Padrões de testes automatizados

Paulo Cheque Bernardo

DISSERTAÇÃO APRESENTADA
AO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA
DA
UNIVERSIDADE DE SÃO PAULO
PARA
OBTENÇÃO DO TÍTULO
DE
MESTRE EM CIÊNCIA DA COMPUTAÇÃO

Programa: Ciência da Computação
Orientador: Prof. Dr. Fabio Kon

Durante o desenvolvimento desta pesquisa, o autor recebeu apoio do Projeto Qualipso financiado pela *European Commission*.

São Paulo, junho de 2011

Padrões de testes automatizados

Esta tese/dissertação contém as correções e alterações sugeridas pela Comissão Julgadora durante a defesa realizada por Paulo Cheque Bernardo em 04/07/2011. O original encontra-se disponível no Instituto de Matemática e Estatística da Universidade de São Paulo.

Comissão Julgadora:

- Prof. Dr. Fabio Kon – IME-USP (Orientador) - IME-USP
- Prof. Dr. Alfredo Goldman vel Lejbman – IME-USP
- Prof. Dr. Márcio Eduardo Delamaro – ICMC-USP

Agradecimentos

Agradeço especialmente à minha família: meus pais Carlos e Gersi e meus irmãos Pedro, Carlos e Sérgio, além de meu tio Nairson, não só por serem os principais responsáveis por tornarem possível a minha trajetória acadêmica, como também por terem me ensinado os valores de integridade e caráter.

Também agradeço ao meu professor e orientador Fabio Kon, por toda confiança que depositou em mim desde a graduação, pelas diversas oportunidades profissionais e acadêmicas proporcionadas, pela paciência e pelo incentivo durante os momentos difíceis, além de ser um grande exemplo de competência e disciplina.

Bons amigos da graduação e mestrado também são responsáveis diretos por este trabalho, devido à troca de conhecimento e, principalmente, por todo o apoio e incentivo. Por isso, agradeço a Alexandre Onishi, Álvaro Miyazawa, Adalberto Kishi, Ana Paula Mota, Camila Pacheco, André Guerra, Beraldo Leal, Celso Shimabukuro, Cristina Fang, Dairton Bassi, Daniel Cordeiro, Eduardo Katayama, Erich Machado, Flávio Mori, Gustavo Duarte, Mario Torres, Paulo Meirelles, Raphel Cobe, Ricardo Lazaro e Ricardo Yamamoto.

Durante o mestrado trabalhei em órgãos públicos, empresas e cooperativas e em todos esses locais adquiri conhecimento e experiência que foram fundamentais para o desenvolvimento deste trabalho. Por isso, agradeço a todos que me ajudaram das instituições AgilCoop, Assembléia Legislativa do Estado de São Paulo, Pró-Reitoria de Pós-Graduação da Universidade de São Paulo, UOL e Nokia Siemens Networks. Em especial, também agradeço ao projeto QualiPSo pelos mesmos motivos e também pelo incentivo à minha pesquisa.

Padrões e Antipadrões de Testes de Unidade

Padrões

- (Testabilidade) Injeção de Dependência (*Dependency Injection*): Seção 6.4.1, Página 75
- (Testabilidade) Objeto Humilde (*Humble Object*): Seção 6.4.2, Página 81
- (Testabilidade) Objeto Tolo (*Dummy Object*): Seção 6.4.3, Página 84
- (Testabilidade) Objeto Stub (*Test Stub*): Seção 6.4.4, Página 86
- (Testabilidade) Objeto Falsificado (*Fake Object*): Seção 6.4.5, Página 87
- (Testabilidade) Objeto Emulado (*Mock Object*): Seção 6.4.6, Página 89
- (Testabilidade) Objeto Espião (*Test Spy*): Seção 6.4.7, Página 90
- (Organizacional/Robustez/Testabilidade) Objeto Protótipo: Seção 6.4.8, Página 95
- (Qualidade) Teste por Comparação de Algoritmos: Seção 6.4.9, Página 98
- (Qualidade) Teste por Probabilidade: Seção 6.4.10, Página 100
- (Qualidade) Verificar Inversibilidade: Seção 6.4.11, Página 103
- (Qualidade) Verificar Valores Limites: Seção 6.4.12, Página 106

Antipadrões

- (Organizacional) Gancho para os Testes (*Test Hook*): Seção 6.5.1, Página 111
- (Organizacional) Testes Encadeados (*Chained Tests*): Seção 6.5.2, Página 112

Padrões de Testes com Persistência de Dados

Padrões

- (Organizacional) Uma Instância de Banco de Dados por Linha de Execução: Seção 7.2.1, Página 118
- (Robustez) Geração Dinâmica de Dados: Seção 7.2.2, Página 119

Padrões e Antipadrões de Testes de Interface de Usuário

Padrões

Antipadrões

Lista de Ferramentas/Arcabouços/Sistemas

Ferramentas de Testes Criadas pelo Autor

- Django Dynamic Fixture: `code.google.com/p/django-dynamic-fixture`
- Python-QAssertions: `code.google.com/p/python-qassertions`
- Util4Testing: `sourceforge.net/projects/util4testing`
- Util4Selenium: `sourceforge.net/projects/util4selenium`

