {
    /*1*/     this.subdomain = subdomain;
53 /*1*/ }
55
    public String getSlogan() {
        return this.slogan;
57    }
59
    public void setSlogan(String slogan) {
```

```
        this.slogan = slogan;
61    }

63    public String getPhone() {
        return this.phone;
65    }

67    public void setPhone(String phone) {
        this.phone = phone;
69    }

71    public String getAddress() {
        return this.address;
73    }

75    public void setAddress(String address) {
        this.address = address;
77    }

79    public String getCity() {
        return city;
81    }

83    public void setCity(String city) {
        this.city = city;
85    }

87    @Override
    public int hashCode() {
89        return new HashCodeBuilder(9, 11)
            .append(name)
91        .toHashCode();
    }

93    @Override
95    public boolean equals(Object obj) {
        if (obj == null) {
97            return false;
        }
99        if (obj == this) {
            return true;
101        }
        if (obj.getClass() != getClass()) {
103            return false;
        }

105        Restaurant rhs = (Restaurant) obj;
107        return new EqualsBuilder()
            .appendSuper(super.equals(obj))
109            .append(name, rhs.getName())
            .isEquals();
111    }
}
```

Código-fonte 33 – Trecho de código da classe *RestaurantFacade.java*

@Service

```
2  @Transactional(propagation=Propagation.MANDATORY)
   public class RestaurantFacade {
4
   @Autowired
6   private CurrentTenantResolver<Long> tenantResolver;

8   @Autowired
   private RestaurantsRepository restaurantsRepository;
10
   @Autowired
12  private CategoriesRepository categoriesRepository;

14  @Autowired
   private ProductsRepository productsRepository;
16

   public Restaurant getRestaurant() {
18     return restaurantsRepository.findOne(tenantResolver.getCurrentTenantId());
   }
20

   public Restaurant getRestaurantById(Long id) {
22     Restaurant restaurant = new Restaurant();
     restaurant = restaurantsRepository.findOne(id);
24     return restaurant;
   }
26

   public Long getIdRestaurantbySubdomain(String sub) {
28     Restaurant restaurant = new Restaurant();
     restaurant = restaurantsRepository.getBySubdomain(sub);
30     return restaurant.getId();
   }
32

   public List<Category> getAllCategories() {
34     List<Category> categories = categoriesRepository.findAll();

36     for(Category category: categories) {
         category.setProducts(productsRepository.findAllByCategory(category));
38     }

40     return categories;
   }
42

   @PreAuthorize("hasRole('admin')")
44  public Product addProduct(Product product) {
     return productsRepository.save(product);
46  }

48  @PreAuthorize("hasRole('admin')")
   public void removeProduct(Long categoryId, Long productId) {
50     Product product = productsRepository.findOne(productId);
     productsRepository.delete(product);
52  }

54  public Restaurant addRestaurant(Restaurant restaurant) {
     return restaurantsRepository.save(restaurant);
56  }

58  public void createMasterData() {
```

```

        if (restaurantsRepository.getBySubdomain("default") == null) {
60     Restaurant restaurant = new Restaurant();
        restaurant.setName("iCardapio");
62     restaurant.setSubdomain("default");
        restaurant.setSlogan("Seu Cardapio na Internet");
64     restaurant.setPhone("11 3114-2334");
        restaurant.setAddress("Av Dr Gentil de Moura, 850");
66     restaurant.setCity("Sao Paulo");
        restaurantsRepository.save(restaurant);
68 }
        if (categoriesRepository.count() <= 0) {
70     for(String categoryName: new String[] { "Pizza", "Massas", "Bebidas", "
        Sobremesas" }) {
        categoriesRepository.save(new Category(categoryName));
72     }
        }
74 }
}

```

Código-fonte 34 – Trecho de código da classe *MultiTenantSimpleJpaRepository.java*

```

1  public class MultiTenantSimpleJpaRepository<T, ID extends Serializable> extends
    SimpleJpaRepository<T, ID> {
    private final CurrentTenantResolver<Long> tenantResolver;
3   private final EntityManager em;

5   public MultiTenantSimpleJpaRepository(JpaEntityInformation<T, ?>
    entityInformation, EntityManager em, CurrentTenantResolver<Long>
    tenantResolver) {
    super(entityInformation, em);
7   this.tenantResolver = tenantResolver;
    this.em = em;
9   }

11  public MultiTenantSimpleJpaRepository(Class<T> domainClass, EntityManager em,
    CurrentTenantResolver<Long> tenantResolver) {
    super(domainClass, em);
13  this.tenantResolver = tenantResolver;
    this.em = em;
15  }

17  /*1*/ protected void setCurrentTenant() {
    /*1*/ em.setProperty(PersistenceUnitProperties.MULTITENANT_PROPERTY_DEFAULT,
        tenantResolver.getCurrentTenantId());
19  /*1*/ }

21  @Override
    public <S extends T> S save(S entity) {
23      setCurrentTenant();
        return super.save(entity);
25  }

27  @Override
    public long count() {
29      setCurrentTenant();
        return super.count();
31  }

```

```
33  @Override
    public long count(Specification<T> spec) {
35      setCurrentTenant();
    return super.count(spec);
37  }

39  @Override
    @Transactional
41  public void delete(ID id) {
    setCurrentTenant();
43      super.delete(id);
    }

45
    @Override
47  @Transactional
    public void delete(Iterable<? extends T> arg0) {
49      setCurrentTenant();
    super.delete(arg0);
51  }

53  @Override
    @Transactional
55  public void delete(T entity) {
    setCurrentTenant();
57      super.delete(entity);
    }

59
    @Override
61  @Transactional
    public void deleteAll() {
63      setCurrentTenant();
    super.deleteAll();
65  }

67  @Override
    @Transactional
69  public void deleteAllInBatch() {
    setCurrentTenant();
71      super.deleteAllInBatch();
    }

73
    @Override
75  @Transactional
    public void deleteInBatch(Iterable<T> entities) {
77      setCurrentTenant();
    super.deleteInBatch(entities);
79  }

81  @Override
    public boolean exists(ID arg0) {
83      setCurrentTenant();
    return super.exists(arg0);
85  }

87  @Override
    public List<T> findAll() {
```

```
89     setCurrentTenant();
90     return super.findAll();
91 }

92
93 @Override
94 public List<T> findAll(Iterable<ID> ids) {
95     setCurrentTenant();
96     return super.findAll(ids);
97 }

98
99 @Override
100 public Page<T> findAll(Pageable pageable) {
101     setCurrentTenant();
102     return super.findAll(pageable);
103 }

104
105 @Override
106 public List<T> findAll(Sort sort) {
107     setCurrentTenant();
108     return super.findAll(sort);
109 }

110
111 @Override
112 public Page<T> findAll(Specification<T> spec, Pageable pageable) {
113     setCurrentTenant();
114     return super.findAll(spec, pageable);
115 }

116
117 @Override
118 public List<T> findAll(Specification<T> spec, Sort sort) {
119     setCurrentTenant();
120     return super.findAll(spec, sort);
121 }

122
123 @Override
124 public List<T> findAll(Specification<T> spec) {
125     setCurrentTenant();
126     return super.findAll(spec);
127 }

128
129 @Override
130 public T findOne(ID id) {
131     setCurrentTenant();
132     return super.findOne(id);
133 }

134
135 @Override
136 public T findOne(Specification<T> arg0) {
137     setCurrentTenant();
138     return super.findOne(arg0);
139 }

140
141 @Override
142 @Transactional
143 public void flush() {
144     setCurrentTenant();
145     super.flush();
146 }
```

```
    }
147
    @Override
149    @Transactional
    public <S extends T> List<S> save(Iterable<S> arg0) {
151        setCurrentTenant();
        return super.save(arg0);
153    }

155    @Override
    @Transactional
157    public T saveAndFlush(T entity) {
        setCurrentTenant();
159        return super.saveAndFlush(entity);
    }

161
    @Override
163    public void setLockMetadataProvider(LockMetadataProvider lockMetadataProvider)
        {
        super.setLockMetadataProvider(lockMetadataProvider);
165    }

