UNIVERSIDADE FEDERAL DE ALFENAS
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

Daniel de Andrade Escobar
Vitor Natariani Muniz

TESTES AUTOMATIZADOS: TEORIA E PRÁTICA

Alfenas/MG
2021

Daniel de Andrade Escobar
Vitor Natariani Muniz

TESTES AUTOMATIZADOS: TEORIA E PRÁTICA

Trabalho apresentado como parte dos requisitos para a disciplina de Trabalho de Conclusão de Curso pelo curso de Ciência da Computação da Universidade Federal de Alfenas - UNIFAL-MG.

Orientador: Rodrigo Martins Pagliares.

Alfenas/MG
2021

Daniel de Andrade Escobar
Vitor Natariani Muniz

TESTES AUTOMATIZADOS: TEORIA E PRÁTICA

A Banca examinadora abaixo-assinada, aprova o Trabalho apresentado como parte dos requisitos para a obtenção do título de Bacharel em Ciência da Computação pela Universidade Federal de Alfenas.

Rodrigo Martins Pagliares
Universidade Federal de Alfenas

Prof. Luiz Eduardo da Silva
Universidade Federal de Alfenas

Fellipe Guilherme Rey de Souza
Instituto Tecnológico de Aeronáutica

Alfenas/MG
2021

RESUMO

A área de automação de testes apresenta constante crescimento nos últimos anos devido à qualidade e praticidade que agrega ao sistema de *software*. A execução de testes de *software* de forma manual ou automatizada visa garantir a qualidade e seu correto funcionamento de acordo com as expectativas descritas nos requisitos. Apesar da evolução da área de testes automatizados de *software* e das vantagens em sua utilização descritas na literatura, é muito comum nos depararmos com projetos de desenvolvimento de *software* que, por diversos motivos, não prescrevem a criação de testes automatizados. Este trabalho tem como objetivos: (i) fazer uma revisão da literatura sobre teste de *software* e (ii) aplicar os conhecimentos adquiridos a partir da revisão da literatura em um *software* exemplo. Inicialmente, este trabalho apresenta uma revisão da literatura sobre conceitos e esquemas de classificação para vários tipos de testes. Na sequência, com base no esquema de classificação conhecido como *Pirâmide de Automação de Testes,* detalha a aplicação de testes automatizados no *software* usado de exemplo, o *HostelApp*, que tem como principal funcionalidade gerenciar as reservas de um albergue, o que inclui as funcionalidades de gerenciar os hóspedes cadastrados e os quartos utilizados para acomodação. O conjunto de testes automatizados desenvolvido neste trabalho, aplicado ao *software* desenvolvido como exemplo, facilita e protege as atividades de evolução, manutenção e melhoria da qualidade do código do *software* de exemplo. Caso *bugs* (defeitos e/ou falhas encontradas no *software*) sejam introduzidos em código coberto por testes automatizados durante essas atividades, eles são identificados imediatamente, facilitando a evolução do exemplo de acordo com os requisitos do projeto. A utilização de testes automatizados beneficia desenvolvedores, testadores, e usuários finais. Desenvolvedores e testadores despendem menor esforço na execução de testes e ficam mais seguros ao realizar alterações no *software*. Usuários obtêm *software* de melhor qualidade. O desenvolvimento do HostelApp junto, da automação de testes, nos fez concluir que a automação de testes traz diversos benefícios no processo de desenvolvimento de software, como a melhoria na qualidade do código e na confiabilidade e facilita a manutenção.

Palavras-chave: Teste de *software,* testes automatizados, evolução de *software*, manutenção de *software*.

ABSTRACT

The testing automation area has shown constant growth in recent years due to the quality and practicality that it adds to the software system. The execution of software tests in a manual or automated way aims to guarantee the software quality and its correct functioning according to the expectations described in the requirements. Despite the evolution of the area of automated software testing and the benefits of its utilization described in the literature, it is very common to come across software development projects that, for many reasons, don't prescribe the creation of automated tests. This work aims to: (i) perform a literature review about software testing and (ii) apply the knowledge acquired from the literature review into a sample software. At first, this work provides a review of the literature on concepts and classification schemes for the various types of tests. In the sequence, based on the classification scheme known as *Tests Automation Pyramid*, it details the application of automated testing into the software used as an example, the *HostelApp*, whose main functionality is to manage the reservations of a hostel, which includes the functionalities to manage registered guests and rooms used for accommodations. The set of automated tests developed in this work, applied in the sample software developed, facilitates, and protects the activities of evolution, maintenance, and improvement of its code quality. If bugs are introduced in the code covered by the automated tests during those activities, they will be identified immediately, facilitating the evolution of the example according to the project requirements. The utilization of automated tests benefits the developer, testers, and final users. Developers and testers spend less effort in the execution of the tests and are safer to make changes to the software. Users get software with more quality. The development of *HostelApp* made us conclude that the tests automation brings several benefits in software development, such as improved code quality and reliability and improved maintenance.

Keywords: Software testing, automated tests, software evolution, software maintenance.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE CÓDIGOS

SUMÁRIO

1 INTRODUÇÃO

A área de automação de testes apresenta constante crescimento nos últimos anos devido à qualidade e praticidade que agrega ao sistema de *software* (LUO, 2001). Apesar da evolução da área de testes de *software* automatizados e das vantagens em sua utilização descritas na literatura, é muito comum nos depararmos com projetos de desenvolvimento de *software* que, por diversos motivos (RAFI *et al.*, 2012), não prescrevem a criação de testes automatizados.

Dentre esses motivos, podemos citar recursos e prazos de entrega limitados — devido à concorrência para entregar o *software* no menor tempo possível, as empresas acabam ignorando o aspecto de qualidade visando o *curto prazo* —, o estado em que o *software* se encontra — quando o *software* principal de uma empresa é construído utilizando códigos e ferramentas antigas, torna-se menos custoso deixar esse código considerado estável da maneira em que está e apenas adicionar novas funcionalidades ao produto —, entre outros motivos.

A criação e execução de testes, realizadas de forma manual ou automatizada, juntamente de outras atividades complementares como revisão e verificação de código, têm como objetivo garantir a qualidade e o correto funcionamento do Sistema Sob Teste (ou SUT - System Under Test) de acordo com as expectativas descritas nos requisitos do projeto (KACZANOWSKI, 2013). A atividade de teste de *software* minimiza a ocorrência de erros e riscos associados ao processo de desenvolvimento de *software*, além de fornecer evidências quanto à confiabilidade do *software* sendo desenvolvido.

O domínio de entrada de um SUT e todas suas combinações podem ser muito amplas, de modo a tornar o tempo da atividade de teste inviável, sendo assim impossível testar tudo (MALDONADO *et al.*, 2007). Com isso, é papel do desenvolvedor do *software* identificar quais são as funcionalidades principais do projeto, para que quando houver mudanças no código, ao executar os testes seja possível identificar se uma funcionalidade foi prejudicada com aquela modificação, para que assim seja consertada o quanto antes.

Além de identificar quais são as principais funcionalidades do projeto que devem ser testadas, é vantajoso ao desenvolvedor do *software* automatizar os testes, implementando combinações e comandos de operações (casos de teste) que validam diversos cenários de uma só vez. No contexto deste trabalho, um caso de teste é uma condição particular a ser testada, composta por valores de entrada, restrições para a execução e um resultado esperado.

Os testes automatizados ajudam a solucionar os problemas encontrados nos testes manuais, diminuindo a quantidade de erros e aumentando a qualidade do *software*. Como é relativamente fácil executar um conjunto (*suíte*) de testes automatizados a qualquer momento,

mudanças no SUT podem ser feitas com segurança, pois basta o desenvolvedor executar a *suíte* de testes ao final de cada mudança que fizer no código para se assegurar que novos *bugs* não foram introduzidos no SUT com a mudança, o que aumenta a vida útil do produto.

Esta monografia apresenta uma revisão de literatura sobre testes de *software*. Com base no resultado da revisão da literatura, delimita-se o escopo do trabalho para um exemplo prático de uso de testes automatizados criados a partir de um esquema de classificação conhecido como *Pirâmide de Automação de Testes* (COHN, 2010). Para suportar o exemplo prático de aplicação de testes automatizados, um *software* desenvolvido pelos próprios autores, o *HostelApp*, que gerencia o sistema de reservas de um albergue, é apresentado. Para o *HostelApp*, são implementados *testes de unidade, integração* e *aceitação (fim-a-fim)* com objetivo de verificar a correta implementação do código.

O restante desta monografia está organizado da seguinte maneira: no Capítulo 2 apresentamos os trabalhos relacionados. O Capítulo 3 apresenta uma revisão da literatura sobre teste de *software*. O Capítulo 4 descreve um exemplo de um sistema *web* para criação de reservas em um albergue chamado *HostelApp*. Implementamos o exemplo em *software* para ser usado como SUT na demonstração do uso de testes automatizados. No Capítulo 5 é descrito o plano de testes com casos de testes para os principais cenários de uso associados à gestão de reservas no *HostelApp*. No Capítulo 6 são apresentados os códigos utilizados para testar os casos de teste descritos no Capítulo 5. O Capítulo 7 apresenta algumas discussões sobre o desenvolvimento do trabalho. Conclusões e trabalhos futuros são apresentados no Capítulo 8.

2 TRABALHOS RELACIONADOS

Este capítulo apresenta trabalhos relacionados ao conteúdo descrito nesta monografia. Em especial, são descritos trabalhos associados à importância de testes para verificação e validação de *software*, bem como as vantagens de se utilizar testes automatizados.

Maldonado, Delamaro e Jino (2007) relatam que no processo de desenvolvimento de *software* todos os erros são erros humanos e, mesmo utilizando-se dos melhores métodos de desenvolvimento e práticas, erros permanecem presentes nos diversos produtos de *software*. Neste contexto, dentre as técnicas de verificação e validação, a atividade de teste é uma das mais utilizadas, constituindo-se em um dos elementos para fornecer evidências da confiabilidade do *software*.

Myers *et al.* (2004) afirmam que o *software* deve ser previsível e consistente, não havendo nenhum tipo de surpresa por parte dos usuários com relação a seu comportamento. Em razão disso, é responsabilidade do teste de *software* determinar a correta funcionalidade do *software* no ambiente para o qual foi projetado. Por meio destes testes, os desenvolvedores colocam à prova o que foi produzido, assegurando a confiabilidade e a qualidade do produto.

Crespo *et al.* (2004) descreve que a atividade de teste de *software* é composta por alguns elementos essenciais que auxiliam na padronização desta atividade, dentre eles: os casos de teste), o procedimento de teste (um passo ou grupo de casos necessários para executar um único caso de teste) e os critérios de teste (condições para selecionar e avaliar os casos de testes de forma a aumentar a cobertura de correção do produto.

Segundo Graham e Fewster (2012) a execução manual de um caso de teste é rápida e efetiva, mas a execução e repetição manual de um vasto conjunto (bateria) de testes é uma tarefa muito dispendiosa e cansativa. É necessário um certo esforço para executar todo o conjunto de testes manuais, pois a cada surgimento e correção de um erro a bateria de testes é executada novamente. Este cenário tem como consequência o efeito "bola de neve": acúmulos de erros de regressão (erros em módulos do sistema que estavam funcionando corretamente e deixam de funcionar após alterações e inclusões de novas funcionalidades).

Testes automatizados são programas ou *scripts* simples que exercitam funcionalidades do SUT e fazem verificações automáticas nos efeitos colaterais obtidos. A grande vantagem desta abordagem é que todos os casos de teste podem ser fácil e rapidamente repetidos, a qualquer momento e com pouco esforço, evitando o efeito "bola de neve".

Ainda sobre testes automatizados, Silva e Siqueira (2013) aplicam os conceitos de SUT e DOC (*Depended On Component*) (MESZAROS, 2007), em um exemplo de um sistema PDV

(Ponto de Venda) descrito por Larman (LARMAN, 2004). Os autores também descrevem ferramentas de teste de *software* como o *framework* para testes de unidade JUnit (JUNIT, 1998) e concluem que sistemas com testes automatizados se tornam mais flexíveis, confiáveis e com maior segurança para alterações, pois com os testes, verifica-se de forma rápida se uma alteração realizada gera algum *bug* ou instabilidade no SUT.

Por último, mas não menos importante, Kumar e Mishra (KUMAR e MISHRA, 2016) analisam o custo-benefício da automação de testes, os impactos na qualidade geral e no tempo de lançamento do *software*. A análise é feita por meio de experimentos em três ferramentas de *software*, usando um modelo matemático com base nos fatores de esforço de teste para quantificar o impacto da automação de testes. Os resultados da comparação confirmam os efeitos positivos da automação de testes nas métricas de custo, qualidade e tempo de entrega do produto de *software* no mercado.

3 TESTE DE SOFTWARE: UMA REVISÃO DA LITERATURA

Este capítulo apresenta uma revisão da literatura sobre teste de *software*. O objetivo é fornecer uma visão teórica ampla sobre diversos aspectos relacionados a testes de *software* presente na literatura, para então delimitarmos o escopo do trabalho. O capítulo discute termos importantes da área de teste de *software*, abordagens, técnicas e tipos de testes, além de esquemas de classificação de testes normalmente utilizados em métodos ágeis.

3.1 INTRODUÇÃO A TESTE DE SOFTWARE

Nos dias atuais, a prática de testar *software* faz parte do dia a dia de desenvolvedores e testadores envolvidos em projetos de desenvolvimento de *software*. Contudo, os testes de *software* começaram a se tornar uma área de estudo da engenharia de *software* somente na década de 1970 (BERNARDO, 2011). Teste de *software* é um termo amplo que engloba desde a definição de um processo de teste no intuito de permitir que desenvolvedores investiguem código-fonte para detectar *bugs* de programação, até a validação do *software* por um usuário final (CRESPO *et al.*, 2004).

Os *erros* de programação cometidos por um desenvolvedor no código fonte do *software* são conhecidos como *bugs*. *Bugs*, em certas condições, produzem resultados incorretos, conhecidos como *falhas*. Entretanto, nem todos os *bugs* resultam necessariamente em *falhas*. Por exemplo, um trecho de código fonte com *bug*, mas que não é invocado durante a execução do *software*, não resulta em *falhas*. Além disso, nem todos os *bugs* são causados por erros de programação. Uma causa comum para o surgimento de *bugs* é a não identificação de requisitos necessários para o correto funcionamento do *software* (KOLAWA e HUIZINGA, 2007).

Testes objetivam encontrar *bugs* no *software*. Um *bug* encontrado e corrigido pode expor algum outro *bug* existente ou desencadear o surgimento de outros *bugs*. O SUT pode se apresentar em diversas granularidades; desde uma única classe a um *software* completo.

3.1.1 Impossibilidade de testar tudo

Testar *software* em sua completude, sob todas as combinações de entradas e pré-condições, é impossível, mesmo para *software* simples (KANER et. al., 1999). Por isso é essencial definir estratégias de seleção e priorização de testes alinhadas com os recursos humanos, tecnológicos e de tempo disponíveis.

Embora não seja possível testar o correto funcionamento do *software* mediante *todas* as combinações de entrada e pré condições necessárias para a execução de um teste, os testes podem garantir que o *software não* funciona corretamente sob *específicas* condições (KANER et. al., 1999). Estabelecer estas condições específicas demanda o esforço de testadores de *software*.

3.1.2 Papéis relacionados a teste de *software*

Hoje em dia, existe uma grande comunidade de profissionais especializados, conhecidos como testadores de *software*. Parte desses profissionais trabalha em regime de dedicação exclusiva desempenhando o papel de testadores, enquanto outra parte executa atividades de testes em conjunto com outras atividades de desenvolvimento de *software*. Por exemplo, testes de unidade são implementados por desenvolvedores desempenhando o papel de testadores.

Além do termo genérico *testador de software*, a indústria de *software* define diversos outros papéis com habilidades específicas de testes, como por exemplo *analista de teste*, *gerente de teste*, *líder de teste*, *designer de teste*, *testador*, *administrador de teste,* dentre outros (GREGORY e CRISPIN, 2014).

3.1.3 Quando e com que frequência testar *software*?

Normalmente o modelo de ciclo de vida de um processo ou método de desenvolvimento de *software* determina quando e como os testes são realizados (GREGORY e CRISPIN, 2014). Por exemplo, em um processo com modelo de ciclo de vida em cascata, a maioria dos testes ocorre no final do ciclo de desenvolvimento do *software* (COLLINS *et al.,* 2011). Nesse caso, algumas empresas dispõem de um setor de Garantia da Qualidade (*Quality Assurance* - QA), com *testadores* dedicados a testes dos sistemas de *software* entregues pelos desenvolvedores.

Já em métodos de desenvolvimento de *software* como modelo de ciclo de vida iterativo e incremental, tais como os métodos ágeis, os testes são projetados e executados durante todo o ciclo de vida de desenvolvimento, simultaneamente com a definição de requisitos, análise, *design* e programação. Em outras palavras, em métodos ágeis, questões de QA são diluídas por todo o ciclo de vida de desenvolvimento do *software* e, não apenas no fim do desenvolvimento como em métodos mais tradicionais.

Além de testar durante todo ciclo de vida de desenvolvimento do *software*, é comum entre praticantes de métodos ágeis o uso de testes como direcionadores do processo de desenvolvimento de *software*, em uma estratégia conhecida como *test-first*. Na estratégia *test-first*, os testes são escritos antes da criação de código fonte necessário para que o teste seja executado. Entre profissionais de desenvolvimento de *software* que utilizam a estratégia *test-first*, é comum o uso dos termos TDD (Desenvolvimento Dirigido por Testes, do inglês, *Test Driven Development*), ATDD (Desenvolvimento Dirigido por Testes de Aceitação, do inglês, *Acceptance Test Driven Development*) e BDD (Desenvolvimento Dirigido por Comportamento, do inglês, *Behavior Driven Development*).

TDD é uma prática de desenvolvimento de *software* (BECK, 2003), baseada no conceito de testes automatizados. Com TDD, requisitos de *software* são transformados por *desenvolvedores* (não por *testadores* dedicados) em *casos de teste* antes do código que implementa os requisitos. TDD é uma prática oposta à prática tradicional de desenvolvimento na qual *software* é desenvolvido antes dos *casos de teste* serem criados e executados.

Tendo em vista que os *casos de testes* são criados antes do código que os atendam, logo após criados, a execução de um *caso de teste* falha. A partir deste ponto, *desenvolvedores* implementam código no intuito de satisfazer o *caso de teste* falho.