Outras Ferramentas Criadas pelo Autor

- Card Game Engine: `code.google.com/p/cardgameengine`

Sistemas Citados que o Autor Ajudou a Desenvolver

- Janus: `sistemas.usp.br/janus`
- GinLab: `ginlab.com`

Ferramentas de Testes Automatizados

- JUnit: `junit.org`
- TestNG: `testng.org`
- Hamcrest: `code.google.com/p/hamcrest`
- Mockito: `mockito.org`
- Python-Mockito: `code.google.com/p/mockito-python`
- EasyMock: `easymock.org`
- JMock: `jmock.org`
- Parallel-Junit: `https://parallel-junit.dev.java.net`
- JUnit-Max: `www.junitmax.com`
- CUnit: `cunit.sourceforge.net`
- Python UnitTest: `pyunit.sourceforge.net/pyunit.html`
- JSUnit: `jsunit.net`
- Jaml-Unit: `www.isr.uci.edu/~lopes/`
- BDoc: `code.google.com/p/bdoc`
- MockEJB: `mockejb.org`
- HTMLUnit: `htmlunit.sourceforge.net`
- JWebUnit: `jwebunit.sourceforge.net`
- Selenium-WebDriver: `seleniumhq.org`, `openqa.org`
- Selenium-Grid: `selenium-grid.seleniumhq.org`
- Selenium-IDE: `seleniumhq.org/projects/ide`
- Selenium-RC: `seleniumhq.org`

- Fest: `code.google.com/p/fest`
- Marathon: `java-source.net/open-source/testing-tools/marathon`
- Jemmy: `https://jemmy.dev.java.net`
- Fit: `fit.c2.com`
- Fitnesse: `fit.c2.com`
- Cucumber: `cukes.info`
- RSpec: `rspec.info`
- JDave: `www.jdave.org`
- JBehave: `jbehave.org`
- TestDox: `agiledox.sourceforge.net`
- Testability Explorer: `code.google.com/p/testability-explorer`
- Emma: `emma.sourceforge.net`
- Eclemma: `eclemma.org`
- Continuum: `continuum.apache.org`
- CruiseControl: `cruisecontrol.sourceforge.net`
- JMeter: `jakarta.apache.org/jmeter`

Outras Ferramentas

- Firefox: `www.mozilla.com`
- Django: `www.djangoproject.com`
- Grails: `grails.org`
- Lift: `liftweb.net`
- Rails: `rubyonrails.org`
- Maven: `maven.apache.org`
- Spring-JDBC: `static.springsource.org/spring/docs/2.0.x/reference/jdbc.html`
- Spring-Framework: `www.springsource.org`
- HyperSQL (HSQLdb): `hsqldb.org`
- SQLite: `www.sqlite.org`
- VirtualBox: `www.virtualbox.org`
- Windows Virtual PC: `www.microsoft.com/windows/virtual-pc`
- VMWare: `www.vmware.com`

Resumo

A qualidade dos sistemas de software é uma preocupação de todo bom projeto e muito tem se estudado para melhorar tanto a qualidade do produto final quanto do processo de desenvolvimento. Teste de Software é uma área de estudo que tem crescido significativamente nos últimos tempos, em especial a automação de testes que está cada vez mais em evidência devido à agilidade e qualidade que pode trazer para o desenvolvimento de sistemas de software. Os testes automatizados podem ser eficazes e de baixo custo de implementação e manutenção e funcionam como um bom mecanismo para controlar a qualidade de sistemas.

No entanto, pouco conhecimento sobre a área e erros comuns na escrita e manutenção dos testes podem trazer dificuldades adicionais aos projetos de software. Testes automatizados de baixa qualidade não contribuem efetivamente com o controle de qualidade dos sistemas e ainda demandam muito tempo do desenvolvimento.

Para evitar esses problemas, esta dissertação apresenta de forma crítica e sistemática as principais práticas, padrões e técnicas para guiar o processo da criação, manutenção e gerenciamento dos casos de testes automatizados. Inicialmente, são feitas comparações entre a automação de testes e outras práticas de controle e garantia de qualidade. Em seguida, são apresentados os problemas e soluções mais comuns durante a automação de testes, tais como questões relacionadas a tipos específicos de algoritmos, sistemas com persistência de dados, testes de interfaces de usuário e técnicas de desenvolvimento de software com testes automatizados. Para finalizar, a dissertação traz uma reflexão sobre o gerenciamento e a abordagem da automação de testes para tornar o processo mais produtivo e eficaz.

Palavras-chave: Testes Automatizados, TDD, XP, Métodos ágeis, Teste de Software

Abstract

The quality of software systems is a concern of every good project and much has been studied to improve the quality of the final product and process development. Software Testing is an increasing area, especially test automation, which is in evidence due to the speed and quality that it may bring to the development of software systems. Automated tests can be effective and can have a low cost of implementation and maintenance to ensure and control the quality of the systems.

However, little knowledge about the area and common errors in writing and maintaining tests may bring additional difficulties to the software projects. Low quality automated tests do not contribute effectively to quality control systems and still take a long time of development.

To avoid these problems, we present critically and systematically the core practices, standards and techniques to guide the process of creation, maintenance and management of automated test cases. Initially, comparisons are made between the test automation, other control practices, and quality assurance. Next, we present the most common problems and solutions for the automation of tests, such as issues related to specific types of algorithms, systems with data persistence, testing user interfaces and techniques for software development with automated tests. Finally, this essay reflects on the management and approach to test automation to make the process more productive and effective.