167 }
```

CASOS DE TESTE E DEFEITOS IDENTIFICADOS NO *MTSHOPFAILS*

Tabela 30 – Casos de teste elaborados com os critérios EPC e BVA para o *MtShopFails*.

PRODUCT						
FIELD: NAME						
Equivalence partitioning criteria (EPC)				Test Cases		
Input	id	Valid Class	id	Invalid Class	Criteria	Expected
Length(l)	1	1<=l<=70	1	0=>l>70	EPC(valid:1,2)	PASS
Data type (td)	2	dt=letters	2	*Special Characters	EPC(invalid:1)	FAIL
Boundary value criteria (BVC)				EPC(invalid:2)	\$%&@%#	FAIL
Boundary				length(l)	mmmmmmmmmm	FAIL
Lim+1	1		71		BVC(1)	FAIL
Lim	2		70		BVC(2)	PASS
Lim-1	3		69		BVC(3)	PASS
Min+1	4		2		BVC(4)	PASS
Min	5		1		BVC(5)	PASS
Min-1	6		0		BVC(6)	FAIL
FIELD: DESCRIPTION						
Equivalence partitioning criteria (EPC)				Test Cases		
Input	id	Valid Class	id	Invalid Class	Criteria	Expected
Length(l)	1	0<=l<=200	1	l>200	EPC(valid:1)	PASS
Boundary value criteria (BVC)				EPC(invalid:1)	TTTTTTTTTTTTTTTT	FAIL
Boundary				length(l)	mmmmmmmmmm	FAIL
Lim+1	1		201		BVC(1)	FAIL
Lim	2		200		BVC(2)	PASS
Lim-1	3		199		BVC(3)	PASS
Min+1	4		2		BVC(4)	PASS
Min	5		1		BVC(5)	PASS
Min-1	6		0		BVC(6)	FAIL
FIELD: PRICE						
Equivalence partitioning criteria (EPC)				Test Cases		
Input	id	Valid Class	id	Invalid Class	Criteria	Expected
Number(n)	1	n>0	1	n<=0	EPC(valid:1,2)	PASS
Data type (td)	2	dt=number	2	*Special Characters	EPC(invalid:1)	FAIL
Boundary value criteria (BVC)				EPC(invalid:2)	bu\$	FAIL
Boundary				length(l)	NULL	FAIL
Lim+1	1				BVC(1)	FAIL
Lim	2				BVC(2)	PASS
Lim-1	3				BVC(3)	PASS
Min+1	4		2		BVC(4)	PASS
Min	5		1		BVC(5)	PASS
Min-1	6		0		BVC(6)	FAIL
FIELD: UNIT						
Equivalence partitioning criteria (EPC)				Test Cases		
Input	id	Valid Class	id	Invalid Class	Criteria	Expected
Number(n)	1	n>0	1	n<=0	EPC(valid:1,2)	PASS
Data type (td)	2	dt=number	2	*Special Characters	EPC(invalid:1)	FAIL
Boundary value criteria (BVC)				EPC(invalid:2)	bu\$	FAIL
Boundary				length(l)	NULL	FAIL
Lim+1	1				BVC(1)	-
Lim	2				BVC(2)	-
Lim-1	3				BVC(3)	-
Min+1	4		2		BVC(4)	2
Min	5		1		BVC(5)	1
Min-1	6		0		BVC(6)	0
FIELD: MANUFACTURER						
Equivalence partitioning criteria (EPC)				Test Cases		
Input	id	Valid Class	id	Invalid Class	Criteria	Expected
Length(l)	1	1<=l<=70	1	0=>l>70	EPC(valid:1,2)	PASS
Data type (td)	2	dt=letters	2	*Special Characters	EPC(invalid:1)	FAIL
Boundary value criteria (BVC)				EPC(invalid:2)	\$%&@%#	FAIL
Boundary				length(l)	mmmmmmmmmm	FAIL
Lim+1	1		71		BVC(1)	PASS
Lim	2		70		BVC(2)	PASS
Lim-1	3		69		BVC(3)	PASS
Min+1	4		2		BVC(4)	PASS
Min	5		1		BVC(5)	PASS
Min-1	6		0		BVC(6)	FAIL

CUSTOMER

FIELD: NAME						
Equivalence partitioning criteria (EPC)				Test Cases		
Input	id	Valid Class	Invalid Class	Criteria	Test data	Expected
Length(l)	1	1=<l<=70	0=>l>70	EPC(valid:1)	Marcos	PASS
Data type (td)	Boundary value criteria (BVC)			EPC(invalid:1)	mmmmmmmmmm	FAIL
	Boundary	length(l)	71	BVC(1)	mmmmmmmmmm	FAIL
				BVC(2)	mmmmmmmmmm	PASS
				BVC(3)	mmmmmmmmmm	PASS
				BVC(4)	Ma	PASS
				BVC(5)	M	PASS
				BVC(6)	empty	FAIL
Lim+1	1					
Lim	2					
Lim-1	3					
Min+1	4					
Min	5					
Min-1	6					

FIELD: EMAIL

FIELD: EMAIL						
Equivalence partitioning criteria (EPC)				Test Cases		
Input	id	Valid Class	Invalid Class	Criteria	Test data	Expected
Length(l)	1	4=<l<=35	4=>l>35	EPC(valid:1,2)	ome@domain.c	PASS
Valid(v)	2	ters)@{letters},{le	atters)@{letters},{	EPC(invalid:1)	mmmmmmmmmm	FAIL
				EPC(invalid:2)	customer.com	FAIL
Boundary	Boundary value criteria (BVC)			BVC(1)	mmmmmmmmmm	FAIL
	length(l)	36		BVC(2)	mmmmmmmmmm	PASS
				BVC(3)	customer@doma	PASS
				BVC(4)	c@.co	PASS
				BVC(5)	c@.c	PASS
				BVC(6)	c@.	FAIL
Lim+1	1					
Lim	2					
Lim-1	3					
Min+1	4					
Min	5					
Min-1	6					

FIELD: NUMBER

FIELD: NUMBER						
Equivalence partitioning criteria (EPC)				Test Cases		
Input	id	Valid Class	Invalid Class	Criteria	Test data	Expected
Length(l)	1	12=<l<=15	12>l>15	EPC(valid:1,2)	\$5 16 98223 1213	PASS
Data type (td)	2	dt=numbers	dt=Special Char	EPC(invalid:1)	\$ 16 98223 1213	FAIL
	3	1=sac<=3	0=sac<3	EPC(invalid:2)	\$5 16 98223 12	FAIL
Area Code Length	Boundary value criteria (BVC)			EPC(invalid:3)	\$34 16 98223 12	FAIL
	length(l)	16		BVC(1)	\$3 16 98223 1213	FAIL
				BVC(2)	\$3 16 98223 1213	PASS
				BVC(3)	\$5 16 98223 121	PASS
				BVC(4)	\$5 16 82223 1213	PASS
				BVC(5)	\$5 16 8222 1213	PASS
				BVC(6)	\$5 16 8222 121	FAIL
Lim+1	1					
Lim	2					
Lim-1	3					
Min+1	4					
Min	5					
Min-1	6					

FIELD: USERNAME						
Equivalence partitioning criteria (EPC)				Test Cases		
Input	id	Valid Class	Invalid Class	Criteria	Test data	Expected
Length(l)	1	1=<l<=10	0=>l>10	EPC(valid:1,2)	marcos	PASS
Data type (td)	2	dt=letters	=Special Character	EPC(invalid:1)	marcosssssss	FAIL
				EPC(invalid:2)	marco\$	FAIL
Boundary	Boundary value criteria (BVC)			BVC(1)	marcosssss	FAIL
	length(l)	11		BVC(2)	marcosssss	PASS
				BVC(3)	marcosssss	PASS
				BVC(4)	Ma	PASS
				BVC(5)	M	PASS
				BVC(6)	empty	FAIL
Lim+1	1					
Lim	2					
Lim-1	3					
Min+1	4					
Min	5					
Min-1	6					

FIELD: PASSWORD

FIELD: PASSWORD						
Equivalence partitioning criteria (EPC)				Test Cases		
Input	id	Valid Class	Invalid Class	Criteria	Test data	Expected
Length(l)	1	4=<l<=10	3=>l>10	EPC(valid:1,2)	pass123	PASS
Boundary	Boundary value criteria (BVC)			EPC(invalid:1)	pas	FAIL
	length(l)	11		BVC(1)	passssssssss	FAIL
				BVC(2)	passssssss	PASS
				BVC(3)	passssssss	PASS
				BVC(4)	pa	PASS
				BVC(5)	p	PASS
Lim+1	1					
Lim	2					
Lim-1	3					
Min+1	4					
Min	5					
Min-1	6					

CUSTOMER - BILLING and SHIPPING

FIELD: STREET						
Equivalence partitioning criteria (EPC)				Test Cases		
Input	id	Valid Class	Invalid Class	Criteria	Test data	Expected
Length(l)	1	1=<k<=70	0=>l>70	EPC(valid:1)	venue 30, New York	PASS
Boundary	Boundary value criteria (BVC)			EPC(invalid:1)	0000000000000000	FAIL
	id	length(l)		BVC(1)	eeeeeeeeeeeeee	FAIL
	Lim+1	1	71	BVC(2)	eeeeeeeeeeeeee	PASS
	Lim	2	70	BVC(3)	eeeeeeeeeeeeee	PASS
	Lim-1	3	69	BVC(4)	eeeeeeeeeeeeee	PASS
	Min+1	4	2	BVC(5)	A	PASS