No TDD, este ciclo de definição, falha na execução e implementação de código para sucesso na execução do *caso de teste*, é feito repetidamente e em intervalo curto de tempo (da ordem de poucos minutos). Em cada repetição do ciclo, os *desenvolvedores* realizam atividades de refatoração (*refactoring*) para melhoria do *design* do código sendo produzido.

ATDD é uma prática de desenvolvimento de *software* que enfatiza a comunicação entre o cliente, desenvolvedor e testador para assegurar que os requisitos estejam bem definidos e sejam testáveis (PUGH, 2010). Testes escritos usando ATDD devem ser escritos na linguagem do cliente e, diferentemente de TDD, não requerem automação dos testes.

BDD é uma prática de desenvolvimento de *software* que combina as práticas de ATDD e TDD (PINHEIRO, 2015). BDD é uma prática *test-first* que foca em testes que descrevem comportamento em vez de testes que que testam uma unidade isolada de implementação. Testes

escritos com BDD usam uma sintaxe que permite que clientes, desenvolvedores e testadores definam juntos os comportamentos a serem escritos como testes automatizados.

3.1.4 Testes manuais *versus* testes automatizados

Testes de *software* podem ser realizados manualmente ou implementados pelos *desenvolvedores* e *testadores* para serem realizados de forma automatizada. Os testes manuais, como o próprio nome já diz, são realizados manualmente por seres humanos de acordo com casos de testes. Desenvolvedores, testadores, e até mesmo usuários finais, podem realizar testes manuais seguindo os casos de teste que descrevem o passo a passo para se obter o resultado esperado. Com testes manuais, não há auxílio de nenhuma ferramenta de *software* ou mesmo um *script* automatizado para execução dos testes.

Já os testes automatizados podem tornar os testes de *software* independentes de intervenção humana, onde as limitações dos testes manuais podem ser superadas utilizando-se de ferramentas de *software* e *scripts* automatizados para a execução dos testes.

3.2 ABORDAGENS DE TESTES

Existem diversas *abordagens* para teste de *software*. Algumas delas estão em uso desde a década de 70. O Quadro 3.1 apresenta um resumo das principais *abordagens* de teste de *software* encontradas na literatura. As *abordagens* descritas no quadro são detalhadas nas subseções seguintes.

Quadro 3.1 - Abordagens de teste de *software*.

Abordagem	**Definição**
Estática	Abordagem na qual o SUT é testado sem execução de código.
Dinâmica	Abordagem que prescreve a execução do SUT com dado conjunto de *casos de testes*.
Passiva	Abordagem que verifica o comportamento do SUT sem nenhuma interação com ele.
Exploratória	Abordagem que prescreve simultaneamente *design* (projeto) e execução de testes para aprender sobre o SUT e se beneficiar das lições aprendidas quando das próximas sessões de testes exploratórios.

Caixa preta	Abordagem que baseia-se em requisitos e especificações sem a necessidade de conhecimento dos caminhos internos do *software* tais como desvios condicionais, ou a estrutura interna do código fonte do SUT.
Caixa branca	Abordagem em que os testes são baseados na estrutura interna do código fonte do SUT.
Caixa cinza	Abordagem intermediária entre *caixa preta* e caixa *branca* em que o *testador* tem acesso suficiente ao código para entender como o SUT foi implementado.

3.2.1 Abordagens estática, dinâmica e passiva

As *abordagens* de testes estática, dinâmica e passiva lidam respectivamente com testes sem que SUT esteja em execução, com o SUT em execução, e com testes que não interagem com o SUT.

Na *abordagem estática de teste*, o SUT é testado sem execução de código. Revisões, demonstrações *(walkthroughs*), e inspeções são exemplos de testes de abordagem estática (RUNESON e WOHLIN, 1995). A execução do SUT com dado conjunto de *casos de testes* é uma abordagem de testes conhecida como *abordagem dinâmica* ou *abordagem de teste dinâmico* (GRAHAM, D. et. al., 2008) (OBERKAMPF, W. L, ROY, C. J., 2010). A abordagem de *teste dinâmico* ocorre quando o próprio SUT é executado. A abordagem de *teste passiva* verifica o comportamento do SUT sem nenhuma interação com ele. Os *testadores* não fornecem dados de teste, mas examinam os *logs* e acompanham a execução do do SUT (LEE, D. et. al, 1997).

3.2.2 Abordagem exploratória

A *abordagem de teste exploratório* (ou *abordagem exploratória de testes*) prescreve ciclos rápidos de aprendizado, *design*, e execução de testes ao longo do processo de desenvolvimento de *software* no intuito de descobrir comportamentos no SUT não cobertos por *scripts* de testes. As lições aprendidas em cada ciclo são usadas como experiência para o próximo ciclo de aprendizado, *design* e execução de testes (HENDRICKSON, 2013). A abordagem enfatiza a liberdade e a responsabilidade de cada *testador* de otimizar continuamente a qualidade de seu trabalho.

A abordagem exploratória de testes é um processo informal quando comparado com *casos de testes* executados a partir de *scripts*, embora requeira disciplina por parte do *testador*

para que seja desempenhada de forma eficaz. Uma forma comum de uso da *abordagem de teste exploratório* é no contexto de sessões delimitadas no tempo (*time box sessions*). Estas sessões focam em um aspecto particular do SUT a ser explorado.

3.2.3 Abordagem de caixas (caixa branca, caixa preta e caixa cinza)

Além das *abordagens de testes de software* discutidas até o momento, algumas outras *abordagens*, conhecidas como *abordagens de caixa*, são amplamente discutidas na literatura. As *abordagens de caixa* são divididas em três categorias; *abordagem de teste de caixa branca*, *abordagem de teste de caixa preta* e *abordagem de teste de caixa cinza*. (COPELAND, L., 2003) (CRESPO *et al.*, 2004).

A abordagem de *teste de caixa preta* baseia-se em requisitos e especificações sem a necessidade de conhecimento dos caminhos internos do *software* tais como desvios condicionais, ou a estrutura interna do código fonte sendo testado. A abordagem não requer conhecimento de programação por parte do testador.

O Quadro 3.2 apresenta algumas *técnicas* comuns usadas na *abordagem de testes de caixa preta*. Copeland (COPELAND, L., 2003) apresenta detalhes de uso de cada uma das *técnicas* apresentadas no Quadro 3.2.

Quadro 3.2 - Técnicas utilizadas na *abordagem de testes de caixa preta*.

Técnica	**Definição**
Teste de classe de equivalência	Usada para reduzir o número de casos de testes a um nível gerenciável mantendo, ao mesmo tempo, uma cobertura de testes razoável.
Teste de valor de fronteira	Auxilia testadores a escolherem um pequeno subconjunto dos casos de testes possíveis, mantendo, ao mesmo tempo, uma cobertura de testes razoável.
Teste com tabela de decisão	Tabela de decisão é uma ferramenta para captura de certos tipos de requisitos e para documentação do *design* interno do *software*. Tabelas de decisão são usadas para registrar regras complexas de negócio baseadas em um conjunto de condições que um *software* deve implementar. Além disso, as tabelas de decisão podem orientar a criação de casos de testes.
Teste par a par (*pairwise*)	Técnica em que casos de teste são projetados para executar todas as combinações discretas de cada par de parâmetros de entrada. Também conhecida como teste de todos os pares (*all-pairs testing*),

Teste de transição de estados	Técnica em que o testador examina o comportamento do SUT mediante várias condições de entrada fornecidas em sequência. Esta técnica é usada quando o SUT é definido em termos de um número finito de estados e as transições entre os estados são determinadas por regras do SUT. Testes de transição de estados são projetados para executar transições válidas e inválidas entre os estados do SUT.
Teste de análise de domínio	Técnica que pode ser usada para identificar casos de testes eficientes e efetivos quando várias variáveis (tais como campos de entrada de texto em um formulário) devem ser testadas em conjunto por questões de eficiência ou interação lógica. Trata-se de uma generalização das técnicas de *classe de equivalência* e *valor de fronteira* com o objetivo de se encontrar situações que os valores de fronteira foram definidos ou implementados incorretamente.
Teste de caso de uso	Técnica que permite identificar casos de testes que cobrem os cenários de um caso de uso. Um cenário de caso de uso descreve um uso particular do SUT por um usuário. Na técnica, os casos de teste são as interações entre o usuário e o SUT formuladas a partir de cenários (sucesso e insucesso) presentes no caso de uso.

Teste de caixa branca é uma *abordagem* em que o teste é baseado nos caminhos internos, na estrutura e na implementação do SUT. O *teste de caixa branca* requer habilidade de programação. Durante o projeto de *casos de testes de caixa branca*, o testador escolhe entradas para exercitar caminhos através do SUT e determinar as saídas apropriadas.

O Quadro 3.3 apresenta as duas principais *técnicas* usadas na *abordagem de testes de caixa branca*. Consulte Copeland (COPELAND, L., 2003) para maiores detalhes de uso de cada uma das técnicas apresentadas no Quadro 3.3.

Quadro 3.3 - Técnicas utilizadas na *abordagem de testes de caixa branca*.

Técnica	**Definição**
Teste de controle de fluxo	Técnica que identifica os caminhos de execução no código do SUT para criação e execução de *casos de teste* que cubram os caminhos identificados.
Teste de fluxo de dados	Técnica usada para detecção de uso impróprio de valores de dados devido a erros de programação. Um erro comum de programação detectado pela técnica, por exemplo, é referenciar o valor de uma variável sem antes atribuir um valor a ela.

Teste de caixa cinza é uma *abordagem de testes* intermediária entre *caixa preta* e *caixa branca* em que o *testador* com conhecimento de programação tem acesso suficiente ao código-fonte para entender como o SUT foi implementado. A partir deste entendimento, o *testador* considera o SUT como uma caixa preta e usa o conhecimento adquirido e estratégias para a criação de testes mais efetivos.

3.3 TIPOS DE TESTES DE *SOFTWARE*

Na área de desenvolvimento de *software*, é comum classificarmos requisitos como *Requisitos Funcionais* - RFs - e *Requisitos Não Funcionais* - RNFs (SOMMERVILLE, 2016). RFs descrevem o que o *software* deve fazer para atender os objetivos de negócios de seus usuários. Os RNFs descrevem como o *software* deve ser implementado para atender às expectativas dos usuários com relação a sua qualidade. A distinção de requisitos de *software* em RFs e RNFs é usada para caracterizar diversos tipos de testes de *software* em dois grandes grupos: *testes funcionais* (associados com RFs) e *testes não funcionais* (associados com RNFs).

3.3.1 Testes funcionais

Teste funcional é um tipo de teste que verifica se as funcionalidades do SUT operam em conformidade com a especificação de RFs. *Testes funcionais* normalmente estão associados à *abordagem de teste de caixa preta*, embora também possam ser executados via *abordagens de teste de caixas branca e cinza*.

Testes funcionais podem ser executados de forma manual ou automatizada. Cada funcionalidade do SUT é testada fornecendo entrada adequada, verificando a saída e comparando os resultados produzidos com os resultados esperados. Caso o SUT seja constituído de múltiplas camadas lógicas (FOWLER, 2003), durante o teste de uma funcionalidade, todas camadas são verificadas; da *camada de apresentação*, passando pela lógica de negócios na *camada de serviços* e chegando até o banco de dados na *camada de recursos* a partir da *camada de integração*.

O Quadro 3.4 apresenta exemplos de *tipos de testes funcionais* encontrados na literatura com suas respectivas descrições. Maiores detalhes sobre cada um deles são apresentados na sequência.

Quadro 3.4 - Testes Funcionais.

Tipo de Teste	**Descrição**	**Referências**
Unidade	Teste escrito e executado com frequência por um desenvolvedor sobre trechos isolados do SUT com objetivo de assegurar que os trechos individuais estão corretos. Cada *caso de teste de unidade* deve ser testado independentemente de outros *casos de teste*.	(KACZANOWSKI, 2013) (HOODA, 2015) (DESIKAN, 2005)
Componente	Teste que verifica o correto funcionamento de *componentes de software* de forma isolada, sem integração com outros *componentes*.	(WHITTAKER, 2000) (IEEE, 1990) (MALDONADO et al., 2007)
Integração	Teste escrito e executado por um *desenvolvedor* objetivando a integração adequada das diferentes partes do SUT, incluindo partes com código desenvolvido por terceiros.	(KACZANOWSKI, 2013) (HOODA, 2015) (DESIKAN, 2005)
Fim-a-Fim	Teste escrito e executado, normalmente por profissionais de testes de um setor de garantia de qualidade, objetivando verificar se o fluxo de execução do SUT, do início ao o fim, está se comportando como esperado.	(KACZANOWSKI, 2013) (HOODA, 2015)
Aceitação	Teste executado pelo *usuário final* para verificar, antes de sua migração para um ambiente de produção, se o *software* sendo entregue atende aos requisitos definidos no projeto de desenvolvimento do *software*.	(HOODA, 2015) (IEEE, 1990) (DESIKAN, 2005)
Fumaça	Teste preliminar que objetiva revelar falhas simples, porém severas o suficiente para, por exemplo, rejeitar a liberação do *software* para o ambiente de produção.	(GUPTA e SAXENA, 2013) (DESIKAN, 2005)
Sanidade	Teste superficial, executado rapidamente, no intuito de verificar se determinada funcionalidade do SUT está funcionando como esperada e se é razoável proceder testando todo o SUT.	(GUPTA e SAXENA, 2013) (DESIKAN, 2005)
Regressão	Teste que prescreve um processo para testar o SUT a fim de garantir que o mesmo esteja funcionando mesmo após alterações.	(IEEE, 1990) (GUPTA e SAXENA, 2013) (DESIKAN, 2005)
Sistema	Teste conduzido em um *sistema* completo e integrado com o objetivo de avaliar a aderência do *sistema* com os requisitos funcionais especificados.	(IEEE, 1990) (DESIKAN, 2005)

Alfa	Trata-se de uma forma de teste de aceitação do *software* realizada por desenvolvedores e testadores que fornecem *feedback* sobre a atual situação do sistema, antes da entrega para os usuários finais.	(MOHD e SHAHBODIN, 2015) (DESIKAN, 2005)
Beta	Trata-se de uma forma de teste de aceitação do *software* realizada com os clientes, com objetivo de testar a recepção do produto no mercado, descobrir características inadequadas ou possíveis melhorias antes do lançamento para o público final.	(MOHD e SHAHBODIN, 2015) (DESIKAN, 2005)
Gama	Trata-se de uma forma de teste de aceitação do *software* executada no estágio final de um processo de testes, antes da liberação do *software* para o mercado.	(MOHD e SHAHBODIN, 2015) (DESIKAN, 2005)

Testes de unidade (Kaczanowski, 2013) são testes tipicamente escritos e executados com frequência por desenvolvedores. O objetivo de testes de unidade é isolar cada trecho do *software* e assegurar que os trechos individuais estão corretos. Cada caso de teste deve ser testado independentemente. Para isolar trechos de *software*, é comum o uso de objetos substitutos tais como *dummies*, *stubs*, *spies*, objetos *mock*, e objetos falsos (*fake objects*). *Frameworks de testes automatizados*, também conhecidos como *test harnesses*, podem ser usados para auxiliar o teste de um módulo de forma isolada. Os *frameworks de testes automatizados* consistem de vários componentes de *software* e dados de testes, configurados para o teste e monitoramento do comportamento de uma unidade do SUT e suas saídas produzidas sob diversas condições.

De maneira geral, *software* como um todo é constituído de vários componentes. O *teste de componentes* (também conhecido como teste de módulo, quando associado a uma visão arquitetural do *software*) verifica o correto funcionamento de componentes de *software* de forma isolada, sem integração com outros componentes. Trata-se de um dos tipos de testes de *abordagem de caixa preta* mais comuns executados por testadores, normalmente após os desenvolvedores terem implementado *testes de unidade*.

Os *testes de integração* focam na integração adequada dos diferentes módulos do SUT, incluindo códigos desenvolvidos por terceiros, que a princípio, não temos controle. Um exemplo é a integração entre classes de negócio e classes de um *framework* objeto-relacional ou classes de um *framework* para desenvolvimento de serviços *web*. Embora *testes de integração* cubram uma área muito mais ampla de código do SUT quando comparado com *testes de unidade*, eles ainda testam código do ponto de vista de um desenvolvedor.

Os *testes de integração* são muito mais lentos de serem executados do que *testes de unidade*. Eles normalmente requerem recursos (por exemplo, o contexto de uma aplicação) a serem configurados antes que os testes possam ser executados e a execução dos *testes de integração* envolve invocar entidades que tendem a responder de forma lenta (por exemplo, banco de dados, sistemas de arquivos, e serviços *web*). Para verificarmos os resultados de testes de integração, é normalmente necessário analisar recursos externos (por exemplo, realizando uma consulta SQL (*Structured Query Language*))

Os *Testes fim-a-fim* existem para verificar que o código funciona a partir da perspectiva do usuário final, imitando a forma como o usuário usa as diversas funcionalidades fornecidas pelo SUT. Dessa forma, *testes fim-a-fim* exercitam todas as camadas lógicas do SUT. Diferentemente de *teste de unidade*, objetos substitutos raramente são usados em *testes fim-a-fim*, já que o objetivo é testar o SUT real. Os *testes fim-a-fim* normalmente requerem bastante tempo para serem executados.

Teste de aceitação do usuário é um tipo de teste executado pelo usuário final ou cliente para verificar/aceitar o *software* sendo entregue antes da migração do *software* para um ambiente de produção. *Testes de aceitação* são normalmente conduzidos nas fases finais de um processo de testes.

Testes de fumaça são testes preliminares que objetivam revelar falhas simples, porém severas o suficiente para, por exemplo, rejeitar a liberação do produto de *software* para o ambiente de produção. Os *testes de fumaça* constituem um subconjunto dos *casos de teste* que cobrem a(s) funcionalidade(s) mais importantes de um componente ou sistema, usado para auxiliar na avaliação se as funcionalidades principais do *software* estão funcionando corretamente.

Testes de sanidade (*sanity tests)* (DESIKAN, 2005) são testes básicos e superficiais, executados rapidamente, no intuito de verificar se determinada funcionalidade do SUT esteja funcionando como esperado e se é possível e razoável proceder testando todo o SUT. Normalmente os *testes de sanidade* são executados antes de uma rodada de testes mais completos que analisam as funcionalidades do SUT de forma exaustiva.