Keywords: Automated Tests, TDD, XP, Agile Methods, Software Testing

Prefácio

Esta dissertação de mestrado é organizada em onze capítulos divididos em três partes: Introdução e Conceitos (Parte I), Práticas, Padrões e Técnicas para Testes de Correção (Parte II) e Gerenciamento de Testes Automatizados (Parte III).

Em todo o decorrer das três partes da dissertação, muitas ferramentas e sistemas são utilizadas ou mencionadas para fortalecer as discussões. Por isso, nas páginas iniciais foi adicionada uma listagem de todos os programas citados, destacando quais tiveram participação do autor desta dissertação. O objetivo é informar, de forma prática e coesa, os respectivos endereços Web para referência.

A Parte I é composta de quatro capítulos introdutórios que abordam inicialmente o contexto e as motivações do estudo de testes automatizados. Posteriormente, é apresentada a nomenclatura dessa área de estudo e que será utilizada no decorrer do trabalho. Para finalizar, serão discutidas algumas recomendações básicas para quaisquer projetos que utilizem Testes Automatizados.

Já a Parte II é dedicada às informações técnicas que ajudam na implementação dos Testes Automatizados de Correção. Essa parte possui muitos exemplos de código-fonte e de ferramentas; as linguagens de programação utilizadas são Python, Java, Scala e C. Grande parte das informações apresentadas nessa parte estão na forma de padrões, que é uma maneira estruturada e coesa de apresentar soluções para problemas recorrentes. Para agilizar o estudo desses padrões por consulta, foi incluído nas páginas inicias uma listagem de todos os padrões e antipadrões citados, contendo os números da seções e das páginas correspondentes.

Por fim, a Parte III discute questões de gerenciamento de projetos que possuem Testes Automatizados. Primeiramente, são apresentadas as principais métricas relacionadas com Testes Automatizados. Por último, são resumidas as principais conclusões encontradas por esse estudo, além de novas pesquisas que podem ser realizadas como extensões do presente trabalho.

Sumário

Lista de Figuras

Parte I

Introdução e Conceitos

Capítulo 1

Introdução

Garantir a qualidade de sistemas de software é um grande desafio devido à alta complexidade dos produtos e às inúmeras dificuldades relacionadas ao processo de desenvolvimento, que envolve questões humanas, técnicas, burocráticas, de negócio e políticas. A falta de qualidade nos sistemas de software causa grandes prejuízos à economia mundial [107] e já foi responsável por grandes tragédias que custaram vidas humanas, mesmo com todo o esforço dedicado ao avanço das tecnologias e das metodologias de desenvolvimento. Idealmente, os sistemas de software devem não só fazer corretamente o que o cliente precisa, mas também fazê-lo com segurança e eficiência. Ainda, para que os sistemas sejam duráveis, é necessário que ele sejam flexíveis e de fácil manutenção.

A nossa experiência [35, 36] mostra que, salvo honrosas exceções, na indústria de software brasileira, essas características são muitas vezes asseguradas através de testes manuais do sistema após o término de módulos específicos ou até mesmo do sistema inteiro. Essa abordagem manual e em muitos casos *ad hoc* leva à ocorrência de muitos problemas, tais como erros de regressão, logo ela deveria ser evitada.

Esta dissertação se inspira na filosofia dos Métodos Ágeis de Desenvolvimento de Software [38] e em práticas recomendadas pela Programação eXtrema (XP) [17], com ênfase em Testes Automatizados, que é uma técnica voltada principalmente para a melhoria da qualidade dos sistemas de software. Ela também se baseia fortemente na teoria de Testes de Software [95, 54] para aplicar as recomendações dos testes manuais na automação dos testes.

Além da teoria dos testes automatizados, serão apresentados detalhes técnicos com exemplos de código-fonte e discussões sobre estratégias de automação de testes. Esses tópicos estão distribuídos em três partes: I - *Introdução e Conceitos*, II - *Práticas, Padrões e Técnicas para Testes de Correção* e III - *Gerenciamento de Testes Automatizados*.

A primeira parte traz discussões sobre testes automatizados e seus benefícios comparados com outras práticas de controle e garantia de qualidade, como análises formais e testes manuais. São apresentados argumentos para apoiar nossa tese de que testes automatizados é uma prática de desenvolvimento eficaz e de baixo custo que ajuda a aumentar a qualidade dos sistemas de software.

A segunda parte da dissertação destaca os principais padrões e antipadrões que tornam os testes automatizados bons e ruins, respectivamente. Ela possui um capítulo introdutório que discute o que é um teste de correção de qualidade e também capítulos especializados em testes de unidade, interface de usuário e persistência de dados. O capítulo de testes de unidade envolve discussões sobre como fazer testes em sistemas programados com reflexão, programação orientada a objetos e orientada a aspectos.

Na Parte III existem informações para ajudar equipes e gerentes a gerenciar a produtividade da criação e manutenção dos testes automatizados assim como a qualidade do produto final. Um dos capítulos apresenta as principais métricas de acompanhamento para testes automatizados, abordando estratégias de acompanhar o progresso dos testes em diferentes tipos de projetos.