	Min	5	1	BVC(6)	empty	PASS
Min-1	6		0			

FIELD: APARTMENT NUMBER						
Equivalence partitioning criteria (EPC)				Test Cases		
Input	id	Valid Class	Invalid Class	Criteria	Test data	Expected
Length(l)	1	1=<k<=40	0=>l>40	EPC(valid:1,2)	123	PASS
Data type (td)	2	dt=numbers	special Characters//	EPC(invalid:1)	32121654565465	FAIL
Boundary	Boundary value criteria (BVC)			EPC(invalid:2)	ll46	FAIL
	id	length(l)		BVC(1)	111111111111	FAIL
	Lim+1	1	11	BVC(2)	111111111111	PASS
	Lim	2	10	BVC(3)	111111111111	PASS
	Lim-1	3	9	BVC(4)	11	PASS
	Min+1	4	2	BVC(5)	1	PASS
	Min	5	1	BVC(6)	empty	PASS
Min-1	6		0			

FIELD: CITY						
Equivalence partitioning criteria (EPC)				Test Cases		
Input	id	Valid Class	Invalid Class	Criteria	Test data	Expected
Length(l)	1	1=<k<=70	0=>l>70	EPC(valid:1,2)	New York	PASS
Boundary	Boundary value criteria (BVC)			EPC(invalid:1)	kkkkkkkkkkkkkk	FAIL
	id	length(l)		BVC(1)	oooooooooooooooo	FAIL
	Lim+1	1	71	BVC(2)	oooooooooooooooo	PASS
	Lim	2	70	BVC(3)	oooooooooooooooo	PASS
	Lim-1	3	69	BVC(4)	Ne	PASS
	Min+1	4	2	BVC(5)	C	PASS
	Min	5	1	BVC(6)	empty	PASS
Min-1	6		0			

FIELD: STATE						
Equivalence partitioning criteria (EPC)				Test Cases		
Input	id	Valid Class	Invalid Class	Criteria	Test data	Expected
Length(l)	1	1=<k<=70	0=>l>70	EPC(valid:1,2)	California	PASS
Boundary	Boundary value criteria (BVC)			EPC(invalid:1)	aaaaaaaaaaaaaa	FAIL
	id	length(l)		BVC(1)	aaaaaaaaaaaaaa	FAIL
	Lim+1	1	71	BVC(2)	aaaaaaaaaaaaaa	PASS
	Lim	2	70	BVC(3)	aaaaaaaaaaaaaa	PASS
	Lim-1	3	69	BVC(4)	Ca	PASS
	Min+1	4	2	BVC(5)	C	PASS
	Min	5	1	BVC(6)	empty	PASS
Min-1	6		0			

FIELD: COUNTRY						
Equivalence partitioning criteria (EPC)				Test Cases		
Input	id	Valid Class	Invalid Class	Criteria	Test data	Expected
Length(l)	1	1=<k<=70	0=>l>70	EPC(valid:1,2)	Belgica	PASS
Boundary	Boundary value criteria (BVC)			EPC(invalid:1)	aaaaaaaaaaaaaa	FAIL
	id	length(l)		BVC(1)	aaaaaaaaaaaaaa	FAIL
	Lim+1	1	71	BVC(2)	aaaaaaaaaaaaaa	PASS
	Lim	2	70	BVC(3)	aaaaaaaaaaaaaa	PASS
	Lim-1	3	69	BVC(4)	Bé	PASS
	Min+1	4	2	BVC(5)	B	PASS
	Min	5	1	BVC(6)	empty	PASS
Min-1	6		0			

FIELD: ZIP CODE						
Equivalence partitioning criteria (EPC)				Test Cases		
Input	id	Valid Class	Invalid Class	Criteria	Test data	Expected
Length(l)	1	1=<k<=16	0=>l>16	EPC(valid:1,2)	13566550	PASS
Data type (td)	2	dt=numbers	special Characters//	EPC(invalid:1)	99999999999999	FAIL
Boundary	Boundary value criteria (BVC)			EPC(invalid:2)	135666550	FAIL
	id	length(l)		BVC(1)	5665502021497	FAIL
	Lim+1	1	17	BVC(2)	95665502021497	PASS
	Lim	2	16	BVC(3)	3566550202149	PASS
	Lim-1	3	15	BVC(4)	11	PASS
	Min+1	4	2	BVC(5)	1	PASS
	Min	5	1	BVC(6)	empty	PASS
Min-1	6		0			

Tabela 31 – Defeitos identificados no MtShopFails.

MtShop				MtShop				MtShop				MtShop			
CustomerNameTest				CustomerNameTest				CustomerNameTest				CustomerNameTest			
TestCase	Fault Type	Description log	Situation	TestCase	Fault Type	Description log	Situation	TestCase	Fault Type	Description log	Situation	TestCase	Fault Type	Description log	Situation
test01	SUE	300.035 org.openqa.selenium.SUE.exception	ERROR	test01	SUE	299.999 org.openqa.selenium.SUE.exception	ERROR	test01	SUE	300.016 org.openqa.selenium.SUE.exception	ERROR	test01	SUE	300.016 org.openqa.selenium.SUE.exception	ERROR
test02	IE	30.774 java.lang.Assert The field does not	FAILURE	test02	IE	30.774 java.lang.Assert The field does not	FAILURE	test02	IE	22.248 java.lang.Assert The field does not	FAILURE	test02	IE	22.248 java.lang.Assert The field does not	FAILURE
test03	IE	15.881 java.lang.Assert The field does not	FAILURE	test03	IE	15.881 java.lang.Assert The field does not	FAILURE	test03	IE	7.715 java.lang.Assert The field does not	FAILURE	test03	IE	7.715 java.lang.Assert The field does not	FAILURE
test04	NONE	19.767 NO LOG	PASS	test04	NONE	12.012 NO LOG	PASS	test04	NONE	37.593 org.openqa.selenium.SUE.exception	ERROR	test04	NONE	37.593 org.openqa.selenium.SUE.exception	ERROR
test05	NONE	20.348 NO LOG	PASS	test05	NONE	12.03 NO LOG	PASS	test05	SUE	37.454 org.openqa.selenium.SUE.exception	ERROR	test05	SUE	37.454 org.openqa.selenium.SUE.exception	ERROR
test06	NONE	21.477 org.openqa.selenium.SUE.exception	ERROR	test06	RCMI	11.791 org.junit.CFefailure: expected <	FAILURE	test06	NONE	9.662 NO LOG	PASS	test06	NONE	9.662 NO LOG	PASS
test07	SUE	44.331 org.openqa.selenium.SUE.exception	ERROR	test07	NONE	11.791 NO LOG	PASS	test07	CPE	8.356 org.junit.CFefailure: expected <	FAILURE	test07	CPE	8.356 org.junit.CFefailure: expected <	FAILURE
test08	NONE	35.853 NO LOG	PASS	test08	NONE	36.382 NO LOG	PASS	test08	NONE	66.052 NO LOG	PASS	test08	NONE	66.052 NO LOG	PASS
CustomerEmailTest				CustomerEmailTest				CustomerEmailTest				CustomerEmailTest			
TestCase	Fault Type	Description log	Situation	TestCase	Fault Type	Description log	Situation	TestCase	Fault Type	Description log	Situation	TestCase	Fault Type	Description log	Situation
test01	NONE	97.883 NO LOG	PASS	test01	NONE	14.85 NO LOG	PASS	test01	SUE	41.53 org.openqa.selenium.SUE.exception	ERROR	test01	SUE	41.53 org.openqa.selenium.SUE.exception	ERROR
test02	MCE	67.001 Unable to locate	ERROR	test02	IE	4.871 java.lang.Assert The field does not	FAILURE	test02	IE	8.228 java.lang.Assert The field does not	FAILURE	test02	IE	8.228 java.lang.Assert The field does not	FAILURE
test03	NONE	37.949 NO LOG	PASS	test03	NONE	35.168 NO LOG	PASS	test03	NONE	5.545 NO LOG	PASS	test03	NONE	5.545 NO LOG	PASS
test04	MCE	68.227 Unable to locate	ERROR	test04	IE	5.904 java.lang.Assert The field does not	FAILURE	test04	IE	40.295 org.openqa.selenium.SUE.exception	ERROR	test04	IE	40.295 org.openqa.selenium.SUE.exception	ERROR
test05	NONE	92.178 NO LOG	PASS	test05	NONE	11.469 NO LOG	PASS	test05	SUE	19.284 NO LOG	PASS	test05	SUE	19.284 NO LOG	PASS
test06	NONE	95.443 NO LOG	PASS	test06	NONE	10.253 NO LOG	PASS	test06	NONE	93.433 NO LOG	PASS	test06	NONE	93.433 NO LOG	PASS
test07	NONE	95.742 NO LOG	PASS	test07	NONE	92.468 NO LOG	PASS	test07	NONE	95.436 NO LOG	PASS	test07	NONE	95.436 NO LOG	PASS