Testes de fumaça e de *sanidade* são muitas vezes usados como sinônimos, em especial pelo fato de serem testes preliminares que evitam desperdício de tempo e esforço no desenvolvimento de *software*. Quando, por exemplo, o SUT ainda possui diversos *bugs*, torna-se mais viável a participação de desenvolvedores em sessões de depuração do código evitando desperdício de recursos com uma investigação mais rigorosa com testes.

Teste de regressão é um tipo de teste baseado na reexecução de testes após alterações no código do SUT para assegurar que os requisitos continuam sendo atendidos. Diversas alterações podem demandar a execução de *testes de regressão*, tais como melhorias no *software*, correções de *bugs*, mudanças de configuração, até mesmo a substituição de componentes eletrônicos (NATIONAL RESEARCH COUNCIL, 2001). O número de *casos de testes* incluídos em *suítes* (conjuntos) de *testes de regressão* tende a crescer na medida em que cada *bug* é encontrado, o que faz com que a automação seja uma atividade essencial para a realização de *testes de regressão*.

Teste de sistema (IEEE, 1990) testa um sistema completo e integrado de forma a verificar o atendimento aos requisitos. Trata-se de teste que verifica todas as camadas de um SUT, da interface gráfica com o usuário até o mecanismo de persistência de dados, atestando o correto funcionamento dos componentes presentes nas camadas e a correta integração entre eles. São similares a *testes de aceitação do usuário*, sendo que uma diferença é que normalmente usuários finais não participam de sessões de *testes de sistema*.

Testes alfa, *beta* e *gama* são testes normalmente conduzidos para garantir que *software* de alta qualidade seja entregue para os clientes. Os *testes alfa* são realizados por testadores e desenvolvedores internos ao time de desenvolvimento para verificar a aceitação do *software*, além de detectar possíveis *bugs* antes do lançamento para os usuários. Os *testes beta* são executados após os *testes alfa* e são considerados *testes de aceitação externa* (com a inclusão de usuários finais). Versões do *software* (conhecidas como versões *beta*) são lançadas para um determinado público-alvo sem participação de desenvolvedores no intuito de se verificar a corretude da implementação. Já os *testes gama* são executados no estágio final de um processo de testes, antes da liberação do *software* para o mercado. *Testes gama* visam assegurar que o *software* está pronto para ser lançado no mercado de acordo com os requisitos especificados.

3.3.2 Testes não-funcionais

Diferentemente de *testes funcionais*, *testes não-funcionais* não são relacionados à funcionalidade da aplicação em si, mas sim a aspectos não-funcionais (escalabilidade, desempenho, confiabilidade, usabilidade etc.). *Testes não-funcionais* estão diretamente associados à qualidade do SUT que afeta a satisfação dos clientes.

O Quadro 3.5 apresenta exemplos de *tipos* de *testes não-funcionais* encontrados na literatura com suas respectivas descrições. Maiores detalhes sobre cada um deles são apresentados na sequência.

Quadro 3.5 - Testes não-funcionais.

Tipo de teste	Descrição	Referências
Interoperabilidade	Teste que busca assegurar que o produto de *software* é capaz de se comunicar com outros componentes ou dispositivos sem problemas de compatibilidade.	(IEEE, 1990) (DESIKAN, 2005) (PRESSMAN, 2016)
Compatibilidade	Tipo de teste usado para verificar se um produto de *software* é capaz de ser executado em diferentes plataformas, aplicações, ambientes de rede e dispositivos móveis.	(IEEE, 1990) (DESIKAN, 2005) (BARTIÉ, 2002)
Localização	*Testes de localização* validam o uso do *software* com idiomas e regiões geográficas específicas.	(DESIKAN, 2005) (SILVA, E. F. I.)
Globalização (I18N)	*Testes de globalização* (internacionalização - I18N) verificam se o *software* está adaptado a uma nova cultura, tais como diferentes moedas e fusos horários.	(SILVA, E. F. I.) (MYERS *et al.*, 2004)
Portabilidade	*Teste de portabilidade* é um tipo de teste no qual o *software* se comporta como esperado, mesmo após mudança para outro ambiente.	(IEEE, 1990) (SINGH e SINGH, 2012)
Volume	Usado para testar o *software* com uma certa quantidade de dados.	(BARTIÉ, 2002) (MYERS *et al.*, 2004)
Estresse	*Teste de estresse* é um tipo de teste de confiabilidade que enfatiza a avaliação de como o *software* responde em condições extremas.	(IEEE, 1990) (DESIKAN, 2005) (SINGH e SINGH, 2012)
Desempenho (*Performance*)	Tipo de teste implementado e executado para caracterizar e avaliar as funcionalidades relacionadas à *performance* do SUT, como fluxo de execução, tempo de resposta, confiabilidade e limites operacionais.	(IEEE, 1990) (DESIKAN, 2005) (SINGH e SINGH, 2012)
Carga	Tipo de teste usado para validar e avaliar a aceitação dos limites operacionais de um SUT em cargas de trabalho variadas.	(SINGH e SINGH, 2012) (BARTIÉ, 2002) (PRESSMAN, 2016)
Segurança	*Teste de segurança* é um tipo de teste que revela ameaças, vulnerabilidades e riscos em um SUT de maneira a prevenir ataques de atores malevolentes.	(SINGH e SINGH, 2012) (BARTIÉ, 2002) (PRESSMAN, 2016)
Intrusão	Tipo de teste de segurança com o objetivo de encontrar vulnerabilidades em um SUT que um possível invasor possa explorar.	(MORENO, 2019) (MENEZES, 2015)

Configuração	Tipo de teste com foco em garantir que SUT funcione conforme pretendido em diferentes configurações de *hardware* e *software*.	(BARTIÉ, 2002) (PRESSMAN, 2016)
Instalação	Tipo de teste que foca em garantir que SUT seja instalado como pretendido em diferentes configurações de *hardware* e *software* e em diferentes condições (como espaço insuficiente em disco e interrupções de energia)	(IEEE, 1990) (BARTIÉ, 2002)
Escalabilidade	Tipo de teste que objetiva medir o desempenho do SUT na medida em que o número de requisições do usuário aumenta.	(DESIKAN, 2005) (SINGH e SINGH, 2012)
Recuperação de desastres	Testes que verificam cada passo de um plano de recuperação de desastres, como especificado no processo de planejamento de recuperação e continuidade do negócio após desastres.	(IEEE, 1990) (PRESSMAN, 2016)
Mutação	Teste que altera determinadas instruções no código fonte e verifica se os casos de teste podem encontrar erros. É um tipo de teste com *abordagem de caixa branca* utilizado principalmente em *testes de unidade*.	(IEEE, 1990) (MALDONADO *et al.*, 2007)
Usabilidade	Tipo de teste executado em um SUT por pessoas que desempenham a atividade de completar uma lista de tarefas enquanto são observadas e suas interações anotadas.	(IEEE, 1990) (SINGH e SINGH, 2012) (BARTIÉ, 2002)
Confiabilidade	Tipo de *teste de desempenho*, executado sob condições pré-determinadas, no qual são validadas as entradas, saídas e operações efetuadas em relação aos requisitos definidos previamente para o SUT.	(IEEE, 1990) (DESIKAN, 2005) (BARTIÉ, 2002)

Interoperabilidade é a capacidade de dois ou mais sistemas de *software* interagirem entre si. Dessa forma, o *teste de interoperabilidade* é um tipo de teste de *software* que visa assegurar que o produto de *software* é capaz de se comunicar com outros componentes ou dispositivos sem problemas de compatibilidade. Em outras palavras, o *teste de interoperabilidade* busca provar que a funcionalidade fim-a-fim, entre dois sistemas de *software* que se comunicam, acontece conforme especificação de requisitos. Teste de comunicação entre *smartphones* e *tablets* para verificar a transferência de dados via *bluetooth* é um exemplo de *teste de interoperabilidade*. Outro exemplo é quando dois produtos de *software* distintos são capazes de trabalhar com um mesmo modelo de dados.

Teste de compatibilidade é um tipo de teste usado para verificar se um sistema de *software* é capaz de ser executado em diferentes plataformas, aplicações, ambientes de rede e dispositivos móveis. Frequentemente, o *teste de compatibilidade* é associado com teste *de interoperabilidade*, já que ambos os termos significam garantir que o produto de *software*, quando lançado, funcione corretamente com outros produtos presentes no mercado de *software*.

Os *testes de localização* validam o uso do *software* com idiomas e regiões geográficas específicas e são realizados para assegurar que não há erros de tradução, que o conteúdo está corretamente organizado e que o produto funciona como esperado para o público-alvo escolhido. Já os *testes de globalização* (internacionalização - I18N) verificam se o *software* está adaptado a uma nova cultura, tais como diferentes moedas e fusos horários. *Testes de globalização* focam nas funcionalidades do *software* em qualquer parte do mundo, enquanto que *testes de localização* focam em um subconjunto de usuários em uma dada cultura ou local.

Teste de portabilidade é um tipo de teste no qual o *software* se comporta como esperado, mesmo após mudança para outro ambiente. Dessa forma, implantamos o *software* em mais de um ambiente e testamos o seu comportamento. Um exemplo é portar um *software* que funcione, por exemplo, no sistema operacional *Microsoft Windows*, para o sistema operacional *Mac OS* da *Apple*. Outro exemplo, no setor de telefonia, é a portabilidade do número de telefone de uma operadora para outra.

Testes de volume são utilizados para testar o *software* com uma certa quantidade de dados. Esta quantidade (volume) pode ser associada ao tamanho do banco de dados em *terabytes*, por exemplo. No caso de um banco de dados sendo testado com *testes de volume*, necessitamos inserir dados no banco de dados até atingirmos o tamanho desejado de forma a testar o desempenho (*performance)* do *software* baseado neste volume. Se o objetivo for aplicar *testes de volume* em um arquivo (leitura ou escrita), podemos criar um arquivo com conteúdo textual (por exemplo, com as extensões `.xml`, `.txt`, etc) que é usado como SUT para os *testes de volume*. Nesse caso, o arquivo criado deverá ter o tamanho desejado para o *teste de volume*, de maneira a podermos aferir a *performance* do SUT.

Teste de estresse é um tipo de teste que enfatiza a avaliação de como o *software* responde em condições extremas. O estresse no *software* pode incluir enormes cargas de trabalho, memória insuficiente, serviços e *hardware* indisponíveis ou recursos compartilhados limitados. De maneira resumida, o *teste de estresse* baseia-se em testar os limites do *software* e avaliar seu comportamento. Assim, avalia-se até quando o *software* pode ser exigido e quais as falhas (se existirem) decorrentes do teste. Os *testes de estresse* são fundamentais em aplicações em que a *performance* seja um requisito importante.

Teste de desempenho é um tipo de testes implementado e executado para caracterizar e avaliar as funcionalidades relacionadas à *performance* do SUT. *Testes de desempenho* determinam, por exemplo, a velocidade, estabilidade, tempos de resposta, confiabilidade e limites operacionais do SUT.

Teste de carga é um tipo de *teste de desempenho* utilizado para validar e avaliar a aceitação dos limites operacionais de um *software* em cargas de trabalho variadas. Geralmente, as medições de *testes de carga* são tomadas com base na taxa de transferência de dados mediante uma carga de trabalho, alinhada com o tempo de resposta da transação sendo testada. Uma prática comum é variar a carga de trabalho incluindo simulações com cargas de trabalho médias e máximas adquiridas a partir de dados históricos do ambiente onde o SUT está implantado.

Teste de segurança é um tipo de teste que revela ameaças, vulnerabilidades e riscos em um SUT de maneira a prevenir ataques de atores malevolentes. De maneira geral, os testes de segurança incluem elementos específicos para proteção de dados como confidencialidade, integridade, disponibilidade, autenticação, autorização e não-repúdio.

Testes de intrusão (também conhecido como *hacking* ético, *teste de penetração*, e *teste de caneta - pentest*) é um tipo de *teste de segurança não-funcional* que avalia a segurança de um SUT a partir da simulação de um ataque por um ator mal intencionado. Testadores de intrusão buscam analisar o SUT objetivando identificar vulnerabilidades oriundas de diversos problemas, tais como má configurações, falhas em *hardware*/*software* desconhecidas, *bugs* no sistema operacional, dentre outros.

Teste de configuração foca em garantir que o SUT funcione conforme pretendido em diferentes configurações de *hardware* e *software*. Um dos objetivos de *testes de configuração* é descobrir a melhor configuração na qual o SUT pode operar sem falhas e de acordo com os requisitos do *software*.

Teste de instalação foca em garantir que SUT seja instalado como pretendido em diferentes configurações de *hardware* e *software* e em diferentes condições (como espaço insuficiente em disco e interrupções de energia). Esse teste é implementado e executado em aplicativos e sistemas em fases finais do desenvolvimento antes da primeira interação do usuário final com o *software*.

Teste de escalabilidade objetiva medir o desempenho do SUT na medida em que o número de requisições do usuário aumenta. Trata-se de um tipo de *teste de desempenho* que determina o limite no número de requisições simultâneas sobre o SUT.

Testes de recuperação de desastres englobam o exame de cada passo de um plano de recuperação de desastres. Normalmente são executados conforme processo de planejamento de recuperação e continuidade do negócio após desastres. Avaliar planos com *testes de recuperação de desastres* ajuda a assegurar que a organização possa recuperar dados, restaurar operações de negócio críticas e dar continuidade a operação após interrupção dos serviços.

Testes de mutação são usados para a criação de novos *casos de testes* e avaliação da qualidade de testes existentes. Com *testes de mutação*, pontos fracos nos dados usados bem como trechos de código que são pouco cobertos por testes ou não acessados durante a execução do SUT são identificados. Trata-se de uma *abordagem de testes de caixa branca*. O *teste de mutação* envolve a modificação de um SUT em pequenos passos (DEMILLO, LIPTON e SAYWARD, 1978), a partir da seleção e aplicação de um conjunto de operadores no SUT, um de cada vez. Cada uma dessas modificações gera uma nova versão do SUT conhecida como mutante. Se os *casos de testes* existentes em uma *suíte de testes* forem capazes de detectar a mudança (ou seja, se pelo menos um *caso de teste* falhar), o mutante é considerado morto. As *suítes de testes* são avaliadas por meio da porcentagem de mutantes que a *suíte* consegue "matar". Novos *casos de testes* podem ser criados para lidar com mutantes que não tenham sido "mortos".

Teste de usabilidade é um *tipo de teste* com *abordagem de caixa-preta* executado em um SUT por pessoas que desempenham a atividade de completar uma lista de tarefas enquanto são observadas e suas interações anotadas. O objetivo do *teste de usabilidade* é entender se o *design* proposto é útil e intuitivo o suficiente para que os usuários consigam realizar seus objetivos durante uso do SUT. O SUT alvo de *testes de usabilidade* pode ser uma aplicação *desktop*, um aplicativo para dispositivos móveis, uma aplicação *web* ou um produto em *hardware* e não necessariamente precisa estar completamente desenvolvido.

Teste de confiabilidade é um tipo de *teste de desempenho*, executado sob condições pré-determinadas (por exemplo, por uma quantidade de tempo pré-estabelecida), no qual são validadas as entradas, saídas e operações efetuadas em relação aos requisitos definidos previamente para o SUT. O principal objetivo é testar o desempenho do SUT, assegurando que o *software* esteja recebendo corretamente os dados, realizando o processamento adequadamente e apresentando os resultados corretamente.

3.4 AUTOMAÇÃO DE TESTES EM MÉTODOS ÁGEIS

Desenvolvedores ágeis (BECK e ANDRES, 2004) usam ferramentas para guiar a

automação de testes. Esta seção aborda duas ferramentas bastante utilizadas: *Quadrante de Testes* (GREGORY e CRISPIN, 2014) e *Pirâmide de Automação de Testes* (COHN, 2010).

3.4.1 Quadrante de testes

Um SUT é testado por uma série de razões: para encontrar *bugs*, para assegurar sua confiabilidade, para verificar a sua utilidade, dentre outros motivos. Os motivos normalmente fazem com que *testadores de software* empreguem diferentes tipos de testes no intuito de verificar o comportamento do SUT.

O *Quadrante de Testes* é uma ferramenta simples, representada por uma matriz, que auxilia *testadores de software* a assegurar que sejam considerados todos os diferentes tipos de testes necessários em atendimento a determinados motivos acima mencionados e para entrega de valor (ROMAN, 2010) aos clientes.

A Figura 3.1 exibe como cada um dos quatro quadrantes (Q1, Q2, Q3 e Q4) da ferramenta refletem os diferentes motivos pelos quais testamos. Eixos verticais dividem a matriz em *testes que suportam o time de desenvolvimento* e *testes que criticam o produto de software*. Eixos horizontais dividem a matriz em *testes voltados para o negócio* e *testes orientados por tecnologia*.

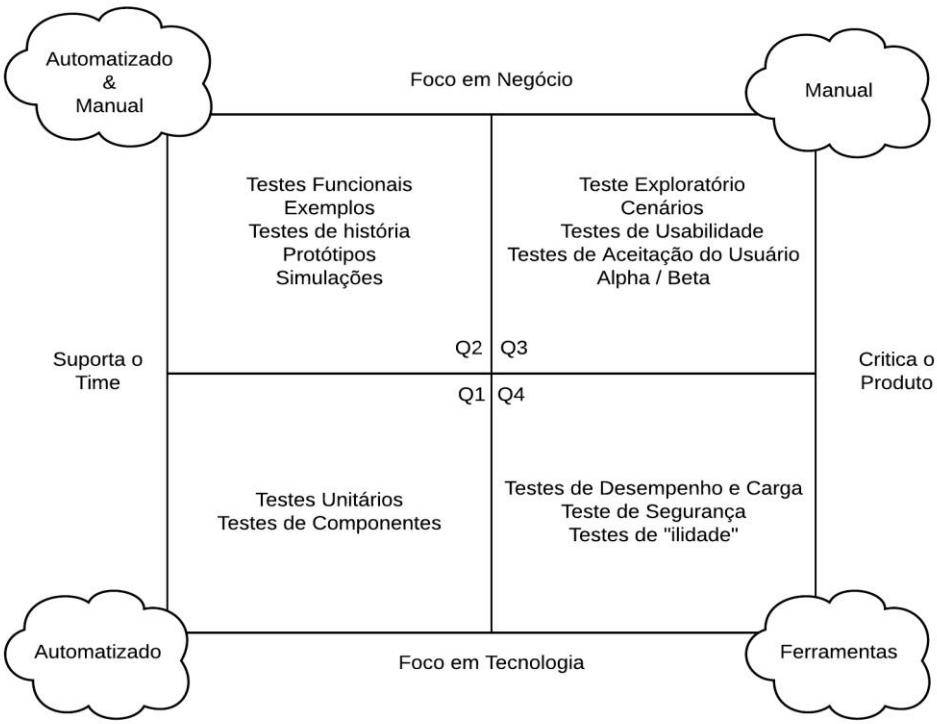

Figura 3.1 - Quadrante de Testes.
Fonte: Adaptada de Brian Marick, 2003.