Depois da discussão de todos esses tópicos, o leitor terá adquirido os conhecimentos fundamentais

de métodos ágeis e testes de software assim como um vasto conhecimento sobre a automação de testes para favorecer a escrita e manutenção produtiva de testes automatizados de qualidade. Tudo isso para alcançar com êxito o objetivo principal do desenvolvimento de software que é a criação de programas e sistemas de qualidade que atendam as necessidades da sociedade.

1.1 Objetivos

Este trabalho tem como objetivo principal ser um guia para estudo, criação e manutenção de testes automatizados de qualidade. A automação de testes é uma prática ágil, eficaz e de baixo custo para melhorar a qualidade dos sistemas de software, mas é necessário conhecimento, organização e experiência para evitar que antipadrões ou falhas no gerenciamento reduzam o custo-benefício dessa prática para o desenvolvimento de software.

A primeira parte da dissertação, *Introdução e Conceitos*, tem como objetivo introduzir o tema e definir os principais conceitos de testes automatizados, assim como relacionar essa prática a outras áreas de estudo como métodos ágeis, testes de software, controle de qualidade e software livre. Testes automatizados englobam conceitos de diversas comunidades de desenvolvedores e testadores, por isso a importância de identificar os aspectos mais pertinentes de cada grupo de estudo.

Já a segunda parte, *Práticas, Padrões e Técnicas para Testes de Correção*, tem como objetivo reunir os aspectos mais importantes para o desenvolvimento e manutenção de bons testes automatizados que verifiquem a correção dos sistemas de software. São detalhadas diferentes situações que exigem cuidados durante a escrita dos testes e também são apresentadas soluções para problemas rotineiros.

A parte *Gerenciamento de Testes Automatizados* tem como objetivo discutir a realização da prática de testes automatizados durante o processo de desenvolvimento para proporcionar a melhoria da qualidade do produto final. Desenvolvedores e gerentes podem seguir várias abordagens para administrar os testes automatizados dependendo do tipo de projeto.

Contudo, este trabalho não pretende ensinar ou detalhar ferramentas e arcabouços de testes automatizados, pois isso tornaria o texto obsoleto em curto espaço de tempo. Também não é objetivo deste trabalho comparar matematicamente ou através de experimentos controlados, testes automatizados com métodos formais e matemáticos, apesar de ser um estudo de grande interesse para trabalhos futuros.

1.2 Motivação

A qualidade de um sistema de software pode ser definida por diversos aspectos como sugere o padrão ISO 9126[1] e a literatura de qualidade de software [126, 45, 115], sendo que a mais básica e importante é a correção. Segundo o Instituto de Padrões e Tecnologias dos Estatos Unidos (*National Institute of Standards and Technology*, NIST), erros de software causaram em 2002 prejuízos de aproximadamente 59,5 bilhões de dólares à economia dos Estados Unidos [107]. Logo, podemos concluir que algo precisa ser melhorado no desenvolvimento de software, incluindo ferramentas, processos e capacitação dos desenvolvedores.

O processo predominante de desenvolvimento de software na indústria se baseia nos métodos de desenvolvimento derivados do modelo em cascata [118], que valorizam a busca por qualidade em fases bem definidas e no fim do processo de desenvolvimento. Esse modelo contraria a nova tendência de controlar a qualidade do software baseada na prevenção de erros [37]. Essa propenção já havia sido prevista em 1979 por Glenford J. Myers [105], quando escreveu o primeiro livro da área de testes de software, e foi comprovada por diversos casos de sucesso [6].

A automação de testes é uma prática útil para prevenir erros durante a implementação e a manutenção do sistema. Testes a priori e Desenvolvimento Dirigido por Testes (vide Seção 3.2) também são forte-

[1] ISO 9126 é uma norma para qualidade de software.

mente baseados na prevenção de erros já que os testes são criados antes mesmo da própria implementação do sistema, proporcionando uma alta cobertura de verificação.

No entanto, implementar e manter grandes e completas baterias de testes automatizados é uma tarefa complexa e sujeita a erros, por isso é necessário conhecimento e experiência para que elas sejam bem escritas para garantir de maneira efetiva a qualidade do sistema e baratear o desenvolvimento.

O tema Automação de Testes ainda é recente no Brasil e é de muito interesse por parte das empresas nacionais. Em 2007 a cooperativa de desenvolvimento ágil AgilCoop[2] realizou uma pesquisa na região sudeste (Figura 1.1) em empresas de diversos tamanhos e tempo de mercado (Figura 1.2) para compreender o interesse das empresas pelos métodos ágeis.

Região	Visitas realizadas
Grande São Paulo	10
Campinas/Hortolândia	4
São Carlos	3
Rio de Janeiro	5
Total	**22**

Figura 1.1: Pesquisa AgilCoop - Cidades onde foram realizadas as entrevistas.

Número de funcionários	Número de empresas
De 1 a 10	1
De 11 a 100	9
De 101 a 1000	7
Mais de 1000	5

Tempo no mercado	Número de empresas
Menos de 5 anos	2
De 5 a 10 anos	8
De 10 a 15 anos	6
De 15 a 20 anos	0
De 20 a 25 anos	3
De 25 a 30 anos	0
Mais de 30 anos	3

Figura 1.2: Pesquisa AgilCoop - Informações das empresas participantes.