test08	NONE	72.079 NO LOG	PASS	test08	NONE	93.631 NO LOG	PASS	test08	NONE	121.033 NO LOG	PASS	test08	NONE	121.033 NO LOG	PASS
test09	MCE	2.854 Unable to locate	ERROR	test09	NONE	66.808 NO LOG	PASS	test09	NONE	121.033 NO LOG	PASS	test09	NONE	121.033 NO LOG	PASS
CustomerPasswordTest				CustomerPasswordTest				CustomerPasswordTest				CustomerPasswordTest			
TestCase	Fault Type	Description log	Situation	TestCase	Fault Type	Description log	Situation	TestCase	Fault Type	Description log	Situation	TestCase	Fault Type	Description log	Situation
test01	MCE	28.15 Unable to locate	ERROR	test01	RCMI	28.157 org.junit.CFefailure: expected <	FAILURE	test01	SUE	51.352 org.openqa.selenium.SUE.exception	ERROR	test01	SUE	51.352 org.openqa.selenium.SUE.exception	ERROR
test02	NONE	34.484 NO LOG	PASS	test02	NONE	34.459 NO LOG	PASS	test02	NONE	34.7 NO LOG	PASS	test02	NONE	34.7 NO LOG	PASS
test03	NONE	34.16 NO LOG	PASS	test03	NONE	36.295 NO LOG	PASS	test03	NONE	33.077 NO LOG	PASS	test03	NONE	33.077 NO LOG	PASS
test04	CPE	32.444 org.junit.CFefailure: expected <	FAILURE	test04	NONE	16.033 NO LOG	PASS	test04	CPE	13.228 org.junit.CFefailure: expected <	FAILURE	test04	CPE	13.228 org.junit.CFefailure: expected <	FAILURE
test05	NONE	22.05 NO LOG	PASS	test05	SUE	39.673 org.openqa.selenium.SUE.exception	ERROR	test05	NONE	12.727 NO LOG	PASS	test05	NONE	12.727 NO LOG	PASS
test06	NONE	94.071 NO LOG	PASS	test06	NONE	93.642 NO LOG	PASS	test06	NONE	93.125 NO LOG	PASS	test06	NONE	93.125 NO LOG	PASS
test07	NONE	96.706 NO LOG	PASS	test07	NONE	95.924 NO LOG	PASS	test07	NONE	97.393 NO LOG	PASS	test07	NONE	97.393 NO LOG	PASS
test08	NONE	34.484 NO LOG	PASS	test08	NONE	34.481 NO LOG	PASS	test08	NONE	43.891 NO LOG	PASS	test08	NONE	43.891 NO LOG	PASS
CustomerPhoneTest				CustomerPhoneTest				CustomerPhoneTest				CustomerPhoneTest			
TestCase	Fault Type	Description log	Situation	TestCase	Fault Type	Description log	Situation	TestCase	Fault Type	Description log	Situation	TestCase	Fault Type	Description log	Situation
test01	CPE	27.035 org.junit.CFefailure: expected <	FAILURE	test01	SUE	41.407 org.openqa.selenium.SUE.exception	ERROR	test01	NONE	18.338 NO LOG	PASS	test01	NONE	18.338 NO LOG	PASS
test02	IE	16.387 java.lang.Assert The field does not	FAILURE	test02	IE	8.645 java.lang.Assert The field does not	FAILURE	test02	IE	8.006 java.lang.Assert The field does not	FAILURE	test02	IE	8.006 java.lang.Assert The field does not	FAILURE
test03	IE	15.016 java.lang.Assert The field does not	FAILURE	test03	IE	7.699 java.lang.Assert The field does not	FAILURE	test03	IE	6.239 java.lang.Assert The field does not	FAILURE	test03	IE	6.239 java.lang.Assert The field does not	FAILURE
test04	IE	15.191 java.lang.Assert The field does not	FAILURE	test04	IE	6.353 java.lang.Assert The field does not	FAILURE	test04	IE	10.602 java.lang.Assert The field does not	FAILURE	test04	IE	10.602 java.lang.Assert The field does not	FAILURE
test05	IE	20.693 java.lang.Assert The field does not	FAILURE	test05	NONE	5.817 java.lang.Assert The field does not	FAILURE	test05	SUE	39.359 org.openqa.selenium.SUE.exception	ERROR	test05	SUE	39.359 org.openqa.selenium.SUE.exception	ERROR
test06	RCMI	20.693 java.lang.Assert The field does not	FAILURE	test06	NONE	13.317 NO LOG	PASS	test06	NONE	12.881 NO LOG	PASS	test06	NONE	12.881 NO LOG	PASS
test07	NONE	20.223 NO LOG	PASS	test07	SUE	36.989 org.openqa.selenium.SUE.exception	ERROR	test07	NONE	10.335 NO LOG	PASS	test07	NONE	10.335 NO LOG	PASS
test08	NONE	23.087 NO LOG	PASS	test08	SUE	38.193 org.openqa.selenium.SUE.exception	ERROR	test08	NONE	10.327 org.junit.CFefailure: expected <	FAILURE	test08	NONE	10.327 org.junit.CFefailure: expected <	FAILURE
test09	RCMI	20.612 org.junit.CFefailure: expected <	FAILURE	test09	CPE	14.49 org.junit.CFefailure: expected <	FAILURE	test09	CPE	5.34 java.lang.Assert The field does not	FAILURE	test09	CPE	5.34 java.lang.Assert The field does not	FAILURE
test10	IE	15.386 java.lang.Assert The field does not	FAILURE	test10	IE	5.9 java.lang.Assert The field does not	FAILURE	test10	IE	5.34 java.lang.Assert The field does not	FAILURE	test10	IE	5.34 java.lang.Assert The field does not	FAILURE
CustomerUsernameTest				CustomerUsernameTest				CustomerUsernameTest				CustomerUsernameTest			
TestCase	Fault Type	Description log	Situation	TestCase	Fault Type	Description log	Situation	TestCase	Fault Type	Description log	Situation	TestCase	Fault Type	Description log	Situation
test01	NONE	92.624 NO LOG	PASS	test01	SUE	38.928 org.openqa.selenium.SUE.exception	ERROR	test01	NONE	12.772 NO LOG	PASS	test01	NONE	12.772 NO LOG	PASS
test02	NONE	33.282 NO LOG	PASS	test02	IE	6.325 java.lang.Assert The field does not	FAILURE	test02	NONE	6.49 java.lang.Assert The field does not	FAILURE	test02	NONE	6.49 java.lang.Assert The field does not	FAILURE
test03	NONE	32.854 NO LOG	PASS	test03	IE	8.247 java.lang.Assert The field does not	FAILURE	test03	IE	6.239 java.lang.Assert The field does not	FAILURE	test03	IE	6.239 java.lang.Assert The field does not	FAILURE
test04	NONE	33.258 NO LOG	PASS	test04	IE	8.986 java.lang.Assert The field does not	FAILURE	test04	IE	6.692 java.lang.Assert The field does not	FAILURE	test04	IE	6.692 java.lang.Assert The field does not	FAILURE
test05	NONE	35.846 NO LOG	PASS	test05	NONE	16.033 NO LOG	PASS	test05	NONE	38.073 NO LOG	PASS	test05	NONE	38.073 NO LOG	PASS
test06	NONE	94.757 NO LOG	PASS	test06	SUE	38.559 org.openqa.selenium.SUE.exception	ERROR	test06	SUE	38.073 NO LOG	PASS	test06	SUE	38.073 NO LOG	PASS
test07	NONE	94.757 NO LOG	PASS	test07	NONE	12.827 NO LOG	PASS	test07	NONE	10.515 NO LOG	PASS	test07	NONE	10.515 NO LOG	PASS
test08	NONE	99.997 NO LOG	PASS	test08	NONE	11.145 NO LOG	PASS	test08	NONE	11.718 NO LOG	PASS	test08	NONE	11.718 NO LOG	PASS
test09	MCE	65.459 Unable to locate	ERROR	test09	NONE	59.654 NO LOG	PASS	test09	NONE	55.96 NO LOG	PASS	test09	NONE	55.96 NO LOG	PASS