A numeração dos quadrantes (Q1, Q2, Q3 e Q4) não implica ordem. Uma equipe de desenvolvimento não necessariamente segue os quadrantes Q1 a Q4 sequencialmente (em cascata). A ordem em que realizamos os diversos tipos de testes depende dos riscos do projeto, dos objetivos dos clientes com o *software*, da disponibilidade de recursos e outros fatores.

Os quadrantes à esquerda no *Quadrante de Testes* (Q1 e Q2) são usados como suporte ao time de desenvolvedores durante o desenvolvimento do *software* com o objetivo de entregar valor de negócio solicitado pelos clientes. O conceito de teste para auxiliar desenvolvedores é a maior diferença entre o processo de teste em projetos tradicionais e o processo de teste em métodos ágeis (GREGORY e CRISPIN, 2014).

O quadrante inferior à esquerda (Q1) na Figura 3.1 representa a prática de desenvolvimento dirigido por testes (do inglês, *Test Driven Development* - TDD) (BECK, 2003). TDD é prática fundamental em métodos ágeis em que testes são escritos e codificados por desenvolvedores antes de implementar código em atendimento aos testes. Desenvolvedores usam *testes de unidade* e *testes de componentes* em Q1.

Como já discutido neste trabalho, os *testes de unidade* testam as funcionalidades presentes em um objeto, de forma isolada de outros objetos que colaboram para o atendimento das funcionalidades. Os *testes de componentes* verificam o comportamento de uma parte maior do sistema quando comparados com os *testes de unidade*, tais como um grupo de classes que colaborativamente fornece algum serviço ao *software* (MESZAROS, 2007).

Ambos os tipos de testes (*unidade* e *componente*) são normalmente automatizados com uma das possíveis soluções da família 'xUNIT' (MESZAROS, 2007), tais como JUnit (TAHCHIEV, 2010) ou PHPUnit (MACHEK, 2014). Estes testes são desenvolvidos por desenvolvedores e não testadores e também são conhecidos como testes voltados a desenvolvedores e tem como objetivo mediar a qualidade interna do código (BECK, 2003). *Testes de unidade* e *componentes* tipicamente são escritos utilizando a mesma linguagem de programação em que o *software* está sendo desenvolvido.

Testes implementados por desenvolvedores, como de *unidade* e *componentes* são normalmente parte de um processo automatizado de construção (*build*) que é executado a cada *commit* de código no repositório de controle de versões, tal como Git (CHACON e BEN, 2014). O resultado da construção fornece *feedback* instantâneo ao time sobre a qualidade interna do código.

Os testes do segundo quadrante (Q2) também suportam o trabalho do time de desenvolvimento, mas em um nível mais alto de abstração. Trata-se de *testes do ponto de vista do cliente* (*customer-facing tests*) ou simplesmente *testes do cliente*. Testes em Q2 definem as

funcionalidades desejadas a partir de exemplos de uso fornecidos pelo cliente. Neste quadrante geralmente é utilizado desenvolvimento dirigido por comportamento, do inglês BDD - *Behavior Driven Development* (SMART, 2014).

Testes do ponto de vista do cliente são escritos usando a linguagem do domínio do cliente de forma a facilitar o entendimento. Os especialistas do negócio usam estes testes para definir a qualidade externa do *software*, sendo muitas vezes responsáveis pela escrita dos mesmos.

A maioria dos *testes voltados ao cliente* e que *suportam o time de desenvolvimento* precisa ser automatizada para que seja executada de forma rápida, frequente e para permitir *feedback* rápido sobre potenciais problemas no código. Sempre que possível, os testes automatizados do quadrante Q2 executam sobre a lógica de negócios do código de produção, sem uso da camada de apresentação responsável pela parte visual do SUT. Conforme mencionado anteriormente, é importante que os testes automatizados sejam executados como parte do processo automatizado de construção (*build*) do *software*.

Também fazem parte de Q2, testes em que especialistas em interação com o usuário usam protótipos e *wireframes* (TEIXEIRA, 2014) como ferramentas para validar propostas de *Interface* Gráfica com o Usuário (do inglês, *Graphical User Interfaces* - GUI). Nesse contexto, os testes também servem para comunicar decisões de *design* de GUIs com os desenvolvedores.

Os quadrantes à direita (Q3 e Q4) no *Quadrante de Testes* são normalmente usados para "criticar" o produto de *software*, por meio de uma avaliação detalhada do mesmo. O verbo criticar no contexto dos *quadrantes de testes* não é usado em um sentido negativo, mas sim no sentido de que uma crítica pode tanto incluir aprovações quanto sugestões de melhorias.

O terceiro quadrante, Q3, classifica os *testes voltados ao negócio* que simulam a forma como um usuário real usa o *software*. Trata-se de testes manuais desempenhados por pessoas. Podemos usar *scripts* automatizados para auxílio na configuração dos dados necessários para o teste, mas temos que usar nosso conhecimento do domínio para verificar se o time de desenvolvimento está entregando o valor de negócio solicitado pelos nossos clientes.

O quadrante Q3 engloba *testes de aceitação do usuário*, do inglês *User Acceptance Tests* - UAT - (Vide Quadro 3.4). *Testes de aceitação* fornecem aos clientes a chance de verificarem novas funcionalidades do *software* e refletirem sobre mudanças futuras, pois trata-se de uma forma de se obter *feedback* dos clientes sobre adaptações necessárias e novos requisitos. Métodos ágeis normalmente prescrevem alguma reunião (SUTHERLAND, 2014) para que os clientes avaliem as *histórias de usuário* sendo entregues pelo time de desenvolvimento. Não é incomum que algumas negociações e contratos demandem testes *de*

aceitação como um passo necessário para aprovação dessas *histórias*.

O Quadrante Q3 também prescreve *testes de usabilidade* (Quadro 3.5). Os *testes de usabilidade* são executados com um grupo de pessoas com o objetivo de obter a reação delas durante o uso do *software*. A obtenção de conhecimento de como as pessoas usam o *software* é uma vantagem quando estamos testando usabilidade.

A abordagem de *testes exploratórios* é central em Q3. Durante sessões de *testes exploratórios*, o *testador* simultaneamente projeta e executa os testes, usando raciocínio crítico para analisar os resultados. Observe que *teste exploratório* não é teste *ad-hoc*, que é feito sem planejamento e de forma improvisada. *Testes exploratórios* são guiados por estratégia e executados dentro de restrições definidas. Os *testes exploratórios* trabalham o *software* da mesma forma que usuários finais. Testadores usam criatividade e intuição. Como resultado, é a partir deste tipo de teste que muitos dos erros críticos do *software* são encontrados.

A abordagem de *testes exploratórios* normalmente inclui a definição de cenários. Desde o início de um projeto de desenvolvimento de *software*, testadores, com auxílio de especialistas do domínio, iniciam o trabalho de definir cenários de uso do *software*. O objetivo é testar o SUT fim-a-fim, mas não necessariamente como uma caixa preta. Na medida que código testável se torna disponível, testadores analisam os resultados da execução dos testes dos cenários identificados. O aprendizado obtido com os resultados da execução dos testes pode indicar a criação de novos cenários e testes.

No Quadrante Q4 encontram-se os *testes voltados à tecnologia (technology facing)*, discutidos em termos técnicos e não de negócios. Estes testes objetivam criticar características de qualidade do produto como desempenho, carga, escalabilidade, usabilidade e segurança. Os testes em Q4 devem ser considerados em cada passo do ciclo de desenvolvimento e não apenas na parte final do projeto.

Executar testes que criticam o produto de *software* o quanto antes no processo de desenvolvimento é crítico para o sucesso do projeto (GREGORY e CRISPIN, 2014). A informação produzida pelos testes que criticam o produto deve ser usada para retroalimentar as técnicas do lado esquerdo do *quadrante de testes* para a criação de novos testes e direcionar desenvolvimento futuro. Usar os quadrantes ajuda o planejamento de testes que criticam o produto e testes que direcionam o desenvolvimento do produto, auxiliando na execução dos testes necessários no momento adequado.

3.4.2 Pirâmide de automação de testes

A *pirâmide de automação de testes* (COHN. 2010) é uma ferramenta de classificação de testes de *software* em grupos de diferentes granularidades. Além da classificação, a pirâmide fornece uma ideia da quantidade de testes a ser executada em cada um dos grupos. A Figura 3.2 apresenta a pirâmide *de automação de testes*.

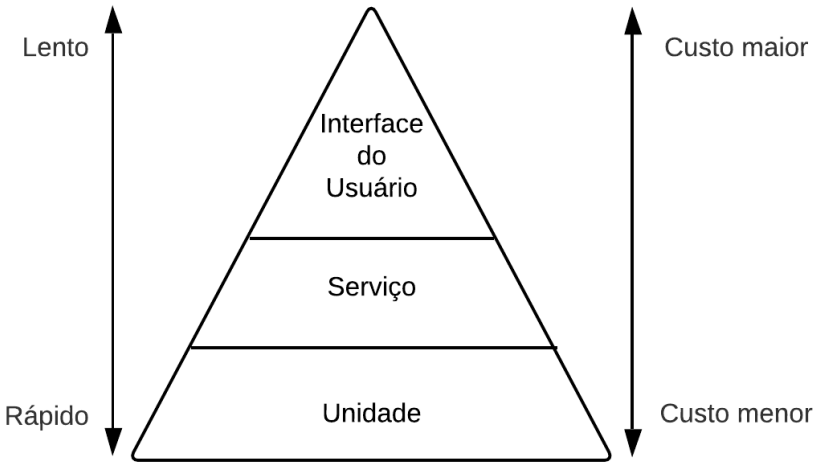

Figura 3.2 - Pirâmide de automação de testes
Fonte: Adaptada de Cohn, 2010.

Conforme pode ser visto na figura, na medida que escalamos a pirâmide em direção ao topo, os testes apresentam aumento no tempo de execução e no custo de desenvolvimento e manutenção, apesar de contribuírem para uma maior cobertura de testes e maior relevância para os clientes e negócios.

A base da pirâmide é constituída de *testes de unidade* e *componentes* voltados à tecnologia e que suportam o time de desenvolvimento. Esta camada representa testes normalmente escritos na mesma linguagem do SUT, usando a família *xUnit* de ferramentas (JUnit, PHPUnit, etc).

A camada do meio da pirâmide inclui a maioria dos testes automatizados voltados ao negócio e que são escritos para suportar o time. Trata-se de testes funcionais que verificam se estamos desenvolvendo o *software* de acordo com seus requisitos. Os testes desta camada podem incluir *testes de histórias de usuário*, de *aceitação* e testes que cobrem grandes conjuntos de funcionalidade quando comparados a *testes de unidade*. Estes testes operam no nível de API (*Application Programming Interface*), testando a funcionalidade diretamente sem uso de uma

GUI. Tendo em vista que estes testes desconsideram questões de GUI, eles são menos custosos de serem escritos e manter quando comparados com testes do topo da pirâmide.

Os testes da camada do meio da pirâmide são normalmente escritos em uma linguagem específica do domínio que os clientes conseguem entender, de forma que eles demandam maior esforço para serem escritos do que *testes de unidad*e. Eles normalmente rodam de maneira mais lenta já que cada teste cobre mais funcionalidade do que um *testes de unidade* e podem necessitar outros recursos, como por exemplo banco de dados. Fit (*Framework for integrated tests*) e FitNesse (MUGRIDGE, R.; CUNNINGHAM, W, 2005), e até mesmo planilhas com *casos de testes* são exemplos de ferramentas usadas na camada do meio da *pirâmide de automação de testes*.

A camada superior da pirâmide inclui testes não necessariamente automatizados, tendo em vista o menor retorno de investimento (*Return of Investiment* - ROI), já que envolvem alto custo e esforço para serem escritos. Estes testes são feitos por meio da GUI e são normalmente escritos quando o código já está finalizado com o objetivo de criticar o produto. De qualquer maneira, compensa frisar que, independentemente do número de testes automatizados criados, a maioria dos sistemas de *software* também necessita de testes manuais, tais como *testes exploratórios*.

Quando comparado com os testes de outras camadas da pirâmide, percebemos que os testes de GUI são mais lentos de serem executados. O resultado da execução de testes do da camada superior da pirâmide, mesmo que de forma automatizada, pode levar horas para ser obtido em vez de minutos quando comparamos com *testes de unidade* ou *componentes* (base da pirâmide).

Alguns autores argumentam que os nomes e tipos de testes de cada uma das camadas da *pirâmide de automação de testes* de Mike Cohn não são ideais (FOWLER, 2018). Para esses autores, os aspectos mais importantes a serem lembrados quando do uso da pirâmide são: (i) escrever testes de diferentes granularidades e; (ii) quanto maior o nível em que você está na pirâmide, menos testes devemos ter.

3.5 ESQUEMA DE CLASSIFICAÇÃO USADO NESTA MONOGRAFIA

Conforme discutido neste capítulo, existem na literatura diversas *abordagens, técnicas, tipos*, e *esquemas de classificação* de *teste de software*, englobando desde *testes manuais* até *testes automatizados*. Em alguns projetos, testes são executados durante todo o ciclo de

desenvolvimento do *software*; em outros, os testes são executados por um setor de garantia de qualidade no final do desenvolvimento.

Comunicar sobre diferentes *abordagens*, *técnicas*, *tipos* e *esquemas de classificação de testes* é difícil (FOWLER, 2018). O que é *teste de integração* para uma pessoa pode ter entendimento completamente diferente para outra pessoa. Para algumas pessoas, o *teste de integração* é uma atividade ampla que testa várias camadas distintas, englobando todo o SUT. Para outras pessoas, o *teste de integração* é uma atividade bastante restrita, que visa apenas testar a integração do SUT com um componente ou camada externa de cada vez. Algumas pessoas usam o termo *testes de integração* enquanto outras se referem a eles como *testes de componentes*; temos também pessoas que preferem o termo *testes de serviço*. Existem pessoas que argumentam que todos esses três termos são coisas totalmente diferentes.

Quando o assunto é classificação de testes, não existe certo ou errado, já que não há consenso entre membros da comunidade de desenvolvimento de *software* sobre termos relacionados à teste de *software*.

Tendo em vista que estamos interessados em métodos ágeis e testes automatizados, ficamos com a opção de usar um dos dois esquemas de classificação para testes automatizados (*quadrante de testes* ou *pirâmide de automação de testes*) descritos na Seção 3.4. Ambos os esquemas possuem o mesmo poder de representação, porém decidimos usar a *pirâmide de automação de testes* pôr a considerarmos mais intuitiva e simples de usar como esquema para direcionar os testes a serem criados nesta monografia. Fizemos pequenas adaptações nos tipos de testes usados em cada camada conforme sugerido por alguns autores (FOWLER, 2018) e optamos pelo não uso de testes manuais. De maneira específica, optamos por usar *testes de unidade* na base da pirâmide, *teste de integração* na camada intermediária e *testes de aceitação* na camada superior da pirâmide.

4 *HOSTELAPP*: PRINCIPAIS INTERESSADOS, REQUISITOS E ARQUITETURA

Este capítulo descreve uma aplicação de exemplo nomeada *HostelApp*, bem como seus principais interessados (*stakeholders*), requisitos e arquitetura. Desenvolvemos o *HostelApp* para atuar como SUT nas demonstrações de uso de diversos testes automatizados a serem discutidos nos Capítulos 5 e 6.

O *HostelApp* tem como principal funcionalidade gerenciar as reservas de um albergue, o que inclui as funcionalidades de gerenciar os hóspedes cadastrados e os quartos utilizados para acomodação.

4.1 PRINCIPAIS INTERESSADOS

Os *stakeholders* são pessoas ou organizações diretamente relacionadas no desenvolvimento de um produto ou serviço, podendo ser da área de *software* ou não (PRESSMAN, 2016). O *stakeholder* desempenha um papel (ou função) em um produto ou serviço, no que se refere a realizar seu objetivo de negócio. O Quadro 4.1 apresenta os principais papéis e seus respectivos objetivos com relação ao *HostelApp*.

Quadro 4.1 - Objetivos dos principais papéis identificados para o *HostelApp*.

Papel	**Objetivos**
Administrador	Seu principal objetivo é fazer a gestão dos dados de hóspedes, reservas e quartos para o *HostelApp*.
Hóspede	Seu principal objetivo é fazer a gestão dos dados de suas reservas e do seu cadastro no *HostelApp*.

4.2 REQUISITOS

Requisitos são as descrições das funções e restrições que o produto a ser desenvolvido deve possuir e, conforme discutido no Capítulo 3, normalmente são classificados em RFs e RNFs.

O Quadro 4.2 apresenta os principais RFs e RNFs do *HostelApp*. Note que cada requisito possui um identificador único o que permite melhor rastreabilidade entre os artefatos. É possível

observar neste quadro que os requisitos são descrições detalhadas dos objetivos presentes no Quadro 4.1.

O *Administrador* é responsável por grande parte das funcionalidades do *HostelApp*: cadastro, listagem, edição e exclusão de quartos, reservas e hóspedes (RF-01).

Quadro 4.2 - Principais RFs e RNFs implementados no *HostelApp*.

Id	Descrição
RF-01	Para o *Administrador*: - Criar, listar, atualizar e excluir quartos. - Criar, listar, atualizar e excluir reservas. - Criar, listar, atualizar e excluir hóspedes.
RF-02	Para o *Hóspede*: - Criar e atualizar seu cadastro junto ao *HostelApp*. - Criar, listar, atualizar e excluir reservas.
RNF-01	Para ambos os papéis acessarem as funcionalidades do *HostelApp* é necessário que estejam autenticados (autenticação de segurança).
RNF-02	Oferecer controle de acesso às páginas de diferentes papéis (autorização de segurança).

Já o *Hóspede* é responsável apenas pela utilização de funcionalidades básicas como a de realizar seu próprio cadastro no *HostelApp* e editá-lo, caso necessário, além de cadastrar, listar, editar e excluir reservas (RF-02).