Nesta pesquisa, a AgilCoop realizava um questionário oferecendo diversos cursos relacionados a métodos ágeis (Figura 1.3). A fim de obter uma ordenação dos cursos mais relevantes, foram feitas diversas análises estatísticas sobre as informações coletadas. Por fim, os critérios adotados foram o interesse pelo curso e seu número esperado de alunos, representados pelos eixos `Interesse Médio` e `Número de Alunos` da Figura 1.4, respectivamente. O resultado indicou, com um bom destaque, que o maior interesse das empresas era obter maior conhecimento sobre testes automatizados, representado pelo item 7 do gráfico.

Esta dissertação tem potencial para ajudar a sanar parte das dificuldades encontradas na automação de testes e, dessa maneira, tornar-se uma contribuição útil, tanto para novas pesquisas sobre o tema

[2] `agilcoop.org.br`

Nº curso	Curso
7	Desenvolvimento de Software de Qualidade através de Testes Automatizados
6	Práticas de Métodos Ágeis para o Dia-a-dia dos Programadores
1	Introdução a Métodos Ágeis de Desenvolvimento de Software
3	Liderança Ágil de Projetos de Software
16	Padrões de Projeto (Design Patterns) e Princípios de Orientação a Objetos
4	Gestão Ágil de Projetos com Scrum
5	Planejamento e Estimativas Ágeis em Projetos de TI
2	Desenvolvimento Ágil de Software para Gerentes e Administradores
15	Desenvolvimento Dirigido por Testes
8	Laboratório Prático de Desenvolvimento Ágil de Software

Figura 1.3: Pesquisa AgilCoop - Tabela de cursos oferecidos.

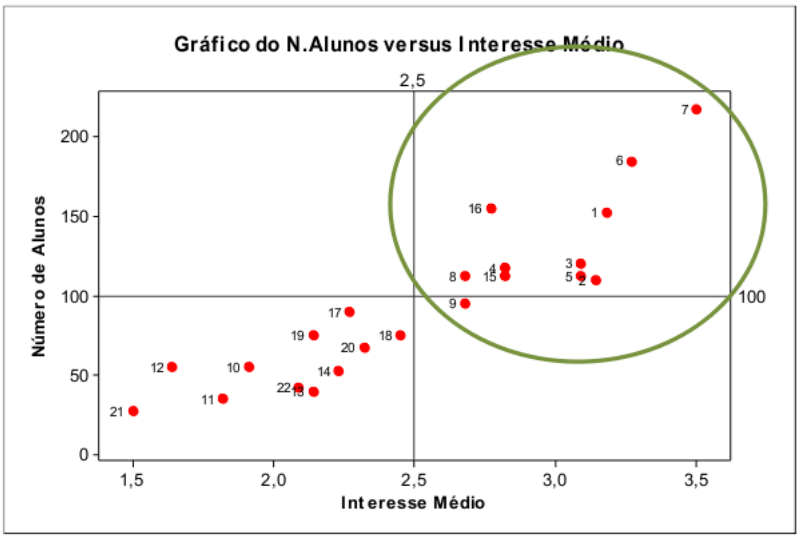


Figura 1.4: Pesquisa AgilCoop - Interesse das empresas pelos cursos.

quanto para a área empresarial. Ela utiliza como base os princípios das Metodologias Ágeis e reune alguns dos principais conceitos da área de Teste de Software, seja através de discussões ou da adaptação de alguns conceitos na forma de padrões. Além disso, são sugeridas novas soluções para problemas recorrentes que ainda não foram bem documentadas ou que foram pouco exploradas.

1.3 A quem se destina

A leitura desta dissertação é indicada principalmente para programadores, mas também pode ser útil para analistas de qualidade e gerentes de projetos. Ela aborda desde temas técnicos de computação até tópicos de engenharia de software.

Os programadores vão conhecer a área Teste de Software e suas principais técnicas que ajudam a melhorar a qualidade do sistema. Também irão encontrar as recomendações básicas sobre a automação de testes e soluções que ajudam a aperfeiçoar a escrita dos Testes Automatizados de Correção, seja através do uso de padrões e ou de outras técnicas.

Para os **programadores**, todos os capítulos do trabalho podem ser úteis. Especialmente para os pouco experientes em Testes Automatizados, é fundamental a leitura dos Capítulo 3 para facilitar o entendimento da dissertação e para ajudar no estudo de trabalho de outos autores, sejam artigos científicos, livros ou mesmo Internet. O Capítulo 4 também é interessante, pois apresenta uma visão geral do processo de automação de testes.

Caso o programador já tenha experiência com Testes Automatizados, ele pode estudar primeiramente os capítulos da Parte II, de acordo com suas necessidades. Ainda, o Capítulo 10 da Parte III possui discussões avançadas sobre implementação do código-fonte dos sistemas e dos testes.

Já os **analistas de qualidade** poderão conhecer novas técnicas de Testes de Software, além de conhecimentos básicos e avançados de Testes Automatizados. A Parte I é fundamental, principalmente devido às comparações feitas com os testes manuais no Capítulo 2 e às definições da terminologia utilizada pela dissertação no Capítulo 3.

A Parte II também é interessante de ser estudada por esses profissionais, em particular, os padrões do tipo Qualidade e o Capítulo 9, que explora técnicas de desenvolvimento com testes automatizados e que possui discussões sobre a interação da equipe de desenvolvimento com clientes e requisitos. A Parte III também possui discussões sobre Qualidade de Software, em especial, métricas interessantes de serem coletadas em sistemas que possuem testes automatizados.