BillingApartmentNumberTest				BillingApartmentNumberTest				BillingApartmentNumberTest				BillingApartmentNumberTest			
TestCase	Fault Type	Description log	Situation	TestCase	Fault Type	Description log	Situation	TestCase	Fault Type	Description log	Situation	TestCase	Fault Type	Description log	Situation
test01	SUE	19.332 org.openqa.selenium.SUE.exception	ERROR	test01	SUE	41.256 org.openqa.selenium.SUE.exception	ERROR	test01	NONE	14.331 NO LOG	PASS	test01	NONE	14.331 NO LOG	PASS
test02	IE	16.628 java.lang.Assert The field does not	FAILURE	test02	IE	5.899 java.lang.Assert The field does not	FAILURE	test02	IE	6.917 SE	SEQUENTIAL	test02	IE	6.917 SE	SEQUENTIAL
test03	IE	16.228 java.lang.Assert The field does not	FAILURE	test03	IE	6.778 java.lang.Assert The field does not	FAILURE	test03	IE	7.482 SE	SEQUENTIAL	test03	IE	7.482 SE	SEQUENTIAL
test04	IE	14.953 java.lang.Assert The field does not	FAILURE	test04	IE	8.042 java.lang.Assert The field does not	FAILURE	test04	IE	11.012 java.lang.Assert The field does not	FAILURE	test04	IE	11.012 java.lang.Assert The field does not	FAILURE
test05	NONE	22.135 NO LOG	PASS	test05	NONE	10.879 NO LOG	PASS	test05	CPE	11.012 java.lang.Assert The field does not	FAILURE	test05	CPE	11.012 java.lang.Assert The field does not	FAILURE
test06	NONE	20.034 NO LOG	PASS	test06	NONE	12.231 NO LOG	PASS	test06	NONE	10.515 NO LOG	PASS	test06	NONE	10.515 NO LOG	PASS
test07	NONE	23.518 NO LOG	PASS	test07	SUE	38.698 org.openqa.selenium.SUE.exception	ERROR	test07	SUE	39.042 org.openqa.selenium.SUE.exception	ERROR	test07	SUE	39.042 org.openqa.selenium.SUE.exception	ERROR
test08	NONE	21.716 NO LOG	PASS	test08	NONE	13.67 NO LOG	PASS	test08	NONE	11.233 NO LOG	PASS	test08	NONE	11.233 NO LOG	PASS
test09	NONE	23.187 NO LOG	PASS	test09	NONE	26.029 NO LOG	PASS	test09	NONE	14.105 NO LOG	PASS	test09	NONE	14.105 NO LOG	PASS

BillingCityTest			BillingCityTest			BillingCityTest			BillingCityTest				
Test Case	Fault	Fault Type	Time (s)	Description log	Details	Situation	Test Case	Fault	Fault Type	Time (s)	Description log	Details	Situation
ts001	CPE	SEQUENTIAL	21.515	org.junit.CPEFailure: expected<...>	-	FAILURE	ts001	SUE	EXCEPTION	30.051	org.opentqa.selenium.SUEException	-	ERROR
ts002	IE	SEQUENTIAL	19.854	java.lang.AssertionThe field does not	-	FAILURE	ts002	IE	SEQUENTIAL	8.854	java.lang.AssertionThe field does not	-	FAILURE
ts003	IE	SEQUENTIAL	16.715	java.lang.AssertionThe field does not	-	FAILURE	ts003	IE	SEQUENTIAL	12.643	org.opentqa.selenium.SUEException	-	FAILURE
ts004	CPE	SEQUENTIAL	21.233	org.junit.CPEFailure: expected<...>	-	FAILURE	ts004	NONE	NO FAULT	14.027	org.opentqa.selenium.SUEException	-	PASS
ts005	CPE	NO FAULT	24.552	NO LOG	-	PASS	ts005	SUE	EXCEPTION	48.499	org.opentqa.selenium.SUEException	-	ERROR
ts006	SUE	EXCEPTION	48.849	org.opentqa.selenium.SUEException	-	ERROR	ts006	NONE	NO FAULT	14.358	NO LOG	-	PASS
ts007	NONE	NO FAULT	19.899	NO LOG	-	PASS	ts007	NONE	NO FAULT	13.2	NO LOG	-	PASS
ts008	NONE	NO FAULT	22.246	NO LOG	-	PASS	ts008	NONE	NO FAULT	14.223	NO LOG	-	PASS
BillingCountryTest			BillingCountryTest			BillingCountryTest			BillingCountryTest				
Test Case	Fault	Fault Type	Time (s)	Description log	Details	Situation	Test Case	Fault	Fault Type	Time (s)	Description log	Details	Situation
ts001	NONE	NO FAULT	24.397	NO LOG	-	PASS	ts001	SUE	EXCEPTION	40.889	org.opentqa.selenium.SUEException	-	ERROR
ts002	IE	SEQUENTIAL	17.462	java.lang.AssertionThe field does not	-	FAILURE	ts002	IE	SEQUENTIAL	8.497	java.lang.AssertionThe field does not	-	FAILURE
ts003	IE	SEQUENTIAL	15.633	java.lang.AssertionThe field does not	-	FAILURE	ts003	IE	SEQUENTIAL	7.883	java.lang.AssertionThe field does not	-	FAILURE
ts004	NONE	NO FAULT	20.633	NO LOG	-	PASS	ts004	NONE	NO FAULT	14.489	NO LOG	-	PASS
ts005	NONE	EXCEPTION	48.733	org.opentqa.selenium.SUEException	-	ERROR	ts005	SUE	EXCEPTION	11.884	NO LOG	-	PASS
ts006	SUE	NO FAULT	19.899	NO LOG	-	PASS	ts006	NONE	NO FAULT	13.439	org.junit.CPEFailure: expected<...>	-	FAILURE
ts007	NONE	NO FAULT	22.627	NO LOG	-	PASS	ts007	CPE	SEQUENTIAL	16.353	org.junit.CPEFailure: expected<...>	-	FAILURE
ts008	NONE	NO FAULT	23.161	NO LOG	-	PASS	ts008	NONE	NO FAULT	14.533	NO LOG	-	PASS
BillingStateTest			BillingStateTest			BillingStateTest			BillingStateTest				
Test Case	Fault	Fault Type	Time (s)	Description log	Details	Situation	Test Case	Fault	Fault Type	Time (s)	Description log	Details	Situation
ts001	CPE	SEQUENTIAL	19.854	org.opentqa.selenium.SUEException	-	FAILURE	ts001	SUE	EXCEPTION	40.887	org.opentqa.selenium.SUEException	-	ERROR
ts002	IE	SEQUENTIAL	18.195	java.lang.AssertionThe field does not	-	FAILURE	ts002	IE	SEQUENTIAL	11.352	java.lang.AssertionThe field does not	-	FAILURE
ts003	IE	SEQUENTIAL	15.633	java.lang.AssertionThe field does not	-	FAILURE	ts003	IE	SEQUENTIAL	10.051	java.lang.AssertionThe field does not	-	FAILURE
ts004	NONE	NO FAULT	25.058	NO LOG	-	PASS	ts004	NONE	NO FAULT	14.052	NO LOG	-	PASS
ts005	NONE	NO FAULT	22.218	NO LOG	-	PASS	ts005	RCMI	CONCURRENT	14.952	org.opentqa.selenium.SUEException	-	FAILURE
ts006	CPE	NO FAULT	23.699	org.opentqa.selenium.SUEException	-	FAILURE	ts006	SUE	EXCEPTION	40.896	org.opentqa.selenium.SUEException	-	ERROR
ts007	NONE	NO FAULT	20.921	NO LOG	-	PASS	ts007	SUE	EXCEPTION	39.207	org.opentqa.selenium.SUEException	-	ERROR
ts008	NONE	NO FAULT	30.539	NO LOG	-	PASS	ts008	SUE	EXCEPTION	43.179	org.opentqa.selenium.SUEException	-	ERROR
BillingStreetTest			BillingStreetTest			BillingStreetTest			BillingStreetTest				
Test Case	Fault	Fault Type	Time (s)	Description log	Details	Situation	Test Case	Fault	Fault Type	Time (s)	Description log	Details	Situation
ts001	NONE	NO FAULT	28.48	NO LOG	-	PASS	ts001	SUE	EXCEPTION	43.77	org.opentqa.selenium.SUEException	-	ERROR
ts002	IE	SEQUENTIAL	20.241	java.lang.AssertionThe field does not	-	FAILURE	ts002	IE	SEQUENTIAL	12.361	java.lang.AssertionThe field does not	-	FAILURE
ts003	IE	SEQUENTIAL	21.641	java.lang.AssertionThe field does not	-	FAILURE	ts003	IE	SEQUENTIAL	9.719	java.lang.AssertionThe field does not	-	FAILURE