Por motivos de segurança, para que *Administrador* e *Hóspede* consigam acessar o *HostelApp* é necessário que ambos se autentiquem no sistema via *login*, com as credenciais de *email* e senha (RNF-01). Além de autenticação, o *HostelApp* oferece controle do acesso dos usuários, restringindo o acesso à páginas que não possuem permissão (autorização). Por exemplo, na tentativa de acesso de uma página de *Administrador* pelo *Hóspede*, ele é automaticamente redirecionado ao seu perfil (RNF-02).

4.3 HISTÓRIAS DE USUÁRIO

Os RFs apresentados no Quadro 4.2 possuem alto nível de abstração, necessitando-se um maior detalhamento. Técnicas como *Casos de Uso* (GUEDES, 2018) e *Histórias de Usuário* (*User Stories*) (COHN, 2004) são amplamente utilizadas na prática. Optamos por usar *Histórias*

de Usuário neste trabalho por serem menos formais, servindo mais como um meio de comunicação sobre o correto entendimento dos requisitos a serem implementados quando comparadas com *Casos de Uso*.

No decorrer deste documento apresentamos exemplos com base nas *Histórias de Usuário* apresentadas no Quadro 4.3. Conforme pode ser visto no quadro, a *História de Usuário* US-001 está associada com o RF-01 descrito no Quadro 4.2 e as *Histórias de Usuário* US-002 e US-003 estão associadas com o RF-02.

Quadro 4.3 - Algumas das *Histórias de usuário* implementadas para o *HostelApp*

Id	**Descrição**	**Critério de aceitação**
US-001	Enquanto *Administrador* eu quero ter acesso centralizado à todas as funcionalidades do *HostelApp*: desde as reservas até os quartos e usuários cadastrados.	- Após realizar o *login* no *HostelApp*, deverão ser exibidas pelo *HostelApp* três opções: Gerenciar hóspedes, gerenciar quartos e gerenciar reservas.
US-002	Enquanto *Hóspede* eu quero efetuar meu cadastro no *HostelApp* para utilizar as funcionalidades oferecidas.	- O hóspede não deve conseguir cadastrar-se no *HostelApp* inserindo um *e-mail* já existente no banco de dados. - O hóspede deve conseguir realizar o cadastro no *HostelApp* tendo em vista que somente hóspedes cadastrados podem fazer reservas no *HostelApp*.
US-003	Enquanto *Hóspede* eu quero efetuar uma reserva no *HostelApp*.	- Após se cadastrar e realizar o *login* no *HostelApp*, deve ser exibido para o *Hóspede* a opção de criar uma nova reserva. - Data do *check-in* deve ser posterior à data de *check-out*.
US-004	Enquanto *Hóspede* eu quero gerenciar os dados das minhas reservas cadastradas.	- O hóspede deve conseguir atualizar os dados de suas reservas. - O hóspede deve conseguir excluir suas reservas.

4.4 TECNOLOGIAS

Nesta monografia utilizam-se várias tecnologias e ferramentas de *software livre*, selecionadas para serem utilizadas na implementação do *HostelApp*.

A parte visual (*front-end*) do *HostelApp* baseia-se na tecnologia React.js (REACT, 2013). Java (JAVA, 1996) é usada no *back-end.* Cabe ao módulo *back-end* a intermediação dos dados entre o *front-end* e o banco de dados do *HostelApp*, além de gerir o sistema de reservas do albergue por meio do estilo arquitetural RESTful (SPRING BOOT, 2014). RESTful é utilizado para realização de requisições de recursos via HTTP (*HyperText Transfer Protocol*) por meio dos métodos GET (usado para realizar consultas em recursos), POST (usado para criar um recurso), PUT (usado para atualizar um recurso) e DELETE (usado para excluir um recurso).

Usamos Spring Boot (SPRING BOOT, 2014) como *framework* MVC (*Model*, *View*, *Controller*). O Spring Boot facilita a criação de aplicações *web* em Java com base no conceito de convenção sobre configuração, evitando que o desenvolvedor necessite estabelecer excessivas configurações. Este *framework* oferece um sítio eletrônico (SPRING INITIALIZR, 2014) para a geração de projetos iniciais com Spring Boot, já com a possibilidade de estabelecimento de dependências iniciais (bibliotecas).

Para a criação do *front-end* utilizou-se, além da linguagem JavaScript (JAVASCRIPT, 1995), a biblioteca React (também denominada React.js ou ReactJS). React.js é uma biblioteca da linguagem JavaScript de código aberto com foco em criar componentes de GUI para páginas *web*.

Por fim, utilizou-se MySQL (MYSQL, 1995) para o armazenamento dos dados de reservas feitas no *HostelApp*. MySQL é um Sistema de Gerenciamento de Banco de Dados (SGBD) que utiliza a linguagem SQL para manipulação de dados.

Tendo apresentado as principais tecnologias usadas para implementação do *HostelApp*, a seguir descrevem-se as ferramentas utilizadas para execução dos testes automatizados criados como parte do objetivo deste trabalho.

JUnit (JUNIT, 1998) é uma ferramenta para realização de testes de *software* que possui recursos de verificação de comportamento do *software* por meio de asserções. Dentre as possíveis asserções, destacam-se `assertEquals` (compara a igualdade entre os valores dos atributos dos objetos passados como parâmetro), `assertTrue` (compara se a informação passada como parâmetro é verdadeira), `assertNull` (verifica se uma dada referência é nula), dentre outras. As asserções, servem para assegurar que os resultados obtidos dos testes são iguais aos resultados esperados. A ferramenta JUnit normalmente trabalha em conjunto com um framework para criação de *mocks*, tal como Mockito (MOCKITO, 2008). Mockito é uma biblioteca que permite criar testes usando *mocks* (protótipos de objetos reais usados para isolar

a funcionalidade a ser testada). Mocks isolam as dependências entre objetos para maior controle sobre os resultados dos testes.

O MockMVC (MOCKMVC, 2014) é usado para realizar requisições aos *endpoints* presentes no *back-end* do *HostelApp*. Com o MockMVC é possível passar para o *backend*, parâmetros, cabeçalhos, conteúdo e tudo aquilo que é aceito por uma requisição HTTP. O objetivo do uso do MockMVC nesta monografia é verificar se o resultado obtido é igual ao esperado, indicando, assim como é feito com o JUnit, se o teste passou ou não.

O Selenium WebDriver (SELENIUM, 2004), é uma ferramenta que permite realizar *testes de aceitação* e *testes fim-a-fim*, simulando a real utilização de um usuário dentro do sistema. Com esta ferramenta é possível navegar pelas páginas do *HostelApp*, além de preencher e testar os valores dos campos de formulários *web*, funcionamento de botões presentes na tela, dentre outras funcionalidades. O Selenium WebDriver oferece suporte a navegadores *web* como Google Chrome, Mozilla Firefox, Microsoft Edge e Opera.

4.5 DIAGRAMA DE IMPLANTAÇÃO UML

O Diagrama de Implantação (*deployment*) UML (LARMAN, 2004) é um diagrama que exibe a distribuição de componentes de *software* em uma infraestrutura física de *hardware*. A Figura 4.1 ilustra o diagrama de *deployment* com componentes de *software* que colaboram entre si para a realização do requisito funcional de criação de reservas (veja Quadro 4.2). Os outros RFs descritos na Seção 4.2 seguem o mesmo fluxo do requisito de criação de reservas, pois a arquitetura do *HostelApp* foi concebida utilizando o padrão arquitetural MVC.

A arquitetura MVC é um padrão de construção de *software* que divide uma aplicação em três camadas: a camada de interação com o usuário (*view*), a camada que encapsula os dados a do domínio da aplicação (*model*) e a camada de controle (*controller*), que coordena o fluxo de controle entra a *view* e o *model*. Regras simples de negócio podem ser implementadas diretamente pelo *controller*, embora esta alternativa apresente problemas de baixa coesão (LARMAN, 2004). A execução de regras de negócio mais complexas normalmente é delegada pelo *controller* a outros componentes de *software*, como por exemplo componentes de serviço (EVANS, 2003). Cabe também ao *controller* redirecionar para as *views* os resultados da execução das regras de negócios presentes no *model*.

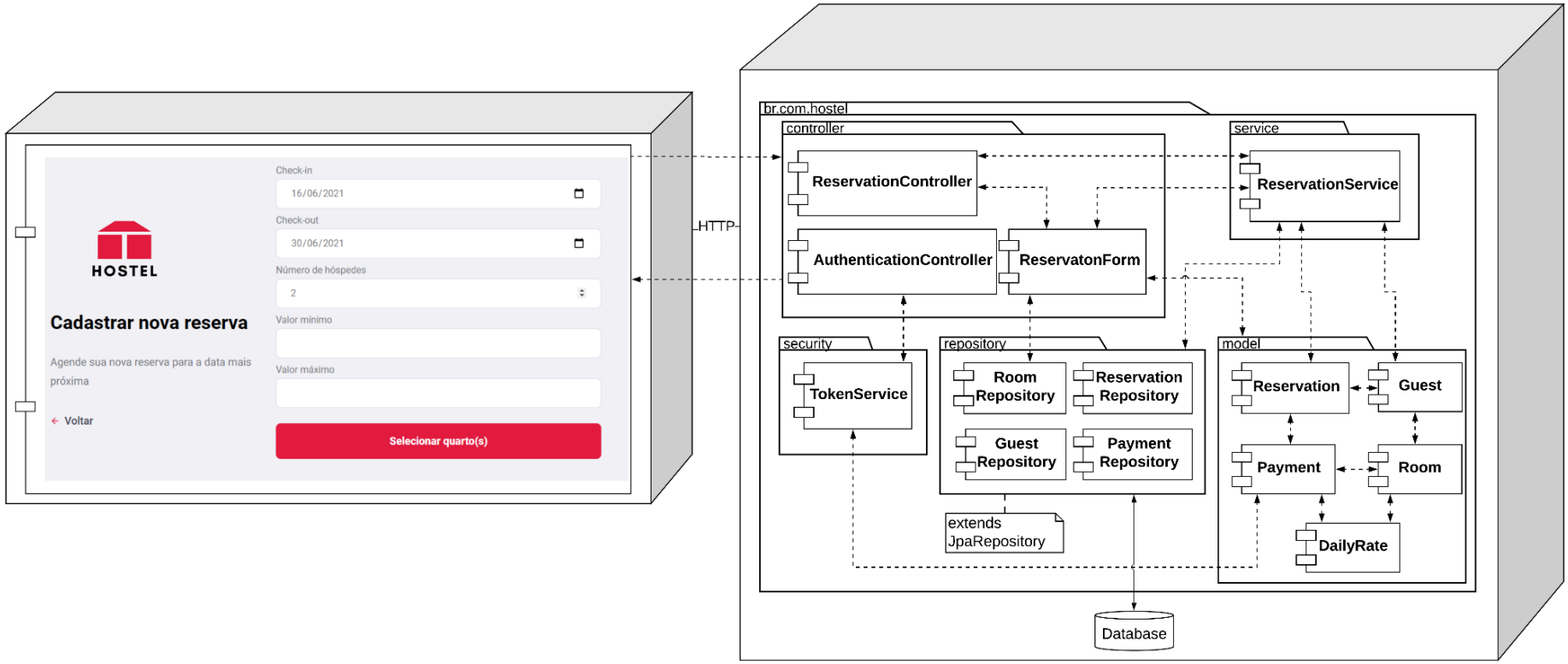

Figura 4.1 - Diagrama UML de *Deployment.*

Fonte: Figura criada pelos autores.

Para que seja possível criar uma reserva, o *hóspede* precisa fazer *login* no *HostelApp*. Caso o *hóspede* não possua um cadastro há uma opção para que ele consiga se cadastrar. Após o *hóspede* cadastrado preencher os campos de *email* e senha, os dados são submetidos para o *back-end* do *HostelApp* via requisição HTTP. O primeiro componente no *back-end* a receber os dados submetidos é o `AuthenticationController`. Cabe a este componente extrair os dados da requisição HTTP e acionar o método de autenticação presente no componente `TokenService`. Este componente verifica se os dados recebidos para autenticação coincidem com o que está armazenado e, em caso afirmativo, autentica o usuário e retorna um *token* de autenticação para o `AuthenticationController`. O *HostelApp* utiliza este *token* para verificar a autenticidade do usuário ao realizar novas requisições.

Após realizar a autenticação no *HostelApp*, o *hóspede* pode acessar a funcionalidade de criação de reservas, selecionar as opções de datas de entrada (*check-in*) e saída (*check-out*), o número de hóspedes para a reserva e, se preferir, os valores mínimos e máximos da diária dos quartos, a fim de que seja possível filtrar os quartos que devem aparecer nos resultados da pesquisa (cubo que representa um nó de *hardware*, à esquerda na Figura 4.1). Então, o *hóspede* escolhe os quartos disponíveis de acordo com as opções anteriormente selecionadas e posteriormente escolhe a forma de pagamento desejada.

Com todas as informações necessárias reunidas, o componente `ReservationController` recebe os dados e aciona o método `registerReservation` (não exibido na Figura 4.1) presente no componente `ReservationService`, passando como parâmetro uma instância do componente `ReservationForm` que contém todas as informações recebidas por meio do *front-end*.

Dentro do componente `ReservationService` realiza-se as validações dos dados enviados pelo *hóspede* e, caso todos estejam corretos, realiza-se o armazenamento dos objetos utilizando os componentes presentes no pacote `repository`, que é o pacote responsável por fazer a comunicação com o banco de dados. Vale ressaltar que no *front-end* do *HostelApp* também são realizadas as validações dos dados logo que o *hóspede* envia o formulário, não permitindo que o *hóspede* realize todo o fluxo de criação de uma reserva para somente no final deste processo descobrir que algum dado está inválido. Fazer a validação de dados tanto no *front-end* quanto no *back-end* de aplicações *web* é uma boa prática de segurança (HALL e BROWN, 2003).

5 PLANO DE TESTES PARA O *HOSTELAPP*

Este capítulo apresenta um plano de testes para o *HostelApp*. O plano inclui cenários de teste para o cadastro, remoção e listagem de hóspedes utilizando *testes de unidade* (camada inferior da *Pirâmide de Automação de Testes*) e cenários de teste para criação de reservas utilizando *testes de integração* (camada intermediária da *Pirâmide de Automação de Testes* discutida no Capítulo 3) e *testes de aceitação* (camada superior da *Pirâmide de Automação de Testes*). Os testes foram implementados a fim de testar os requisitos listados na Seção 4.2 deste documento.

5.1 CASOS DE TESTE DE UNIDADE PARA OS REQUISITOS CADASTRAR, EXCLUIR E LISTAR HÓSPEDES

Os Quadros 5.1, 5.2 e 5.3 apresentam *casos de teste de unidade* que objetivam testar a camada de serviço do *HostelApp*, verificando o correto cadastro, exclusão e listagem de hóspedes. Por se tratar de *testes de unidade*, isolamos a camada de serviço do *HostelApp* das demais camadas. Dessa forma, os hóspedes, repositórios e demais camadas que não estão contidas na camada de serviço do *HostelApp* são simulados, utilizando o *framework* Mockito.

Conforme pode ser visto nos quadros, em cada caso de teste é feita uma validação em um cenário específico, com o intuito de verificar resultados inesperados a fim de evitar possíveis erros e conflitos no *HostelApp*.

Quadro 5.1 - *Caso de teste* 1: Cadastro de um usuário com o nome Washington Ferrolho.

Descrição	**Cenário**	**Resultados esperados**
Neste *caso de teste* realiza-se o cadastro de um hóspede no *HostelApp*.	Em um cenário simulado é realizado o cadastro de um hóspede chamado Washington Ferrolho, com o *e-mail* washington@orkut.com, nascido em 12 de dezembro de 1900, residente na Rua Governador Valadares, CEP: 13900-000, Amparo, São Paulo, Brasil.	O hóspede de nome Washington Ferrolho deverá ser cadastrado com sucesso.

Quadro 5.2 - *Caso de teste* 2: Exclusão de um hóspede simulado.

Descrição	Cenário	Resultados esperados
Neste *caso de teste* realiza-se a exclusão de um hóspede do *HostelApp*.	Exclusão de um hóspede previamente cadastrado no *HostelApp*.	O hóspede deve ser excluído com sucesso e não deve ser mais encontrado ao realizar-se uma busca utilizando seu *id*, ocasionando o lançamento de uma exceção.

Quadro 5.3 - *Caso de teste* 3: Listagem de hóspedes simulados.

Descrição	Cenário	Resultados esperados
Neste *caso de teste* realiza-se a listagem de hóspedes do *HostelApp*.	Cadastrar dois hóspedes em um banco de dados inicialmente vazio (sem nenhum hóspede). Na sequência exibe-se a lista com os hóspedes cadastrados no *HostelApp*.	O tamanho da lista de hóspedes retornada deverá ser igual à dois.

5.2 CASOS DE TESTE DE INTEGRAÇÃO PARA O REQUISITO CRIAR RESERVAS

Os Quadros 5.4, 5.5, 5.6 e 5.7 apresentam *casos de teste de integração* que objetivam verificar a correta criação de reservas no *HostelApp*. Conforme pode ser visto nos quadros, em cada caso de teste é feita uma requisição HTTP POST[1] utilizando o endereço eletrônico descrito na coluna *Requisição*. O corpo da requisição contém dados a respeito de uma reserva que será submetida para o servidor do *HostelApp* (para não poluir os quadros, optamos por não incluir todos os dados de uma reserva, usando reticências para indicar a existência de mais atributos). Ao chegar no servidor, os dados da reserva encapsulados na requisição são instanciados no formato de um objeto do tipo `ReservationForm`, utilizado no código do *back-end* para obter os atributos necessários para a realização do cadastro.

Os *casos de teste* têm o intuito de verificar se os resultados retornados pela requisição estão de acordo com os resultados esperados, presentes na coluna *Resultados esperados* em cada quadro.

[1] De acordo com a especificação HTTP, para métodos de envio do tipo POST, os parâmetros não são enviados na URL da requisição e sim no corpo da requisição.

Quadro 5.4 - *Caso de teste* 4: Criação de reserva associada a um hóspede inexistente.

Descrição	Requisição	Resultados esperados

Neste *caso de teste* é feita a tentativa de cadastro de uma reserva para um hóspede com um *id* inexistente no banco de dados.	POST em `/api/reservations`
	
	`{`
	`  "guest_ID": 0,`
	`  ...`
	`}`
		Status de resposta HTTP 404 (*Not Found*).