Por fim, os **gerentes de projetos**, além de aumentar seus conhecimentos sobre testes automatizados e métodos ágeis, conhecerão técnicas de desenvolvimento que são recomendadas para analistas de qualidade e desenvolvedores. Ainda, irão se defrontar com recomendações referentes ao gerenciamento de software com automação de testes. Resumidamente, as Partes I e III são as mais interessantes para esses leitores, pois são mais abrangentes e menos técnicas.

1.4 Trabalhos Relacionados

A prática de automação de testes surgiu da integração de conceitos das áreas de orientação a objetos, testes de software e métodos ágeis de desenvolvimento [13, 16]. Por isso, todo o conhecimento destas áreas é pertinente e ajuda para um melhor aproveitamento do texto. A Figura 1.5 apresenta alguns dos principais autores de cada grupo de pesquisa para referência.

Testes Automatizados é uma técnica disseminada principalmente pela comunidade de métodos ágeis, por isso, conhecimento dos princípios ágeis ou de algum método ágil específico pode melhorar o aproveitamento da leitura deste trabalho. Dissertações anteriores em nosso grupo de pesquisa apresentam reflexões [46] e experiências [57] sobre o assunto, além de outros trabalhos que abordam práticas pontuais, como métricas de software para acompanhamento da evolução de projetos [128]. Em

Figura 1.5: Intersecção de áreas de estudos - alguns dos principais autores.

relação ao tema específico do trabalho, existem teses [53, 141] e livros [95, 99] que servem de referência e complementam o conhecimento de automação de testes.

Capítulo 2

Testes Automatizados

Testes automatizados (em oposição aos testes manuais) é a prática de tornar os testes de software independentes de intervenção humana. Testar é uma prática intrínseca ao desenvolvimento de sistemas, mas testes de software só começaram a se tornar uma área de estudo da engenharia de software na década de 1970 e, desde então, têm ganho cada vez mais importância.

Hoje, existem grandes comunidades de profissionais especializados, conhecidos como testadores ou analistas de qualidade, e diversos estudos e cursos com ênfase nessa prática. Muitas empresas possuem grandes setores dedicados exclusivamente ao controle e garantia de qualidade.

Assim como os testes manuais, os testes automatizados têm como objetivo melhorar a qualidade de sistemas através da verificação e validação, mas a automação dos testes expande a área de estudo de testes de software e muda os paradigmas de implementação, manutenção e execução dos testes. Contudo, todo o conhecimento de testes de software pode ser aproveitado para automação.

A efetiva automação requer o uso de ferramentas específicas e de linguagens de programação de alto nível, portanto, é necessário um sólido conhecimento de ciência da computação para a criação de testes de qualidade. Como veremos no Capítulo 9, existem técnicas de escrita de testes automatizados que mudam completamente a maneira que programadores implementam um software.

Os testes automatizados também podem ser aproveitados para outros fins. Por exemplo, é possível utilizá-los para conhecer os efeitos colaterais de ferramentas e arcabouços. Já os testes de interface de usuário e de aceitação podem ser utilizados para demonstrações do software ou mesmo como um manual do usuário.

Ainda, relatórios gerados a partir dos casos de testes podem ser utilizados como documentação dos requisitos e do sistema. Esse tipo de documentação é dinâmica, pois pode ser gerada automaticamente e sem esforço, sempre que as baterias de testes forem executadas. Um benefício desse tipo de documentação é que ela dificilmente se torna obsoleta, já que ela se autoverifica toda vez que os testes são executados, isto é, se algum teste falhar, o relatório indica que aquele requisito não é mais satisfeito.

2.1 Cenário de Desenvolvimento com Testes Manuais

O modo convencional de desenvolvimento de uma funcionalidade é estudar o problema, pensar em uma solução e, em seguida, implementá-la. Após esses três passos, o desenvolvedor faz testes manuais para verificar se está tudo funcionando como o esperado. É normal que erros sejam detectados ao longo do processo de desenvolvimento; os desenvolvedores precisam encontrar o erro com técnicas de depuração e, então, corrigir e refazer o conjunto de testes manuais. Este ciclo se repete até que os desenvolvedores sintam-se seguros com o código-fonte produzido ou, em situações desastrosas, até que o prazo termine.

Com o objetivo de identificar possíveis erros remanescentes, também é comum submeter o software a uma avaliação de qualidade após o término do desenvolvimento e antes de colocá-lo em produção. Esse

controle de qualidade *a posteriori* geralmente é realizado com o auxílio de testes manuais executados por desenvolvedores, usuários ou mesmo por equipes especializadas em teste de software.

Este cenário é comum principalmente em empresas que utilizam metodologias rígidas que possuem fases bem definidas, geralmente derivadas do modelo de cascata [123]. Esse tipo de metodologia frequentemente leva à aparição de diversos problemas recorrentes na indústria de software, tais como atrasos nas entregas, criação de produtos com grande quantidade de erros e dificuldade de manutenção e evolução, devido principalmente às limitações da realização dos testes manuais.