ts004	CPE	SEQUENTIAL	25.058	org.opentqa.selenium.SUEException	-	FAILURE	ts004	NONE	NO FAULT	16.812	NO LOG	-	PASS
ts005	NONE	NO FAULT	25.058	NO LOG	-	PASS	ts005	SUE	EXCEPTION	40.114	org.opentqa.selenium.SUEException	-	ERROR
ts006	NONE	NO FAULT	24.562	NO LOG	-	PASS	ts006	NONE	NO FAULT	13.831	NO LOG	-	PASS
ts007	SUE	EXCEPTION	49.512	org.opentqa.selenium.SUEException	-	ERROR	ts007	NONE	NO FAULT	15.955	NO LOG	-	PASS
ts008	NONE	NO FAULT	25.294	NO LOG	-	PASS	ts008	SUE	EXCEPTION	40.892	org.opentqa.selenium.SUEException	-	ERROR
BillingZipCodeTest			BillingZipCodeTest			BillingZipCodeTest			BillingZipCodeTest				
Test Case	Fault	Fault Type	Time (s)	Description log	Details	Situation	Test Case	Fault	Fault Type	Time (s)	Description log	Details	Situation
ts001	NONE	NO FAULT	23.723	NO LOG	-	PASS	ts001	NONE	NO FAULT	17.106	NO LOG	-	PASS
ts002	IE	SEQUENTIAL	19.866	java.lang.AssertionThe field does not	-	FAILURE	ts002	IE	SEQUENTIAL	9.854	java.lang.AssertionThe field does not	-	FAILURE
ts003	IE	SEQUENTIAL	18.755	java.lang.AssertionThe field does not	-	FAILURE	ts003	IE	SEQUENTIAL	9.857	java.lang.AssertionThe field does not	-	FAILURE
ts004	SUE	EXCEPTION	19.755	java.lang.AssertionThe field does not	-	FAILURE	ts004	IE	SEQUENTIAL	9.857	java.lang.AssertionThe field does not	-	FAILURE
ts005	NONE	NO FAULT	22.779	org.opentqa.selenium.SUEException	-	ERROR	ts005	NONE	NO FAULT	13.687	NO LOG	-	PASS
ts006	NONE	NO FAULT	27.261	NO LOG	-	PASS	ts006	NONE	NO FAULT	17.632	NO LOG	-	PASS
ts007	NONE	NO FAULT	24.333	NO LOG	-	PASS	ts007	NONE	NO FAULT	15.085	NO LOG	-	PASS
ts008	NONE	NO FAULT	26.991	NO LOG	-	PASS	ts008	SUE	EXCEPTION	41.951	org.opentqa.selenium.SUEException	-	ERROR
ts009	NONE	NO FAULT	31.773	NO LOG	-	PASS	ts009	NONE	NO FAULT	26.514	NO LOG	-	PASS
ShippingApartmentNumberTest			ShippingApartmentNumberTest			ShippingApartmentNumberTest			ShippingApartmentNumberTest				
Test Case	Fault	Fault Type	Time (s)	Description log	Details	Situation	Test Case	Fault	Fault Type	Time (s)	Description log	Details	Situation
ts001	NONE	NO FAULT	23.24	NO LOG	-	PASS	ts001	NONE	NO FAULT	19.195	NO LOG	-	PASS
ts002	IE	SEQUENTIAL	18.55	java.lang.AssertionThe field does not	-	FAILURE	ts002	IE	SEQUENTIAL	10.641	java.lang.AssertionThe field does not	-	FAILURE
ts003	IE	SEQUENTIAL	32.976	java.lang.AssertionThe field does not	-	FAILURE	ts003	IE	SEQUENTIAL	12.395	java.lang.AssertionThe field does not	-	FAILURE
ts004	IE	SEQUENTIAL	21.345	java.lang.AssertionThe field does not	-	FAILURE	ts004	IE	SEQUENTIAL	17.997	java.lang.AssertionThe field does not	-	FAILURE
ts005	NONE	NO FAULT	27.43	NO LOG	-	PASS	ts005	NONE	NO FAULT	22.671	NO LOG	-	PASS
ts006	NONE	EXCEPTION	52.894	org.opentqa.selenium.SUEException	-	ERROR	ts006	RCMI	CONCURRENT	18.917	org.opentqa.selenium.SUEException	-	FAILURE
ts007	NONE	NO FAULT	24.515	NO LOG	-	PASS	ts007	NONE	NO FAULT	16.903	NO LOG	-	PASS
ts008	NONE	NO FAULT	27.315	NO LOG	-	PASS	ts008	NONE	NO FAULT	21.389	NO LOG	-	PASS
ts009	IE	SEQUENTIAL	22.106	java.lang.AssertionThe field does not	-	FAILURE	ts009	IE	SEQUENTIAL	11.789	java.lang.AssertionThe field does not	-	FAILURE
ShippingCityTest			ShippingCityTest			ShippingCityTest			ShippingCityTest				
Test Case	Fault	Fault Type	Time (s)	Description log	Details	Situation	Test Case	Fault	Fault Type	Time (s)	Description log	Details	Situation
ts001	NONE	NO FAULT	27.999	NO LOG	-	PASS	ts001	NONE	NO FAULT	16.061	NO LOG	-	PASS
ts002	NONE	NO FAULT	27.999	NO LOG	-	PASS	ts002	NONE	NO FAULT	16.061	NO LOG	-	PASS

CASOS DE TESTE E DEFEITOS IDENTIFICADOS NO *ICARDAPIO*

Tabela 32 – Casos de teste elaborados com os critérios EPC e BVA para o *iCardapio*.

PRODUCT

FIELD: NAME						
Equivalence partitioning criteria (EPC)				Test Cases		
Input	id	Valid Class	id	Invalid Class	Criteria	Expected
Length(l)	1	1=<l<=30	1	0=>l>30	EPC(valid:1,2)	PASS
Data type (td)	2	dt=letters	2	=Special Character	EPC(invalid:1)	FAIL
Boundary value criteria (BVC)				Criteria	Test data	Expected
Boundary	id	length(l)		EPC(invalid:2)	!@#\$%^&*~	FAIL
Lim+1	1		31	BVC(1)	{a}*31	FAIL
Lim	2		30	BVC(2)	{a}*30	PASS
Lim-1	3		29	BVC(3)	{a}*29	PASS
Min+1	4		2	BVC(4)	Be	PASS
Min	5		1	BVC(5)	B	PASS
Min-1	6		0	BVC(6)	empty	FAIL

FIELD: DESCRIPTION						
Equivalence partitioning criteria (EPC)				Test Cases		
Input	id	Valid Class	id	Invalid Class	Criteria	Expected
Length(l)	1	1=<l<=255	1	0=>l>255	EPC(valid:1,2)	PASS
Boundary	id	length(l)		EPC(invalid:1)	great_beer // \	FAIL
Lim+1	1		121	BVC(1)	{a}*300	FAIL
Lim	2		120	BVC(2)	{a}*256	FAIL
Lim-1	3		119	BVC(3)	{a}*255	PASS
Min+1	4		2	BVC(4)	{a}*254	PASS
Min	5		1	BVC(5)	Ag	PASS
Min-1	6		0	BVC(6)	A	PASS
					empty	FAIL

FIELD: PRICE						
Equivalence partitioning criteria (EPC)				Test Cases		
Input	id	Valid Class	id	Invalid Class	Criteria	Expected
Length(l)	1	1=<l<=10	1	0=>l>10	EPC(valid:1,2)	PASS
Length after ", "(t)	2	0=<la<=2	2	0>l>2	EPC(invalid:1)	FAIL
Data type (td)	3	dt=lettres	3	dt=lettres	EPC(invalid:2)	FAIL
Boundary value criteria (BVC)				Criteria	Test data	Expected
Boundary	id	length(l)		EPC(invalid:3)	500,454	FAIL
Lim+1	1		11	BVC(l){1}	aaaa	FAIL
Lim	2		10	BVC(l){2}	111111111111	FAIL
Lim-1	3		9	BVC(l){3}	222222222222	PASS
Min+1	4		2	BVC(l){4}	333333333333	PASS
Min	5		1	BVC(l){5}	1	PASS
Min-1	6		0	BVC(l){6}	empty	FAIL
Boundary	id	length(la)		BVC(la){1}	50,444	FAIL
Lim+1	1		3	BVC(la){2}	50,44	PASS
Lim	2		2	BVC(la){3}	50,4	PASS
Lim-1	3		1	BVC(la){4}	123,4	PASS
Min+1	4		1	BVC(la){5}	50	PASS
Min	5		0	BVC(la){6}	empty	FAIL
Min-1	6		-1			

RESTAURANT

FIELD: NAME				
Equivalence partitioning criteria (EPC)			Test Cases	
Input	id	Valid Class	Invalid Class	Expected
Length(l)	1	1=<l<=35	1 0=>l>35	PASS
Data type (td)	2	dt=letters	2 -Special Character	FAIL
Bondary value criteria (BVC)				
Bondary	id	length(l)		
Lim+1	1	(a)*36	BVC(1)	FAIL
Lim	2	35	BVC(2)	PASS
Lim-1	3	34	BVC(3)	PASS
Lim+1	4	2	BVC(4)	PASS
Min	5	1	BVC(5)	PASS
Min-1	6	0	BVC(6)	FAIL

FIELD: PHONE NUMBER				