Quadro 5.5 - *Caso de teste* 5: Criação de reserva com data de *check-in* inválida.

Descrição	Requisição	Resultados esperados

Neste *caso de teste* é feita a tentativa de cadastro de uma reserva em que um de seus atributos é uma data de *check-in* anterior à data atual da realização da reserva.	POST em `/api/reservations`
	
	`{`
	`  "checkinDate: "1900-10-10",`
	`  ...`
	`}`
		Status de resposta HTTP 400 (*Bad Request*).

Quadro 5.6 - *Caso de teste* 6: Criação de uma reserva com a data de *check-out* inválida.

Descrição	Requisição	Resultados esperados

Neste *caso de teste* é feita a tentativa de cadastro de uma reserva em que um de seus atributos é uma data de *check-out* anterior à data de *check-in*.	POST em `/api/reservations`
	
	`{`
	`  "checkinDate: "2022-10-10",`
	`  "checkoutDate: "2022-10-09",`
	`  ...`
	`}`
		Status de resposta HTTP 400 (*Bad Request*).

Quadro 5.7 - *Caso de teste 7*: Criação de uma reserva com todos os atributos válidos.

Descrição	**Requisição**	**Resultados esperados**

Neste *caso de teste* é feita a tentativa de cadastro de uma reserva com todos os atributos válidos.	POST em `/api/reservations`
	
	`{`
	`"checkinDate: "2025-04-01",`
	`"amount": 3000.00,`
	`"bankName: "Itau",`
	`...`
	`}`
		Status de resposta HTTP 201 (*Created*)
		
		Os dados do objeto retornado deverão ser iguais aos dados do objeto enviado na requisição

Além dos *casos de testes* descritos nesta seção, criamos diversos outros *casos de testes de integração* no intuito de verificar as funcionalidades implementadas no *HostelApp*. Consulte o Apêndice A para maiores detalhes.

5.3 CASO DE TESTE DE ACEITAÇÃO PARA O REQUISITO CRIAR RESERVAS

O Quadro 5.8 apresenta o *caso de teste de aceitação* que objetiva assegurar a correta criação de reservas no *HostelApp*. Conforme pode ser visto no quadro, o *teste de aceitação* tem como objetivo verificar o funcionamento do sistema em relação aos seus requisitos originais e às necessidades do *hóspede*. Assim, o teste passa por todo o fluxo de criação de reservas em um único cenário, simulando a interação do usuário com as telas.

Quadro 5.8 - *Caso de teste 8*: Criação de reserva fim-a-fim.

Descrição	**Cenário**	**Resultados esperados**
Neste *caso de teste* um *Hóspede* realiza o cadastro de uma reserva com sucesso.	Um *hóspede* cadastrado no *HostelApp* faz o *login* na aplicação digitando seu usuário e senha. Após realizar o *login*, o *hóspede* seleciona a opção "Cadastrar nova reserva" e informa a data de *check-in* para o dia 24/10/2025 e a de *check-out* para o dia 26/10/2025. Na sequência, informa 3 pessoas como número de hóspedes da reserva e seleciona a opção "Selecionar Quartos". Na tela exibida de seleção de quartos, o *hóspede* seleciona dois quartos disponíveis e seleciona a opção "Forma de pagamento". Na tela exibida para forma de pagamento, o *hóspede* opta por pagamento em dinheiro e finaliza o cadastro da reserva.	A reserva deve ser criada com sucesso e ser listada na lista de reservas presente na tela de perfil do hóspede.

6 IMPLEMENTAÇÃO DOS TESTES AUTOMATIZADOS PARA O *HOSTELAPP*

Este capítulo apresenta as implementações dos *casos de testes* descritos no capítulo anterior. Decidimos descrever apenas um exemplo de implementação para cada tipo de *caso de teste*, visto que as classes de testes seguem o mesmo padrão de implementação de código, não havendo necessidade de replicação dos trechos de código dessas classes neste documento.
Os Apêndices B e C apresentam, respectivamente, mais exemplos de implementação de *casos de testes* e o *link* para o repositório do projeto onde são encontrados todos os exemplos de testes automatizados implementados.

6.1 IMPLEMENTAÇÃO DO CASO DE TESTE DE UNIDADE PARA O REQUISITO CADASTRAR HÓSPEDE

O Código 6.1 apresenta um trecho de código presente na classe `CreateGuestsTest`, contendo a implementação dos *testes de unidade* do requisito de cadastrar hóspedes.

```
1. @ExtendWith(SpringExtension.class)
2. @SpringBootTest(classes = GuestService.class)
3. public class CreateGuestsTest {
4.     @MockBean
5.     private GuestRepository guestRepository;
6.
7.     @MockBean
8.     private AddressRepository addressRepository;
9.
10.    @MockBean
11.    private GuestForm guestForm;
          ...
```

Código 6.1 - Variáveis utilizadas nos *testes de unidade* para o requisito cadastrar hóspede.
Fonte: Código implementado pelos autores.

Inicialmente simula-se todo o repositório de Hóspedes (Linhas 04 e 05), de Endereços (Linhas 07 e 08) e também o objeto Formulário de Hóspedes (Linhas 10 e 11). A simulação desses objetos é feita através da anotação `@MockBean`, que é utilizada para criar uma instância de um objeto que simula o comportamento de uma classe ou objeto real. Essa simulação é feita com o intuito de isolar a camada de serviço das demais camadas.

As implementações dos casos de testes estão organizadas em métodos denotados pelas anotações do JUnit `@BeforeAll` ou `@BeforeEach` e `@Test`. Nos métodos denotados por `@BeforeAll` — executado antes de todos os métodos de teste presentes na classe — e `@BeforeEach` — executado antes de cada método de teste presente na classe — há implementações e configurações iniciais dos objetos que são necessárias para a realização dos testes. Os métodos denotados por `@Test` possuem a implementação do que será realmente testado pela classe de teste.

```
1. @BeforeAll
2. public static void beforeAll() throws Exception {
3.
4.     GuestsInitializer.initialize(address, guest);
5. }
```

Código 6.2 - Método de configurações iniciais com as informações necessárias para os testes de unidade para cadastrar hóspede.

Fonte: Código implementado pelos autores.

No Código 6.2 realiza-se a chamada do método `initialize` presente na classe `GuestsInitializer` (Linha 04), onde são definidos os atributos do objeto do *hóspede* que é utilizado nos testes.

```
1. @Test
2. public void shouldCreateOneGuestSuccessfully() throws Exception
3. {
4.
5.     Optional<Guest> nonexistentGuest = Optional.empty();
6.
7.     when(guestRepository.findByEmail(any()))
8.         .thenReturn(nonexistentGuest);
9.     when(guestForm.returnGuest(any())).thenReturn(guest);
10.    when(guestRepository.save(any())).thenReturn(guest);
11.
12.    Guest reqGuest =
13.    guestService.createGuest(guestForm, uriBuilder);
14.
15.    assertEquals(guest.getName(), reqGuest.getName());
16.    assertEquals(guest.getLastName(), reqGuest.getLastName());
17. }
```

Código 6.3 - Método de teste para cadastrar um hóspede com sucesso.

Fonte: Código implementado pelos autores.

O Código 6.3 apresenta o método de teste `shouldCreateOneGuestSuccessfully`, que tem como objetivo assegurar o correto funcionamento da criação de um hóspede.

O método é iniciado com a criação de um objeto vazio `Optional` de `Guest` denominado `nonexistentGuest` (Linha 05). Em seguida, temos implementações do método `when`, que simula o retorno do método parametrizado, através da chamada do método `thenReturn` (Linhas 07 e 08).

Na primeira chamada do método `when` (Linhas 06 e 07), simula-se que não existe nenhum registro de hóspede quando a chamada do método `findByEmail` (método presente dentro fluxo de criação de hóspedes, que verifica se existe um *hóspede* com o mesmo *e-mail* do objeto a ser cadastrado no HostelApp) for realizada.

Nas chamadas subsequentes do método `when` (Linhas 09 e 10), simula-se que o retorno do método `returnGuest` presente dentro de `guestForm` e o retorno do método `save` presente dentro de `guestRepository` devem ser o objeto de `Guest` inicializado no Código 6.2. Assim, realiza-se a chamada do método de criação de hóspedes `createGuest` (Linha 12 e 13), que contém os cenários simulados nas Linhas 07 a 10 do Código 6.3.

Por fim, os métodos `assertEquals` (Linhas 15 e 16) validam, respectivamente, se o nome e o último nome do *hóspede* previamente inicializado no Código 6.1 são os mesmos do objeto retornado pelo método `createGuest`.

6.2 IMPLEMENTAÇÃO DO CASO DE TESTE DE INTEGRAÇÃO PARA O REQUISITO CRIAR RESERVAS

Os Códigos 6.4 e 6.5 apresentam trechos de código presentes na classe `CreateReservationsTest`, contendo a implementação necessária para que seja possível testar o requisito de criação de reservas (RF-02 no Quadro 4.2).

No código 6.4 é realizado a implementação do `@BeforeEach` para as configurações iniciais: dados de *login* (pois é necessária a autenticação para realizar as requisições), a criação de objetos e definições de seus atributos, o armazenamento destes objetos no banco de dados quando necessário, entre outras configurações possíveis.

```
1. @BeforeEach
2. public void init() throws JsonProcessingException, Exception {
3.
4.    uri = new URI("/api/reservations/");
5.
6.   ReservationInitializer.initialize(headers, reservationForm,
7.      checkPayment, rooms_ID, mockMvc,
8.      objectMapper);
9. }
```

Código 6.4 - Método de configurações iniciais necessárias para os testes de criar reservas.
Fonte: Código implementado pelos autores.

Inicialmente no método `init`, denotado pela anotação `@BeforeEach`, define-se a URL (*endpoint*) na qual são realizadas as requisições dos testes (Linha 04). Feito isso realiza-se a chamada do método `initialize` (Linhas 06 e 07), que por sua vez, invoca um método necessário para realização do *login* (objeto `LoginInitializer`, não exibido no código) e definição dos atributos dos objetos utilizados nos testes.

```
1. @Test
2. public void
3. shouldCreateOneReservationUsingCheckPaymentSuccessfully()
4. throws Exception {
5.
6.   MvcResult result =
7.         mockMvc
8.         .perform(post(uri)
9.         .headers(headers)
10.        .content(objectMapper.writeValueAsString(
11.         reservationForm)))
12.        .andDo(print())
13.        .andExpect(status().isCreated())
14.        .andReturn();
15.
16.   String contentAsString =
17.   result.getResponse().getContentAsString();
18.
19.   ReservationDto reservationObjResponse =
20.   objectMapper.readValue(contentAsString, ReservationDto.class);
21.
22.   CheckPayment checkObjResponse =
23.   (CheckPayment) reservationObjResponse.getPayments();
24.
25.   assertEquals(reservationForm.getCheckinDate(),
26.   reservationObjResponse.getCheckinDate());
27.
```

```
29.    assertEquals(checkPayment.getAmount(),
30.    reservationObjResponse.getPayment().getAmount());
31.
32.    assertEquals(checkPayment.getBankName(),
33.    checkObjResponse.getBankName());
34. }
```

Código 6.5 - Método de teste para criar uma reserva utilizando cheque como forma de pagamento, retornando o código de *status* HTTP *"created"* (201).
Fonte: Código implementado pelos autores.

Feito *login* e definidos os atributos para realizar a reserva, utiliza-se o `MockMvc` (Linhas 06-14 no Código 6.5) para realizar a requisição HTTP POST para a URL mencionada na Linha 03 do Código 6.4. Insere-se no cabeçalho (*header*) da requisição a autenticação (Linha 09) e o objeto da reserva (Linhas 10 e 11), ambos pré definidos no método `init` do Código 6.4. Feita a requisição, espera-se receber um status HTTP 201 "*CREATED*" como resposta (Linha 13), indicando que a requisição obteve êxito e que o objeto foi criado.

Realizada a requisição utilizando o `MockMvc`, armazena-se os dados obtidos na resposta da requisição em objetos do tipo `String`, `ReservationDto` e `CheckPayment` (Linhas 16-23), pois são utilizados como parâmetro de validação para os métodos do tipo `assertEquals` do JUnit, a fim de verificar se o que foi retornado pelo servidor contém os mesmos atributos do objeto previamente definido no método `init` do Código 6.4 descrito anteriormente.

O primeiro `assertEquals` (Linhas 26 e 27) verifica se a data de *check-in* do objeto de reserva previamente armazenado para os testes é o mesmo do objeto retornado pela requisição.

O segundo `assertEquals` (Linhas 29 e 30) verifica se valor total do pagamento contido dentro do objeto de reserva previamente armazenado para os testes é o mesmo valor total do pagamento contido dentro do objeto retornado pela requisição.

Por fim, o terceiro `assertEquals` (Linhas 32 e 33) verifica se o nome do banco do objeto de reserva previamente armazenado para os testes é o mesmo nome do banco do objeto retornado pela requisição.

6.3 IMPLEMENTAÇÃO DO CASO DE TESTE DE ACEITAÇÃO PARA O REQUISITO CRIAR RESERVAS

Para os *testes de aceitação* utilizamos o Selenium WebDriver, que tem como pré-requisito a definição de uma conexão com um navegador. O Código 6.6 apresenta um trecho de código presente na classe `ChromeConnection`, responsável por realizar a configuração da conexão com o Google Chrome (navegador *web* escolhido) para os *testes de aceitação*.

Para realizar a conexão é necessário efetuar o *download* do *driver* do navegador *web* escolhido para os testes na máquina local. Os *drivers* dos navegadores *web* estão disponíveis no próprio sítio eletrônico do Selenium.

```
1. public WebDriver Connection() {
2.     File file =
3.               new File("./src/test/resources/chromedriver.exe");
4.     System.setProperty("webdriver.chrome.driver",
5.     file.getAbsolutePath());
6.     ChromeOptions options = new ChromeOptions();
7.     WebDriver driver = new ChromeDriver(options);
8.
9.     return driver;
10. }
```

Código 6.6 - Configurações iniciais necessárias para os *testes de aceitação* para conexão com o navegador *web*.

Fonte: Código implementado pelos autores.

Inicialmente, ao instanciar o objeto `file` (Linhas 02 e 03), indica-se o caminho — local dentro da máquina local em que está sendo executado o projeto — onde está instalado o *driver* do navegador *web*. Em seguida, define-se as propriedades do sistema: qual é o nome da propriedade do *driver* do navegador *web* escolhido e o caminho do *driver* (Linha 04 e 05) descrito anteriormente. Na Linha 06, instancia-se o objeto `ChromeOptions` — utilizado para gerenciar configurações específicas para o `ChromeDriver` — e adiciona-o como parâmetro do construtor do objeto `WebDriver` (Linha 07) — responsável por conduzir o navegador *web* nativamente utilizando o servidor do Selenium, da mesma forma que um usuário faria —, que posteriormente é retornado pelo método `Connection` (Linha 09).

Com o `WebDriver` já conectado e configurado, o próximo passo é acessar o *HostelApp*. O Código 6.7 apresenta como é realizado o *login*.

```
1. @BeforeEach
2. public void init() throws InterruptedException {
3.     driver.get("http://localhost:3000/");
4.     driver.manage().window().maximize();
5.     driver.findElement(
6.     By.xpath(
7.          "//*[@id=\"root\"]/div/div/section/form/input[1]"))
8.          .sendKeys("daniel@email.com");
9.     Thread.sleep(1000);
10.
11.    driver.findElement(
12.    By.xpath(
13.         "//*[@id=\"root\"]/div/div/section/form/input[2]"))
14          .sendKeys("123456");
15.    Thread.sleep(1000);
16.
17.    driver.findElement(
18.
19.   By.xpath("//*[@id=\"root\"]/div/div/section/form/button"))
20     .click();
21.    Thread.sleep(3000);
22. }
```

Código 6.7 - Método inicial para realização do *login*.

Fonte: Código implementado pelos autores.

Inicialmente é necessário definir o endereço eletrônico (Linha 03) da página de *login* que será acessada pelo `WebDriver` e em seguida preencher os campos de *e-mail* e senha presentes na tela.

Existem diversas maneiras de localizar os campos de um formulário exibido na tela. No Código 6.7 todos os campos são localizados através do `xpath` (uma sintaxe ou linguagem para localizar qualquer elemento em uma página da *web* usando expressão de caminho XML). Com os caminhos XML é possível encontrar o campo que se procura e realizar a ação desejada, onde o método `sendKeys` é utilizado para preencher os campos e o método `click` para clicar nos campos e botões da tela.

Por meio do `xpath`, localiza-se e preenche-se os campos *e-mail* e senha (Linhas 05-14) e localiza-se e clica-se no botão "Entrar" (Linhas 19-20). A cada execução dessas ações realizam-se intervalos, com o objetivo de respeitar o tempo de espera entre cada uma dessas ações realizadas pelo `WebDriver`. Esses intervalos são realizados utilizando o método `sleep` da classe `Thread`, passando como parâmetro o tempo de intervalo desejado em milissegundos (Linhas 09, 15 e 21).

Com WebDriver conectado e configurado e o *login* já efetuado, o Código 6.8 apresenta como o método para registro de uma nova reserva foi implementado. O teste inicia criando-se a variável `amountReservations`, que recebe o número de reservas já cadastradas (Linhas 04 e 05).

```
1. @Test
2. public void registerANewReservation()
3. throws InterruptedException {
4.     int amountReservations = driver.findElement(
5.     By.xpath("//*[@id=\"root\"]/div/div/div/ul/li")).size();
6.
7.     driver.findElement(
8.     By.xpath("//*[@id=\"root\"]/div/div/div/a")).click();
9.     Thread.sleep(3000);
10.
11.    //preenchimento dos campos omitido
12.
13.    driver.findElement(
14.       By.xpath("//*[@id=\"root\"]/div/div/div/form/button"))
15.    .click();
16.    Thread.sleep(3000);
17.
18.   driver.findElement(
19.       By.xpath("//*[@id=\"root\"]/div/div/div/a")).click();
20.    Thread.sleep(3000);
21.
22.     //seleção da forma de pagamento omitida
23.
24.    driver.findElement(
25.       By.xpath("//*[@id=\"root\"]/div/div/div/form/button"))
26.    .click();
27.     Thread.sleep(3000);
28.
29.     assertEquals(
30.         driver.findElement(
31.               By.xpath("//*[@id=\"root\"]/div/div/div/ul/li"))
32.               .size(), amountReservations + 1);
33.
34.     driver.close();
35. }
```

Código 6.8 - Método de teste de registrar uma nova reserva.