A execução manual de um caso de teste é rápida e efetiva, mas a execução e repetição manual de um vasto conjunto de testes é uma tarefa dispendiosa e cansativa. É comum e compreensivo que os testadores priorizem os casos de testes mais críticos e não verifiquem novamente todos os casos a cada mudança significativa do código; é desse cenário que surgem diversos erros de software. Erros de software podem trazer grandes prejuízos para as equipes de desenvolvimento que perdem muito tempo para identificar e corrigir os erros e também para o cliente que, entre outros problemas, sofre com constantes atrasos nos prazos combinados e com a entrega de software de qualidade comprometida.

Mas o aspecto mais crítico deste cenário é o efeito "bola de neve". Como é necessário muito esforço para executar todo o conjunto de testes manuais, dificilmente a bateria inteira de testes é executada novamente a cada correção de um erro, como seria desejável. Muitas vezes, a correção de uma falha pode adicionar erros de regressão que são defeitos adicionados em módulos do sistema que estavam funcionando corretamente mas que foram danificados por alguma manutenção desastrada. A tendência é esse ciclo se repetir até que a manutenção do sistema se torne uma tarefa tão custosa que passa a valer a pena reconstruí-lo completamente.

2.2 A Abordagem dos Testes Automatizados

Muitos métodos ágeis (vide Seção 2.4), como Lean [112], Scrum [131] e XP [17] recomendam que todas as pessoas envolvidas em um projeto trabalhem controlando a qualidade do produto todos os dias e a todo momento, pois baseiam-se na ideia de que prevenir defeitos é mais fácil e barato que identificá-los e corrigi-los a posteriori. A Programação eXtrema (XP), em particular, recomenda explicitamente testes automatizados para ajudar a garantir a qualidade dos sistemas desenvolvidos.

Testes automatizados são programas ou *scripts* simples que exercitam funcionalidades do sistema em teste e fazem verificações automáticas nos efeitos colaterais obtidos [56]. A independência da intervenção humana permite o aproveitamento dos benefícios de um computador, como a velocidade de execução, reprodutibilidade exata de um conjunto de ações, possibilidade de execução paralela de testes, flexibilidade na quantidade e momento das execuções dos testes e a facilidade da criação de casos complexos de testes.

Uma das grandes vantagens dessa abordagem é que os casos de teste podem ser facilmente e rapidamente repetidos a qualquer momento e com pouco esforço. Os testes podem ser executados paralelamente[1], por exemplo através de grades computacionais[2]. A reprodutibilidade dos testes permite simular identicamente e inúmeras vezes situações específicas, garantindo que passos importantes não serão ignorados por falha humana e facilitando a identificação de um possível comportamento não desejado.

Além disso, como os casos para verificação são descritos através de um código interpretado por um computador, é possível criar situações de testes bem mais elaboradas e complexas do que as realizadas manualmente, possibilitando qualquer combinação de comandos e operações. Ainda, a magnitude dos testes pode também facilmente ser alterada. Por exemplo, é relativamente fácil simular centenas de usuários acessando um sistema ou inserir milhares de registros em uma base de dados, o que não é factível com testes manuais.

[1] Muitos arcabouços já facilitam a criação de testes paralelos, tais como Parallel-Junit, JUnit versões superiores a 4.6 e TestNG.

[2] A ferramenta SeleniumGrid cria uma grade computacional para execução de testes escritos com a ferramenta Selenium.

Todas essas características ajudam a solucionar os problemas encontrados nos testes manuais, contribuindo para diminuir a quantidade de erros [147] e aumentar a qualidade do software [43]. Como é relativamente fácil executar todos os testes a qualquer momento, mudanças no sistema podem ser feitas com segurança, o que ajuda a aumentar a vida útil do produto.

Na maioria das vezes, os testes automatizados são escritos programaticamente, por isso é necessário conhecimento básico de programação, mas existem também diversas ferramentas gráficas que escondem os detalhes de implementação possibilitando que clientes e outros profissionais que não sejam desenvolvedores também consigam escrever seus próprios testes. Ainda existem as ferramentas que separam a descrição do cenário de teste da sua implementação. Dessa forma, os clientes podem descrever os testes enquanto os desenvolvedores implementam trechos de código que ligam a especificação do cliente ao sistema em teste.

2.3 História

A ideia de testar manualmente sempre existiu, desde a época dos cartões perfurados até a do software de milhões de linhas de código, pois é uma prática comum e trivial de verificar algo que precise ser posto à prova. Segundo Craig Larman e Victor L. Basili, nos anos de 1960, a Agência Espacial Americana (NASA) utilizou práticas de desenvolvimento dirigido por testes (Seção 9.3) nos cartões perfurados do Projeto Espacial Mercúrio [84].

Testar é uma prática intrínseca ao desenvolvimento e é antiga a necessidade de criar *scripts* ou programas para testar cenários específicos [54], principalmente para os cenários nos quais testar manualmente era inviável. No entanto, utilizar testes automatizados como uma premissa básica do desenvolvimento é um fenômeno relativamente recente, com início em meados da década de 1990 [132]. Algumas práticas de desenvolvimento já evidenciavam a necessidade da criação de *scripts* de teste, tais como trechos de código utilizados para imprimir valores de variáveis (depuração com comandos `print`) e métodos `main` espalhados em trechos internos do código-fonte para fazer execuções pontuais do programa.