Equivalence partitioning criteria (EPC)			Test Cases	
Input	id	Valid Class	Invalid Class	Expected
Length(l)	1	12=<l<=14	1 12=>l>14	PASS
Data type (td)	2	dt=numbers	2 ters/Special Char	FAIL
Area Code Length	3	1=<ad<=3	3 0>ad>3	FAIL
Bondary value criteria (BVC)				
Bondary	id	length(l)		
Lim+1	1	(a)*36	BVC(1)	FAIL
Lim	2	35	BVC(2)	PASS
Lim-1	3	34	BVC(3)	PASS
Lim+1	4	2	BVC(4)	PASS
Min	5	1	BVC(5)	PASS
Min-1	6	11	BVC(6)	FAIL

FIELD: SLOGAN				
Equivalence partitioning criteria (EPC)			Test Cases	
Input	id	Valid Class	Invalid Class	Expected
Length(l)	1	1=<l<=120	1 0=>l>120	PASS
Data type (td)	2	dt=letters/EsSpCha	2 dt=numbers	FAIL
Bondary value criteria (BVC)				
Bondary	id	length(l)		
Lim+1	1	121	BVC(1)	FAIL
Lim	2	120	BVC(2)	PASS
Lim-1	3	119	BVC(3)	PASS
Lim+1	4	2	BVC(4)	PASS
Min	5	1	BVC(5)	PASS
Min-1	6	0	BVC(6)	FAIL

FIELD: ADDRESS				
Equivalence partitioning criteria (EPC)			Test Cases	
Input	id	Valid Class	Invalid Class	Expected
Length(l)	1	1=<l<=120	1 0=>l>120	PASS
Bondary value criteria (BVC)				
Bondary	id	length(l)		
Lim+1	1	121	BVC(1)	FAIL
Lim	2	120	BVC(2)	PASS
Lim-1	3	119	BVC(3)	PASS
Lim+1	4	2	BVC(4)	PASS
Min	5	1	BVC(5)	PASS
Min-1	6	0	BVC(6)	FAIL

FIELD: SUBDOMAIN				
Equivalence partitioning criteria (EPC)			Test Cases	
Input	id	Valid Class	Invalid Class	Expected
Length(l)	1	1=<l<=20	1 0=>l>20	PASS
Data type (td)	2	dt=letters	2 -Special Character	FAIL
Bondary value criteria (BVC)				
Bondary	id	length(l)		
Lim+1	1	21	BVC(1)	FAIL
Lim	2	20	BVC(2)	PASS
Lim-1	3	19	BVC(3)	PASS
Lim+1	4	2	BVC(4)	PASS
Min	5	1	BVC(5)	PASS
Min-1	6	0	BVC(6)	FAIL

FIELD: CITY NAME				
Equivalence partitioning criteria (EPC)			Test Cases	
Input	id	Valid Class	Invalid Class	Expected
Length(l)	1	1=<l<=30	1 0=>l>30	PASS
Data type (td)	2	dt=letters	2 -Special Character	FAIL
Bondary value criteria (BVC)				
Bondary	id	length(l)		
Lim+1	1	31	BVC(1)	FAIL
Lim	2	30	BVC(2)	PASS
Lim-1	3	29	BVC(3)	PASS
Lim+1	4	2	BVC(4)	PASS
Min	5	1	BVC(5)	PASS
Min-1	6	0	BVC(6)	FAIL

Tabela 33 – Defeitos identificados no *iCardapio*.

iCardapio									
Product/Description/Item	Fault	Fault Type	Time (s)	Description log	Details	Situation	Product/Description/Item	Fault	Fault Type
test01	RCMI	CONCURRENT	24,281	org.junit.CFefailure: expected<...> java.lang.Assert: The field does not	-	FAILURE	test01	RCMI	CONCURRENT
test02	IE	EXCEPTION	44,525	Unable to locate Selenium except	-	ERROR	test02	IE	EXCEPTION
test03	MCE	EXCEPTION	48,905	Unable to locate Selenium except	-	ERROR	test03	MCE	EXCEPTION
test04	RCMI	CONCURRENT	25,658	org.junit.CFefailure: expected<...> java.lang.Assert: The field does not	-	FAILURE	test04	RCMI	CONCURRENT
test05	RCMI	CONCURRENT	20,761	org.junit.CFefailure: expected<...> java.lang.Assert: The field does not	-	FAILURE	test05	RCMI	CONCURRENT
test06	IE	EXCEPTION	22,026	NO LOG	-	FAILURE	test06	IE	EXCEPTION
test07	RCMI	CONCURRENT	20,736	org.junit.CFefailure: expected<...> java.lang.Assert: The field does not	-	FAILURE	test07	RCMI	CONCURRENT
test08	IE	EXCEPTION	16,509	java.lang.Assert: The field does not	-	FAILURE	test08	IE	EXCEPTION
Product/Name/Item									
Test/Case	Fault	Fault Type	Time (s)	Description log	Details	Situation	Test/Case	Fault	Fault Type
test01	NONE	NO FAULT	25,943	NO LOG	-	PASS	test01	NONE	NO FAULT
test02	IE	SEQUENTIAL	16,911	java.lang.Assert: The field does not	-	FAILURE	test02	IE	SEQUENTIAL
test03	IE	SEQUENTIAL	22,333	java.lang.Assert: The field does not	-	FAILURE	test03	IE	SEQUENTIAL
test04	IE	SEQUENTIAL	21,554	java.lang.Assert: The field does not	-	FAILURE	test04	IE	SEQUENTIAL
test05	NONE	NO FAULT	25,927	NO LOG	-	PASS	test05	NONE	NO FAULT
test06	RCMI	CONCURRENT	20,91	org.junit.CFefailure: expected<...> java.lang.Assert: The field does not	-	FAILURE	test06	RCMI	CONCURRENT
test07	RCMI	CONCURRENT	20,699	org.junit.CFefailure: expected<...> java.lang.Assert: The field does not	-	FAILURE	test07	RCMI	CONCURRENT
test08	NONE	NO FAULT	20,452	NO LOG	-	PASS	test08	NONE	NO FAULT
test09	IE	SEQUENTIAL	16,232	java.lang.Assert: The field does not	-	FAILURE	test09	IE	SEQUENTIAL
Product/Photo/Item									
Test/Case	Fault	Fault Type	Time (s)	Description log	Details	Situation	Test/Case	Fault	Fault Type
test01	NONE	NO FAULT	25,644	NO LOG	-	PASS	test01	NONE	NO FAULT
test02	IE	SEQUENTIAL	16,689	java.lang.Assert: The field does not	-	FAILURE	test02	IE	SEQUENTIAL
test03	IE	SEQUENTIAL	22,076	java.lang.Assert: The field does not	-	FAILURE	test03	IE	SEQUENTIAL
test04	MCE	EXCEPTION	47,666	Unable to locate Selenium except	-	FAILURE	test04	MCE	EXCEPTION
test05	IE	SEQUENTIAL	21,324	java.lang.Assert: The field does not	-	FAILURE	test05	IE	SEQUENTIAL
test06	NONE	NO FAULT	25,935	NO LOG	-	PASS	test06	NONE	NO FAULT
test07	NONE	NO FAULT	21,459	NO LOG	-	PASS	test07	NONE	NO FAULT
test08	NONE	NO FAULT	21,04	NO LOG	-	PASS	test08	NONE	NO FAULT
test09	NONE	NO FAULT	21,7	NO LOG	-	PASS	test09	NONE	NO FAULT
test10	IE	SEQUENTIAL	18,78	java.lang.Assert: The field does not	-	FAILURE	test10	IE	SEQUENTIAL
test11	IE	SEQUENTIAL	21,324	java.lang.Assert: The field does not	-	FAILURE	test11	IE	SEQUENTIAL
test12	MCE	EXCEPTION	46,786	Unable to locate Selenium except	-	ERROR	test12	MCE	EXCEPTION
test13	RCMI	CONCURRENT	31,094	org.junit.CFefailure: expected<...> java.lang.Assert: The field does not	-	FAILURE	test13	RCMI	CONCURRENT
test14	NONE	NO FAULT	22,199	NO LOG	-	PASS	test14	NONE	NO FAULT
test15	MCE	EXCEPTION	40,695	Unable to locate Selenium except	-	ERROR	test15	MCE	EXCEPTION
Restaurant/Address/Item									
Test/Case	Fault	Fault Type	Time (s)	Description log	Details	Situation	Test/Case	Fault	Fault Type
test01	NONE	NO FAULT	9,151	NO LOG	-	PASS	test01	NONE	NO FAULT
test02	IE	SEQUENTIAL	8,742	java.lang.Assert: The field does not	-	FAILURE	test02	IE	SEQUENTIAL
test03	IE	SEQUENTIAL	8,665	java.lang.Assert: The field does not	-	FAILURE	test03	IE	SEQUENTIAL
test04	NONE	NO FAULT	8,873	NO LOG	-	PASS	test04	NONE	NO FAULT
test05	NONE	NO FAULT	8,628	NO LOG	-	PASS	test05	NONE	NO FAULT
test06	NONE	NO FAULT	8,44	NO LOG	-	PASS	test06	NONE	NO FAULT