Fonte: Código implementado pelos autores.

Em seguida, o WebDriver localiza o botão de realizar uma nova reserva e realiza um clique nele (Linhas 07 e 08), aguarda-se 3 segundos para página renderizar (Linha 09) e preenche os campos *check-in*, *check-out*, número de hóspedes e preço máximo da diária para buscar os quartos (omitidos no código 6.8, conforme descrito na Linha 11).

Após preencher os campos desejados, localiza o botão "Selecionar Quartos" e realiza o clique nele (Linhas 13 a 15) e redireciona-se para a próxima tela. Após aguardar 3 segundos (Linha 16), com a tela de selecionar os quartos já renderizada, selecionam-se os quartos e em seguida localiza-se e realiza-se o clique no botão "Selecionar forma de pagamento" (Linhas 19 e 19).

Feito isso, aguarda-se 3 segundos (Linha 20) para que a página de "Selecionar forma de pagamento" seja renderizada com o valor da reserva já calculado, em que o cálculo é feito multiplicando o valor da(s) diária(s) pela quantidade de dias reservados. Então, o `WebDriver` seleciona a forma de pagamento em dinheiro (não exibida no Código 6.8) e em seguida localiza e realiza o clique no botão de "Cadastrar reserva" (Linhas 24-26).

Feito isso, redireciona-se para a tela de Perfil do hóspede e verifica-se se há um número de reservas maior do que havia anteriormente (Linhas 29-32), que é a condição deste cenário para indicar se o teste obteve sucesso ou não. Ao final da execução o WebDriver fecha a página do navegador *web* (Linha 34).

7 DISCUSSÕES

Este capítulo apresenta discussões referentes às reflexões geradas durante o desenvolvimento deste trabalho.

Inicialmente, pretendíamos usar como estudo de caso nesta monografia o *software* denominado ForPDI (FORPDI, 2018): uma plataforma aberta para gestão e acompanhamento do Plano de Desenvolvimento Institucional (PDI) de universidades federais e outras instituições. Tínhamos o intuito de contribuir para a melhoria da qualidade do *software*, colocando em prática o que foi visto na revisão da literatura e nos trabalhos relacionados (o ForPDI não possui testes automatizados). Porém, com a falta da documentação do projeto do ForPDI e a falta do fornecimento de suporte por parte dos desenvolvedores, optamos por desenvolver nossa própria ferramenta de exemplo, o *HostelApp*, simultaneamente com a implementação dos testes automatizados, trazendo maior legibilidade, confiabilidade e facilidade de manutenção no *software*.

A fim de testar a viabilidade dos testes discutidos no Capítulo 5 para o *HostelApp*, utilizamos o *framework* JUnit, que fornece informações a respeito da cobertura de teste do código, evidenciando os casos de testes que foram testados e os que ainda não foram.

Contudo, é importante dizer que cobertura de código não significa o mesmo que cobertura de testes. A cobertura de código trata-se de uma métrica quantitativa que visa medir em porcentagem o quanto do *software* está sendo coberto/exercitado, quando executado um determinado conjunto de *casos de testes*. Por outro lado, a cobertura de testes refere-se a uma métrica qualitativa que visa medir a eficácia dos testes perante os requisitos testados, determinando se os *casos de testes* existentes cobrem os requisitos que estão sendo testados.

Dessa forma, ao invés de garantirmos a cobertura total de código de cada pacote do *HostelApp*, decidimos somente garantir a cobertura total dos pacotes em que existem regras de negócio, como os pacotes *"service"* e *"security"*. Pacotes em que não há regras de negócio foram desconsiderados por considerarmos que possuem implementações triviais, como por exemplo o que contém a método `main`, responsável pela execução do *HostelApp*.

Com relação ao mecanismo de autenticação usado no *HostelApp*, durante a implementação do *front-end*, refletimos sobre qual propriedade de armazenamento do navegador *web* seria usada para armazenar os dados de navegação do usuário: *localStorage* ou *sessionStorage*. Ao utilizar o *localStorage*, os dados de navegação armazenados (*token* de autenticidade do usuário por exemplo) perduram até mesmo quando a aba do navegador *web* é

encerrado, podendo acessar o endereço do *front-end* do *HostelApp* a partir de várias abas e de várias instâncias do navegador. Já ao utilizar o *sessionStorage,* os dados perduram somente enquanto o acesso é feito em uma aba específica do navegador *web*. Se o *hóspede* tentar acessar o *front-end* a partir de outra aba ou instância do navegador *web,* sua autenticação é perdida e o *hóspede* terá de realizar o *login* novamente. Para este trabalho optamos em utilizar o *sessionStorage,* uma vez que o *HostelApp* não necessita de um alto nível de escalonamento, em que é necessário manter milhares ou até milhões de hóspedes simultaneamente conectados.

Em relação aos *frameworks* utilizados para testar APIs RESTful no Spring Boot surgiram duas opções: *MockMvc* e *RestTemplate* (RESTTEMPLATE, 2014). No *MockMvc,* é pré-configurado um contexto de aplicação *web* que simula as requisições e respostas HTTP. Assim, é possível "fingir" requisições HTTP, ou seja, não há conexões de rede reais realizadas (conexões simuladas). Já com o *RestTemplate,* você precisa implantar uma instância real do servidor para atender às solicitações HTTP enviadas (conexões não simuladas), o que deixa a implementação mais custosa.

Para este trabalho optamos pela utilização do *MockMvc*, pois já vem integrado e pré-configurado com o *framework* Spring Boot. Além disso, acreditamos que o *MockMvc* é uma melhor opção por se basear em API fluida (Fluent API) (BUTTING *et al.*, 2018) que torna o código mais legível.

8 CONCLUSÃO E TRABALHOS FUTUROS

O desenvolvimento do *HostelApp* junto da automação de testes nos faz concluir que a automação de testes traz diversos benefícios no processo de desenvolvimento de *software*, como a melhoria na qualidade do código (no que tange se o *software* está de acordo com o requisitos), melhoria na confiabilidade (retorna um *feedback* rápido quando um erro surge na aplicação assim que implementa-se uma nova funcionalidade), facilidade na manutenção (com os testes é possível saber exatamente onde ocorreu o erro, evitando que o desenvolvedor passe horas analisando o código até encontrar a causa do erro), entre outros benefícios.

A implementação dos testes é um investimento cujo retorno se dá a longo prazo. Por conta disso, este trabalho descreve de forma a sugerir de que o uso de testes de *software* se faz necessário, uma vez que eventos futuros como alterações ou atualizações em uma aplicação podem comprometer o código como um todo, sendo possível evitar esse tipo de problema utilizando os testes.

A automação de testes tem como objetivo agregar valor ao *software*, e não deve ser empregada como um substituto do teste manual, devendo ser introduzida como uma técnica adicional. O foco deve ser na melhoria do processo de testes. Consequentemente, a necessidade de automatizar os testes acontece naturalmente, como resultado da evolução da maturidade do processo de testes.

Por outro lado, os testes de *software* não garantem que o sistema esteja livre de *bugs*. Os testes são feitos para prevenir os erros que são cobertos pela bateria de testes e não para garantir que o *software* está livre de erros. Além disso, não dá pra garantir que um *software* está livre de erros; uma das razões é porque os testes não conseguem cobrir a enorme quantidade de possibilidades de entrada e combinações de resultados a que um *software* pode estar sujeito.

Portanto, como trabalhos futuros nós sugerimos explorar outras técnicas de testes — como as discutidas no Capítulo 3 — que não foram utilizadas na implementação deste trabalho, para assim aumentar a cobertura do código e beneficiar o *software* aplicando outras técnicas de teste. Além disso, ainda sobre o *HostelApp*, também é possível a implementação de *testes de unidade* adicionais que envolvem reservas, assim como a implementação de *testes de integração* para cenários referentes a quartos e hóspedes, uma vez que estes não foram implementados.

REFERÊNCIAS

BARTIÉ, A. **Garantia de Qualidade de Software: adquirindo maturidade organizacional**. 1. ed. Rio de Janeiro: Editora Campus, 2002. ISBN 9788535211245.

BECK, K. **Test-driven development: by example**. 1. ed. Boston: Addison-Wesley Professional, 2003. ISBN 978-0321146533.

BECK, K.; ANDRES, C. **Extreme Programming Explained: Embrace Change**. 2. ed. Westford: Addison-Wesley, 2004. ISBN 9780321278654.

BERNARDO, P. C. **Padrões de testes automatizados**. São Paulo: Instituto de Matemática e Estatística, 2011. Disponível em: <https://www.teses.usp.br/teses/disponiveis/45/45134/tde-02042012-120707/en.php>. Acesso em: 08 Fev. 2021.

BUTTING, A.; DALIBOR, M.; LEONHARDT, G.; RUMPE, B.; WORTMANN, A. **Deriving fluent internal domain-specific languages from grammars**. Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering, 2018. Disponível em: https://martinfowler.com/articles/practical-test-pyramid.html. Acesso em: 02 Mar. 2021.

CHACON, S.; STRAUB, B. **Pro git**. 2. ed. Nova Iorque: Springer Nature, 2014. ISBN 9781484200773.

CRESPO, A. N; da SILVA, O. J; BORGES, C. A; SALVIANO, C. F; JUNIOR, M de T. e; JINO, M. **Uma Metodologia para Teste de Software no Contexto da Melhoria de Processo**, 2004. Disponível em: <https://www.researchgate.net/publication/237497188_Uma_Metodologia_para_Teste_de_Software_no_Contexto_da_Melhoria_de_Processo>. Acesso em: 02 Fev. 2021

COHN, M. **Succeeding with agile: software development using Scrum**. Boston: Addison-Wesley Professional, 2010. ISBN 978-0321579362.

COHN, M. **User Stories Applied: For Agile Software Development**. Crawfordsville: Addison-Wesley Professional, 2004. ISBN 0321205685.

COLLINS, E. F.; LOBÃO, L. M. A.; LUCENA JR, V. F.. **Experiência em Aplicação de Processo de Teste de Software Interativo e Automático.** São Paulo: Brazilian Workshop on Systematic and Automated Software Tests, 2011, v. 1. p. 1-6. ISBN 978-6139794881.

COPELAND, L. A. **Practitioner's Guide to Software Test Design. Norwood**, MA, USA: Artech House, Inc., 2003. ISBN 158053791X.

DA SILVA, D. M.; SIQUEIRA, R. D. **Framework Para Automação de Testes**. 2013. 71f. Trabalho de Conclusão de Curso (Bacharel em Ciência da Computação) Universidade Federal de Alfenas, UNIFAL, Alfenas, 2013. Disponível em: https://www.bcc.unifal-mg.edu.br/biblioteca/producao-discente/monografias/monografias-201 2-2/monografia-douglas-miranda-da-silva-e-renan-domingues-siqueira/. Acesso em: 02 Jan. 2021.

DEMILLO, R. A.; LIPTON, R. J; SAYWARD, F. G. **Hints on Test Data Selection: Help for the Practicing Programmer**. Naples PlazaLong Beach, CA: The Institute of Electrical and Electronics Engineers, Inc, 1978. Disponível em: <https://edisciplinas.usp.br/pluginfile.php/1943431/mod_resource/content/1/Hints_on_Test_Data_Selection-Demillo.pdf>. Acesso em: 12 Agos. 2021.

DESIKAN, S. **Software Testing: Principles and Practices**. Harlow: Addison-Wesley Professional, 2005. ISBN 9788177581218.

EVANS, E. **Domain-Driven Design: Tackling Complexity in the Heart of Software**. 2003. ISBN 9780321125217.

FORPDI, 2018. Disponível em <http://www.forplad.andifes.org.br/?q=forpdi>. Acesso em: 03 Ago. 2021.

FOWLER, M. **Patterns of Enterprise Application Architecture**. Harlow: Addison-Wesley Professional, 2003. ISBN 9780321127426.

FOWLER, M. **The Practical Test Pyramid** [S. l.]: Fowler, 2018. Disponível em: https://martinfowler.com/articles/practical-test-pyramid.html. Acesso em: 02 Jan. 2021.

GRAHAM, D; FEWSTER, M. **Experiences of test automation: case studies of software test automation**. 1. ed. Crawfordsville: Addison-Wesley Professional, 2012. ISBN 978-0321754066.

GRAHAM, D.; VAN VEENENDALLl, E.; EVANS, I. **Foundations of Software Testing.** 1. ed. Connecticut: Cengage Learning. Cengage Learning Business Press, 2008. pp. 57–58. ISBN 9781844809899.

GREGORY, J.; CRISPIN, L. **More Agile Testing.** 1. ed. Crawfordsville: Addison-Wesley Professional, 2014. pp. 23–39. ISBN 9780133749564.

GUEDES, G. T. A. **UML 2 - Uma Abordagem Prática**. 3. ed. São Paulo, SP, Brasil: Novatec Editora Ltda, 2018. ISBN 9788575226445.

GUPTA, V.; SAXENA, V. S. **Software Testing: Smoke and Sanity- International Journal of Engineering Research & Technology**. Disponível em: <https://www.ijert.org/software-testing-smoke-and-sanity>. Acesso em: 20 Mar. 2021.

HALL, M.; BROWN, L.. **Core Servlets and Java Server Pages.** 2. ed. Harlow: Addison-Wesley Professional, 2003. Vol 1. ISBN 9780130092298.

HENDRICKSON, E. **Explore it!: reduce risk and increase confidence with exploratory testing**. Pragmatic Bookshelf, 2013. ISBN 9781937785024.

HOODA, I. **International Journal of Computer Applications**, 2015. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.695.1299&rep=rep1&type=pdf>. Acesso em: 21 Jan. 2021.

IEEE. **IEEE Standard Glossary of Software Engineering Terminology**. IEEE Computer Society, 1990. p. 1-84. ISBN 9789994749805.

JAVA, 1996. Disponível em: <https://www.java.com/>. Acesso em 21 Jul. 2021.

JAVASCRIPT, 1995. Disponível em: <https://www.javascript.com/>. Acesso em: 21 Jul. 2021.

JUNIT, 1998, Disponível em: <https://junit.org/>. Acesso em: 07 Fev. 2021.

KANER, C.; FALK, J.; NGUYEN, H. Q. **Testing Computer Software**, 2. ed. Nova Iorque: John Wiley and Sons, Inc, 1999. ISBN 978-0471-35846-6.

KACZANOWSKI, T. **Practical Unit Testing with JUnit and Mockito**. Polônia: publicado por Tomas Kaczanowskii, 2013. ISBN 9788393489398.

KOLAWA, A.; HUIZINGA, D. **Automated Defect Prevention: Best Practices in Software Management**. Wiley-IEEE Computer Society Press, 2007. ISBN 978-0-470-04212-0.

KUMAR, D.; MISHRA, K. K. **The Impacts of Test Automation on Software's Cost, Quality and Time to Market**. Allahabad: Instituto de Tecnologia de Allahabad, 2016. Disponível em: <https://www.sciencedirect.com/science/article/pii/S1877050916001277>. Acesso em: 09 Mar. 2020.

LARMAN, C. **Applying UML and patterns: an introduction to object oriented analysis and design and iterative development**. 3. ed. Crawfordsville: Addison-Wesley Professional, 2004. ISBN 9780131489066.

LEE, D.; NETRAVALI, A. N.; SABNANI, K. K.; SUGLA, B.; JOHN, A. **Passive testing and applications to network management**. Atlanta: International Conference on Network Protocols, 1997. ISBN 081868061X.

LUO, L. **Software Testing Techniques**. Pitsburgo: School of Computer Science Carnegie Mellon University, 2001. Disponível em: https://tero.pw/vi.pdf. Acesso em: 14 Jan. 2021.

MACHEK, Z. **PHPUnit Essentials**. Packt Publishing Ltd, 2014. ISBN 9781783283439.

MALDONADO, J. C.; DELAMARO, M. E.; JINO, M. **Introdução ao Teste de Software**. 1. ed. São Paulo: Elsevier Editora Ltda, 2007. ISBN 9788535267495.

MENEZES, P. M. **Segurança em redes de computadores uma visão sobre o processo de Pentest**. Universidade Federal do Sergipe, UFS, Sergipe, 2015. Disponível em: https://doi.org/10.17564/2359-4942.2015v1n2p85-96. Acesso em: 20 Fev. 2021.

MESZAROS, G. **Xunit Test Patterns, Refactoring Test Code**. 1. ed. Boston: Addison-Wesley Professional, 2007. ISBN 978-0131495050.

MOCKITO, 2008. Disponível em: https://site.mockito.org/>. Acesso em: Jul. 2021.

MOCKMVC, 2014. Disponível em: <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/test/web/servlet/MockMvc.html>. Acesso em: 21 Jul. 2021.

MOHD, C. K. N. C. K.; SHAHBODIN, F. **Procedia-Social and Behavioral Sciences**, 2015 - Elsevier. Disponível em: <https://www.sciencedirect.com/science/article/pii/S1877042815037982>. Acesso em: 23 Jun. 2021.

MORENO, D. **Introdução ao PENTEST.** 2. ed. Editora Novatec. 2019. ISBN 978-8575224311.

MUGRIDGE, R.; CUNNINGHAM, W. **Fit for developing software: framework for integrated tests**. 1. ed. Boston: Addison-Wesley Professional, 2005. ISBN 9780321269348.

MYERS, G. J.; SANDLER, C.; BADGETT, T.; THOMAS, T. M. **The Art of Software Testing**, 2. ed. Hoboken: New Jersey, John Wiley & Sons, Inc, 2004, 8p. ISBN 9780471469124.

MYSQL, 1995. Disponível em: <https://www.mysql.com>. Acesso em: 21 Jul. 2020.

NATIONAL RESEARCH COUNCIL. **Aging Avionics in Military Aircraft**. Washington, DC, USA: The National Academies Press, 2001. ISBN 9780309074490.

OBERKAMPF, W. L.; ROY, C. J. **Verification and Validation in Scientific Computing.** Cambridge: Cambridge University Press, 2010. pp. 154–5. ISBN 9781139491761.

PINHEIRO, S. A. M. **Estudo e implementação de testes de software em desenvolvimento ágil**, 2015. Disponível em: <https://repositorium.sdum.uminho.pt/handle/1822/40173>. Acesso em: 12 Fev. 2021.

PRESSMAN, R. S. **Engenharia de Software: Uma Abordagem Profissional**. 8. ed. AMGH. São Paulo, 2016. ISBN 9788580555332.