test07	NONE	NO FAULT	8,495	NO LOG	-	PASS	test07	NONE	NO FAULT
test08	IE	SEQUENTIAL	8,405	java.lang.Assert: The field does not	-	FAILURE	test08	IE	SEQUENTIAL
Restaurant/City/Item									
Test/Case	Fault	Fault Type	Time (s)	Description log	Details	Situation	Test/Case	Fault	Fault Type
test01	NONE	NO FAULT	9,035	NO LOG	-	PASS	test01	NONE	NO FAULT
test02	IE	SEQUENTIAL	9,011	java.lang.Assert: The field does not	-	FAILURE	test02	IE	SEQUENTIAL
test03	IE	SEQUENTIAL	8,837	java.lang.Assert: The field does not	-	FAILURE	test03	IE	SEQUENTIAL

test04	IE	SEQUENTIAL	9.197 java.lang.Assert: The field does not	test04	E	SEQUENTIAL	8.983 java.lang.Assert: The field does not	test04	IE	SEQUENTIAL	9.432 java.lang.Assert: The field does not	FAILURE
test05	NONE	NO FAULT	8.692 NO LOG -	test05	NONE	NO FAULT	8.539 NO LOG -	test05	NONE	NO FAULT	8.654 NO LOG -	PASS
test06	NONE	NO FAULT	8.477 NO LOG -	test06	NONE	NO FAULT	8.699 NO LOG -	test06	NONE	NO FAULT	8.694 NO LOG -	PASS
test07	NONE	NO FAULT	8.465 NO LOG -	test07	NONE	NO FAULT	8.491 NO LOG -	test07	NONE	NO FAULT	8.602 NO LOG -	PASS
test08	NONE	NO FAULT	8.659 NO LOG -	test08	NONE	NO FAULT	8.522 NO LOG -	test08	NONE	NO FAULT	8.695 NO LOG -	PASS
test09	IE	SEQUENTIAL	8.426 java.lang.Assert: The field does not	test09	IE	SEQUENTIAL	8.599 java.lang.Assert: The field does not	test09	IE	SEQUENTIAL	8.523 java.lang.Assert: The field does not	FAILURE
RestaurantNameTest												
TestCase	Fault	Fault Type	Time (s)	Description log	Details	Situation	Time (s)	Description log	Details	Fault	Fault Type	Situation
test01	NONE	NO FAULT	8.622 NO LOG -	-	-	PASS	9.073 NO LOG -	-	-	NONE	NO FAULT	PASS
test02	NONE	NO FAULT	8.729 java.lang.Assert: The field does not	-	-	FAILURE	8.92 java.lang.Assert: The field does not	-	-	IE	SEQUENTIAL	FAILURE
test03	IE	SEQUENTIAL	8.539 java.lang.Assert: The field does not	-	-	FAILURE	9.143 java.lang.Assert: The field does not	-	-	IE	SEQUENTIAL	FAILURE
test04	IE	SEQUENTIAL	9.077 java.lang.Assert: The field does not	-	-	FAILURE	8.932 java.lang.Assert: The field does not	-	-	IE	SEQUENTIAL	FAILURE
test05	NONE	NO FAULT	8.693 NO LOG -	-	-	PASS	9.107 NO LOG -	-	-	NONE	NO FAULT	PASS
test06	NONE	NO FAULT	8.832 NO LOG -	-	-	PASS	9.107 NO LOG -	-	-	NONE	NO FAULT	PASS
test07	NONE	NO FAULT	8.597 NO LOG -	-	-	PASS	8.498 NO LOG -	-	-	NONE	NO FAULT	PASS
test08	NONE	NO FAULT	8.419 NO LOG -	-	-	PASS	8.6 NO LOG -	-	-	NONE	NO FAULT	PASS
test09	IE	SEQUENTIAL	8.449 java.lang.Assert: The field does not	-	-	FAILURE	8.472 java.lang.Assert: The field does not	-	-	IE	SEQUENTIAL	FAILURE
RestaurantPhoneTest												
TestCase	Fault	Fault Type	Time (s)	Description log	Details	Situation	Time (s)	Description log	Details	Fault	Fault Type	Situation
test01	NONE	NO FAULT	9.027 NO LOG -	-	-	PASS	8.841 NO LOG -	-	-	NONE	NO FAULT	PASS
test02	NONE	NO FAULT	9.249 NO LOG -	-	-	PASS	9.206 NO LOG -	-	-	NONE	NO FAULT	PASS
test03	NONE	NO FAULT	9.709 NO LOG -	-	-	PASS	8.699 NO LOG -	-	-	NONE	NO FAULT	PASS
test04	NONE	NO FAULT	8.03 NO LOG -	-	-	PASS	9.482 NO LOG -	-	-	NONE	NO FAULT	PASS
test05	NONE	NO FAULT	8.697 NO LOG -	-	-	PASS	9.536 NO LOG -	-	-	NONE	NO FAULT	PASS
test06	NONE	NO FAULT	8.695 NO LOG -	-	-	PASS	9.497 NO LOG -	-	-	NONE	NO FAULT	PASS
test07	NONE	NO FAULT	8.691 NO LOG -	-	-	PASS	9.484 NO LOG -	-	-	NONE	NO FAULT	PASS
test08	NONE	NO FAULT	9.196 NO LOG -	-	-	PASS	9.12 NO LOG -	-	-	NONE	NO FAULT	PASS
test09	NONE	NO FAULT	8.897 NO LOG -	-	-	PASS	8.598 NO LOG -	-	-	NONE	NO FAULT	PASS
test10	NONE	NO FAULT	8.637 NO LOG -	-	-	PASS	9.747 NO LOG -	-	-	NONE	NO FAULT	PASS
RestaurantSloganTest												
TestCase	Fault	Fault Type	Time (s)	Description log	Details	Situation	Time (s)	Description log	Details	Fault	Fault Type	Situation
test01	NONE	NO FAULT	8.669 NO LOG -	-	-	PASS	9.102 NO LOG -	-	-	NONE	NO FAULT	PASS
test02	IE	SEQUENTIAL	9.029 java.lang.Assert: The field does not	-	-	FAILURE	9.06 java.lang.Assert: The field does not	-	-	IE	SEQUENTIAL	FAILURE
test03	IE	SEQUENTIAL	8.674 java.lang.Assert: The field does not	-	-	FAILURE	8.894 java.lang.Assert: The field does not	-	-	IE	SEQUENTIAL	FAILURE
test04	NONE	NO FAULT	8.697 java.lang.Assert: The field does not	-	-	PASS	9.895 java.lang.Assert: The field does not	-	-	NONE	NO FAULT	PASS
test05	NONE	NO FAULT	8.602 NO LOG -	-	-	PASS	9.865 NO LOG -	-	-	NONE	NO FAULT	PASS
test06	NONE	NO FAULT	8.741 NO LOG -	-	-	PASS	9.865 NO LOG -	-	-	NONE	NO FAULT	PASS
test07	NONE	NO FAULT	8.433 NO LOG -	-	-	PASS	9.777 NO LOG -	-	-	NONE	NO FAULT	PASS
test08	NONE	NO FAULT	8.448 NO LOG -	-	-	PASS	9.837 NO LOG -	-	-	NONE	NO FAULT	PASS
test09	IE	SEQUENTIAL	8.622 java.lang.Assert: The field does not	-	-	FAILURE	8.882 java.lang.Assert: The field does not	-	-	IE	SEQUENTIAL	FAILURE
RestaurantSubdomainTest												
TestCase	Fault	Fault Type	Time (s)	Description log	Details	Situation	Time (s)	Description log	Details	Fault	Fault Type	Situation
test01	IE	SEQUENTIAL	7.999 java.lang.Assert: The field does not	-	-	FAILURE	7.614 java.lang.Assert: The field does not	-	-	IE	SEQUENTIAL	FAILURE
test02	NONE	NO FAULT	7.272 NO LOG -	-	-	PASS	7.594 NO LOG -	-	-	NONE	NO FAULT	PASS
test03	NONE	NO FAULT	7.148 NO LOG -	-	-	PASS	7.474 NO LOG -	-	-	NONE	NO FAULT	PASS
test04	NONE	NO FAULT	7.44 NO LOG -	-	-	PASS	7.404 NO LOG -	-	-	NONE	NO FAULT	PASS
test05	IE	SEQUENTIAL	7.228 java.lang.Assert: The field does not	-	-	FAILURE	7.369 java.lang.Assert: The field does not	-	-	IE	SEQUENTIAL	FAILURE
test06	IE	SEQUENTIAL	7.464 java.lang.Assert: The field does not	-	-	FAILURE	7.209 java.lang.Assert: The field does not	-	-	IE	SEQUENTIAL	FAILURE
test07	IE	SEQUENTIAL	7.2 java.lang.Assert: The field does not	-	-	FAILURE	7.09 java.lang.Assert: The field does not	-	-	IE	SEQUENTIAL	FAILURE
test08	IE	SEQUENTIAL	7.34 java.lang.Assert: The field does not	-	-	FAILURE	7.233 java.lang.Assert: The field does not	-	-	IE	SEQUENTIAL	FAILURE
test09	NONE	NO FAULT	7.972 NO LOG -	-	-	PASS	7.422 NO LOG -	-	-	NONE	NO FAULT	PASS