PUGH, K. **Lean-agile acceptance test-driven development: better software through collaboration**. Boston: Addison-Wesley Professional, 2010. ISBN 9780321714084.

RAFI, D.; MOSES, K.; KAI, P.; MÄNTYLÄ, M.. **Benefits and limitations of automated software testing: Systematic literature review and practitioner survey**. Suécia: 7th International Workshop on Automation of Software Test, AST 2012 - Proceedings. 36-42. 10.1109/IWAST.2012.6228988.

REACT, 2013. Disponível em: <https://reactjs.org/>. Acesso em 21 Jul. 2021.

RESTTEMPLATE, 2014. Disponível em <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/client/RestTemplate.html>. Acesso em: 03 Ago. 2021.

ROMAN, P. **Agile Product Management with Scrum: Creating Products that Customers Love**. 1. ed. Boston: Addison-Wesley Professional, 2010. ISBN 9780321605788.

RUNESON, P.; WOHLIN, C. **Statistical Usage Testing for Software Reliability Control**. Londres: Department of Electrical and Information Technology, 1995. Vol. 19, No. 2, pp. 195–207.

SELENIUM, 2004, Disponível em: <https://www.selenium.dev/>. Acesso em: 02 Fev. 2020.

SILVA, E. F. I.. **Testes de Internacionalização e Localização em Sistemas de Software: Levantamento do Estado da Arte**. Trabalho de Conclusão de Curso (Bacharel em Ciência da Computação) Universidade Federal de Pernambuco, UFPE, Recife, 2019. Disponível em: https://www.cin.ufpe.br/~tg/2019-2/TG_CC/tg_efis.pdf. Acesso em: 19 Fev. 2021.

SINGH S. K.; SINGH A. **Software testing**. Vandana Publications Lucknow. 2012. ISBN 978-81-941110-6-1.

SMART, J. F. **BDD in Action: Behavior-Driven Development for the Whole Software Lifecycle**. 1. ed. New York: Manning Publications, 2014. Vol. 12. ISBN 9781617291654.

SOMMERVILLE, I. **Software Engineering**. Harlow: Addison-Wesley Professional Limited Edinburgh Gate, 2016. ISBN 9780133943030.

SPRING BOOT, 2014. Disponível em: <https://spring.io/projects/spring-boot/>. Acesso em 21 Jul. 2021.

SPRING INITIALIZR, 2014. Disponível em: <https://docs.spring.io/initializr/docs/current/reference/html/#initializr-documentation/>. Acesso em: 21 Jul. 2021.

SUTHERLAND, J. **The art of doing twice the work in half the time**. Crown Business, 2015. ISBN 9788544100882.

TAHCHIEV, P.; LEME, F.; MASSOL, Vincent; GREGORY, Gary. **JUnit in action**. 2. ed. Manning Publications Co., 2010. ISBN 9781935182023.

TEIXEIRA, F. **Introdução e boas práticas em UX Design**. 1. ed. Casa do Código, 2014. ISBN 9788566250480.

WHITTAKER, J.A. **What is software testing? And why is it so hard?**. IEEE Software, 2000. vol. 17. Disponível em: https://ieeexplore.ieee.org/abstract/document/819971. Acesso em: 14 Jan. 2021.

APÊNDICES

APÊNDICE A - CASOS DE TESTE DE INTEGRAÇÃO PARA OS REQUISITOS LISTAR E EXCLUIR RESERVAS

Os Quadros A.1, A.2 e A.3 apresentam casos de *testes de integração* que objetivam assegurar a correta listagem de reservas do *HostelApp*. Conforme pode ser visto nos quadros, em cada teste é feita uma requisição HTTP GET utilizando o endereço eletrônico descrito na coluna *Requisição*. Dentre os testes, há aqueles sem a passagem de parâmetros e aqueles com a passagem de parâmetros (quando é necessário filtrar as reservas que serão retornadas pelo *HostelApp*). Dois objetos de Reserva são previamente criados para facilitar a realização dos testes, com o objetivo de utilizar os valores presentes em seus atributos como parâmetro ao filtrar as reservas, além da utilização na validação dos resultados.

Estes testes têm o intuito de verificar se os resultados retornados pela requisição estão de acordo com os resultados esperados, presentes na coluna *Resultados esperados* em cada quadro.

Quadro A.1 - *Caso de teste 9*: Listagem de todas as reservas feitas no *HostelApp*.

Descrição	**Requisição**	**Resultados esperados**
Neste caso de teste é feita uma requisição sem a passagem de parâmetros, com o objetivo de listar todas as reservas presentes no banco de dados.	GET em `/api/reservations`	Supondo a existência de 2 reservas cadastradas no banco de dados, o número de reservas retornadas deverá ser igual a 2.

Quadro A.2 - *Caso de teste* 10: Listagem de reservas filtrando pelo nome do hóspede.

Descrição	**Requisição**	**Resultados esperados**
Neste caso de teste é feita uma requisição com a passagem do parâmetro "*guestId*", com o objetivo de listar todas as reservas do hóspede que possua o valor do atributo "*guestId*" igual ao valor do parâmetro da requisição.	GET em `/api/reservations?guestId=1`	Supondo que exista apenas um hóspede cadastrado no banco de dados com o `id` igual a 1 e com duas reservas associadas: - O número de reservas retornado deverá ser igual à 2; - O nome do hóspede para essas duas reservas deverá ser igual ao nome do hóspede que possui o *id* informado na requisição.

Quadro A.3 - *Caso de teste* 11: Listagem de reservas filtrando pelo *id* da reserva.

Descrição	**Requisição**	**Resultados esperados**
Neste caso de teste é feita uma requisição com a passagem do parâmetro "*id*", com o objetivo de de retornar a reserva associada a este *id*.	GET em `/api/reservations/1`	Supondo que exista uma reserva cadastrada no banco de dados com o *id* 1: - A data de *check-in* do objeto retornado deve ser igual à data de *check-in* da reserva cadastrada no banco de dados; - A data de *check-out* do objeto retornado deve ser igual à data de *check-out* da reserva cadastrada no banco de dados.

Os Quadros A.4 e A.5 apresentam casos de teste que objetivam assegurar a correta exclusão de reservas no *HostelApp*. Conforme pode ser visto nos quadros, em cada *caso de teste* é feita uma requisição HTTP DELETE utilizando o endereço eletrônico descrito na coluna *Requisição*, passando como parâmetro da requisição o ID da reserva a ser excluída.

Estes testes têm o intuito de verificar se os resultados retornados pela requisição estão de acordo com os resultados esperados, presentes na coluna *Resultados esperados* em cada quadro.

Quadro A.4 - *Caso de teste 12*: Exclusão de reserva informando um *id* válido.

Descrição	**Requisição**	**Resultado esperado**
Neste teste é feita a exclusão de uma reserva, passando como parâmetro junto à URL um *id* existente no banco de dados.	DELETE em `/api/reservations/1`	Status de resposta HTTP 200 (OK) indicando que a requisição obteve sucesso e o objeto foi excluído.

Quadro A.5 - *Caso de teste 13*: Exclusão de reserva informando um *id* inválido.

Descrição	**Requisição**	**Resultado esperado**
Neste teste é feita a tentativa de exclusão de uma reserva, passando como parâmetro junto à URL um *id* inexistente no banco de dados, esperando-se obter falha na requisição. Utiliza-se o valor 0 como valor do parâmetro *id*, pois não há nenhuma reserva previamente cadastrada no banco de dados que possua o valor do *id* igual a 0.	DELETE em `/api/reservations/0`	Status de resposta HTTP 404 (Not found) indicando que a requisição falhou, pois o objeto não foi encontrado.

APÊNDICE B - IMPLEMENTAÇÃO DOS CASOS DE TESTE PARA OS REQUISITOS LISTAR E EXCLUIR RESERVAS

Os Códigos B.1 e B.2 apresentam trechos de código presentes na classe `ListReservationsTest`, contendo a implementação necessária para que seja possível testar o requisito de listar reservas (RF-02 no Quadro 4.2 e *casos de teste* descritos no Apêndice A).

Inicialmente no método `init`, denotado pela anotação `@BeforeAll`, define-se a URL (*endpoint*) na qual são realizadas as requisições dos testes (Linha 05). Feito isso, realiza-se a chamada do método `initialize` presente na classe `ReservationInitializer` (Linha 07 e 08) onde é realizado o *login* — pois é necessária a autenticação do usuário — e definição dos atributos dos objetos utilizados nos testes.

Feito isso, armazena-se no banco de dados o objeto de pagamento (Linha 10) e o objeto de reserva (linhas 12-14), que também é adicionado a uma lista de reservas (Linha 16). Realizados os armazenamentos e utilizando os mesmos dados da primeira reserva, alterando apenas a data de *check-in* e *check-out* (Linhas 18 e 20) e o quarto atribuído (Linhas 23 e 24), armazena-se uma segunda reserva que também será utilizada nos testes (Linhas 26-29), além de também adicioná-la à lista de reservas (Linhas 31).

Com as reservas definidas, atribui-se ao hóspede pré-armazenado na base de dados de testes a lista de reservas (Linhas 33-35) e salva-o na base de dados (Linha 39), a fim de que seja possível realizar os testes utilizando todas as informações descritas neste parágrafo.

O método `shouldReturnAllReservationsByGuestId`, denotado pela anotação `@Test` (Linha 00 do Código B.2) — que indica ao JUnit o método de teste a ser executado —, inicia utilizando-se o `MockMvc` (Linhas 05-10) para realizar a requisição HTTP GET para a URL mencionada na Linha 04 do Código B.1. Em seguida, passa-se como parâmetro o atributo "*guestId*" com o valor do *id* do hóspede que deseja-se listar as reservas (Linha 07), além de passar no *header* (Linha 08) o *token* de autenticação obtido no Código B.1.

Feito isso, armazena-se os dados obtidos na resposta da requisição em objetos do tipo `String` e do tipo `Array` de `ReservationDto` (Linhas 13-18), pois serão utilizados como parâmetro de validação para os métodos do tipo `assertEqual` do JUnit, a fim de verificar se o que foi retornado pelo servidor contém os mesmos atributos do objeto previamente definido no método `init` do Código B.1 descrito anteriormente.

```
1. @BeforeAll
2. public static void init(params...) throws
3. JsonProcessingException, Exception {
4.
5.    uri = new URI("/api/reservations/");
6.
7.    ReservationInitializer.initialize(headers, reservationForm,
8.        checkPayment, rooms_ID, mockMvc, objectMapper);
9.
10.   paymentsRepository.save(reservationForm.getPayment());
11.
12.   reservation1 =
13.   reservationRepository.save(reservationForm.returnReservation(
14.      paymentsRepository, roomRepository));
15.
16.   reservationsList.add(reservation1);
17.
18.   reservationForm.setCheckinDate(LocalDate.of(2021, 05, 01));
19.
20.   reservationForm.setCheckoutDate(
21.      LocalDate.of(2021, 05, 24. 04));
22.
23.   rooms_ID.remove(2L);
24.   rooms_ID.add(3L);
25.
26.   reservation2 =
27.   reservationRepository
28.      .save(reservationForm
29.      .returnReservation(paymentsRepository, roomRepository));
30.
31.   reservationsList.add(reservation2);
32.
33.   guest =
34.   guestRepository
35.       .findById(reservationForm.getGuest_ID()).get();
36.
37.   guest.setReservations(reservationsList);
38.
39.   guestRepository.save(guest);
41. }
```

Código B.1 - Método de configurações iniciais necessárias para os testes de listagem de reservas.

Fonte: Código implementado pelos autores.

O primeiro `assertEquals` (Linhas 20 e 21) verifica se o número de reservas retornado é igual ao número de reservas previamente cadastradas no método `init` do Código B.1.

O segundo e terceiro `assertEquals` (Linhas 23-27) verificam se os nomes armazenados na primeira e segunda reservas retornadas são os mesmos dos hóspedes a quem foram atribuídas as reservas no método `init` do Código B.1.

O Código B.3 apresenta o trecho de código presente na classe `DeleteReservationsTest`, correspondente à implementação do *caso de teste* descrito no Quadro A.4 presente no Apêndice A, o qual objetiva assegurar a correta remoção de reservas no *HostelApp*. O método `init` denotado por `@BeforeEach` desta seção é o mesmo do Código 6.4 presente na Seção 6.2.

```
1. @Test
2. public void shouldReturnAllReservationsByGuestId() throws
3. Exception {
4.
5.    MvcResult result =
6.         mockMvc.perform(get(uri)
7.             .param("guestId", guest.getId().toString())
8.            .headers(headers))
9.            .andDo(print())
10.           .andReturn();
11.
12.
13.   String contentAsString =
14.   result.getResponse().getContentAsString();
15.
16.   ReservationDto[] reservationObjResponse =
17.   objectMapper.readValue(contentAsString,
18.   ReservationDto[].class);
19.
20.   assertEquals(reservationsList.size(),
21.   reservationObjResponse.length);
22.
23.   assertEquals(reservation1.getGuestName(),
24.   guest.getName());
25.
26.   assertEquals(reservation2.getGuestName(),
27.   guest.getName());
28. }
```

Código B.2 - Método de teste para listar todas as reservas do hóspede utilizando seu nome como parâmetro.

Fonte: Código implementado pelos autores.

```
1. @Test
2. public void
3. shouldAutenticateAndDeleteOneReservationWithId1() throws
4. Exception {
5.
6.   paymentsRepository.save(reservationForm.getPayment());
7.
8.   reservationRepository.save(
9.   reservationForm.returnReservation(
10.     paymentsRepository, roomRepository));
11.
12.  mockMvc
13.     .perform(delete(uri + "1")
14.      .headers(headers))
15.      .andDo(print())
16.      .andExpect(status().isOk());
17. }
```

Código B.3 - Método de teste para excluir uma reserva pelo *id.*

Fonte: Quadro criado pelos autores.

Realizado o *login* e definidos os atributos da reserva, primeiramente são armazenados no banco de dados de teste o objeto de `pagamento` retornado pelo objeto `reservationForm` (Linha 06) e o objeto de `reserva` também retornado pelo objeto `reservationForm` (Linhas 08-10) usado para testar a exclusão de reservas.

Feito isso, utiliza-se o `MockMvc` para realizar a requisição HTTP DELETE (Linhas 13-17) para a URL mencionada na Linha 03 do Código 6.4 presente na Seção 6.2, passando 1 como valor do *id* da reserva a ser excluída. Além disso, envia-se no cabeçalho HTTP (Linha 15) o *token* de autenticação obtido método de *login* (pois é necessária a autenticação do usuário para realizar a exclusão), esperando um código de *status* HTTP 200 - "*OK*" - como resposta da requisição (Linha 17), indicando que a requisição obteve êxito e que o objeto foi removido.

Em requisições HTTP DELETE não é retornado nada no corpo da resposta pelo servidor, obtendo apenas um *status* HTTP 200 - "*OK"* como descrito no Quadro A.4 presente do Apêndice A, indicando que a exclusão foi bem sucedida.

APÊNDICE C - *LINK* DO REPOSITÓRIO DO PROJETO NO GITHUB

O código do *HostelApp* está disponível no repositório presente no GitHub (Figura C.1), que pode ser acessado através do endereço eletrônico https://github.com/pagliaresbcc/Hostel.

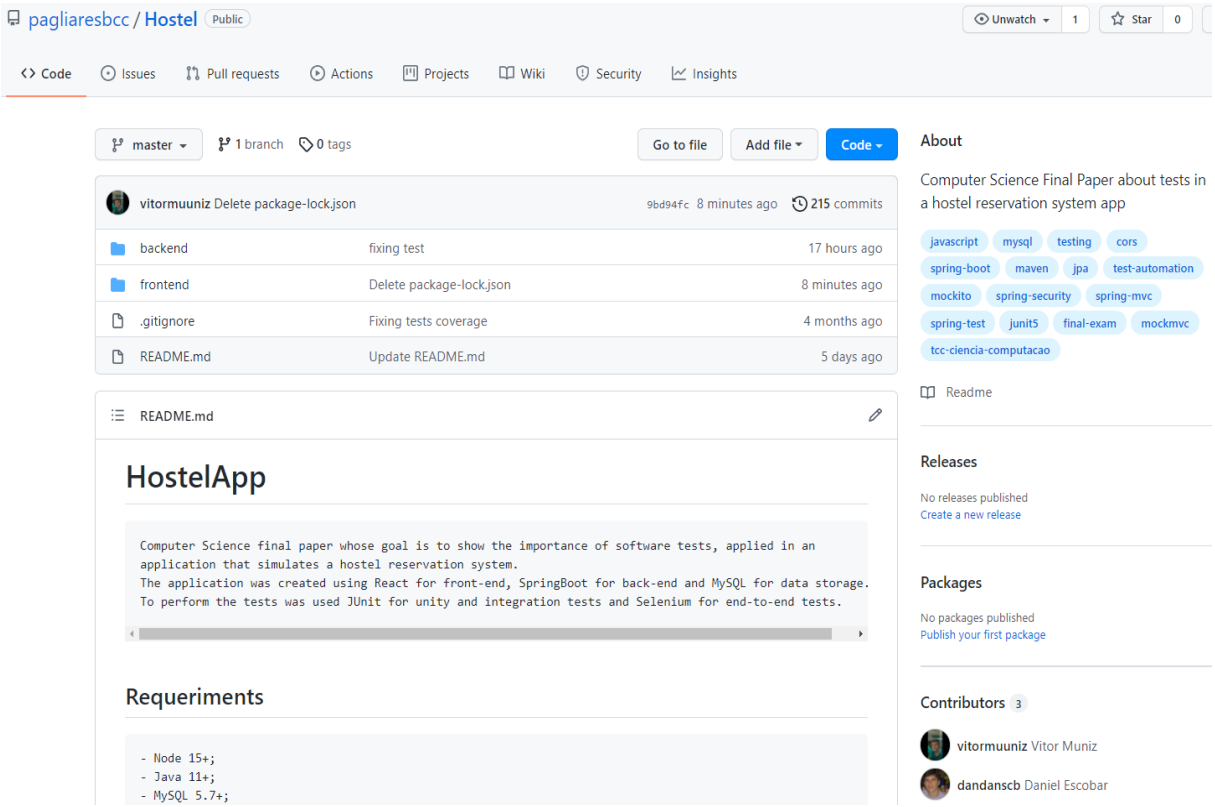

Figura C.1 - Página principal do repositório com o código fonte e testes automatizados do *HostelApp*.

Fonte: Figura criada pelos autores.