Contudo, o termo teste de software começou a se tornar um jargão na computação após o lançamento do livro *The Art of Software Testing*, de Glenford J. Myers [105, 120] e publicado em 1979. Já em 1988, David Gelperin e Bill Hetzel escreveram o artigo *The Growth of Software Testing* [62] onde classificavam a história do teste de software e previam sua tendência para o futuro.

Como foi discutido no artigo *The Growth of Software Testing*, até 1956 o desenvolvimento de software era orientado para depuração, isto é, fazia-se o software e, quando encontrado um erro, era feita sua busca e correção. Já entre 1957 e 1978, o desenvolvimento era orientado para demonstração, onde havia uma camada adicional no desenvolvimento para verificação dos erros. Entre 1979 e 1982, era orientado para destruição, onde havia uma preocupação em provar que o software estava correto através de muitos testes que procuravam encontrar erros no sistema, i.e., "destruí-lo". Depois de 1983 até 1987, orientado para avaliação, onde havia uma tentativa de encontrar os erros o mais cedo possível, para evitar depuração. Após 1988, os autores sugeriram que o desenvolvimento de software seria orientado para prevenção, o que foi concretizado com a popularização dos testes automatizados, com as recomendações dos métodos ágeis e com o desenvolvimento dirigido por testes [33].

Os testes automatizados disseminaram-se através de ferramentas especializadas, que facilitam a escrita, manutenção e execução dos testes. Essas ferramentas permitem separar o código-fonte dos testes e do sistema, evitando que os casos de testes interfiram no comportamento do sistema. O primeiro arcabouço conhecido é o `Taligent Test Framework` criado por Alan Liu e David McCusker em 1991 e publicado em 1995 [132], mas que não se popularizou. Por volta de 1994, Kent Beck criou o arcabouço `SUnit` para linguagem SmallTalk [13], que até hoje é utilizado como referência para arcabouços semelhantes para outras linguagens de programação.

O SUnit era tão simples que Kent Beck não imaginava que ele teria algum valor, mas ao utilizá-lo para testar algumas estruturas de dados de um sistema em que trabalhava, ele acabou encontrando um número considerável de defeitos. A partir desse caso de sucesso, a prática de testes automatizados disseminou-se entre diversas comunidades. Em 1998, Kent Beck e Erich Gamma desenvolveram o arcabouço JUnit para a linguagem Java, inspirado no SUnit, e foi então que os testes automatizados começaram a se tornar uma prática altamente disseminada nas boas equipes de desenvolvimento de software em todo o mundo. Em 2002, Kent Beck lançou o livro *Test-Driven Development: By Example* [16], que propõe um novo paradigma de desenvolvimento, onde a implementação do software é guiada pela criação de casos de testes automatizados.

Atualmente, a automação de testes é considerada por diversas metodologias, principalmente entre aquelas identificadas com os métodos ágeis, como uma prática básica para garantir a qualidade de um software. Muito se tem estudado, novos conceitos e técnicas de desenvolvimento foram criadas e as ferramentas estão cada vez mais poderosas e práticas. Isso tudo facilita não apenas a escrita de testes para trechos específicos de código, mas também para a integração de módulos, interfaces gráficas e bancos de dados.

O retrato da importância que se tem dado aos testes automatizados está nas implementações de muitas ferramentas de software populares que já trazem consigo módulos para integração e realização dos testes. Por exemplo, Maven que é uma ferramenta para gerenciamentos de projetos, assim como Ruby on Rails, Grails e Lift que são arcabouços para aplicações Web, fornecem uma arquitetura padronizada para a criação de casos de testes automatizados. A linguagem Python também incentiva a automação dos testes, fornecendo em sua biblioteca padrão um arcabouço para testes de unidade. A linguagem Ruby possui o RSpec que é integrado por diversos arcabouços para programação Web com Ruby.

2.4 Métodos Ágeis de Desenvolvimento

A evolução da engenharia de software deu-se a partir do modelo de cascata que propunha fases estanques para o desenvolvimento de software [123]. Do aprimoramento do modelo de cascata surgiram novos processos, tais como o modelo em espiral e o Rational Unified Process (RUP) [31], todos com grande ênfase na documentação do processo.

Devido à grande quantidade de fracassos de projetos de software [66, 67, 65], nas últimas décadas alguns líderes de projetos adotaram modos de trabalho que se opunham a este modelo tradicional, e tiveram grandes sucessos [57]. Até que em 2001, 17 desses líderes, que possuíam formas de trabalho semelhantes, juntaram-se para debater metodologias de desenvolvimento na tentativa de criar um novo método que agregasse as melhores ideias. No entanto, essa discussão levou à conclusão de que era difícil definir um método perfeito para todas as situações; no entanto, chegou-se a um consenso de 12 princípios, que foram sintetizados nas premissas do Manifesto Ágil [18].

Dentre os métodos ágeis que satisfazem o manifesto, existem os que focam em aspectos mais gerenciais, como Lean [109, 112] e Scrum [131], e outros que também dão ênfase a práticas de desenvolvimento de software tal como a Programação eXtrema (XP) [17]. Todos preconizam o controle de qualidade disseminado por toda a equipe e durante todo o desenvolvimento.

O controle de qualidade no desenvolvimento com métodos ágeis normalmente é associado à automação de testes, já que essa prática surgiu da mesma comunidade. Automação de testes é uma das práticas primárias de XP [15]. As baterias de testes podem ser executadas sem esforço a todo momento, o que possibilita a verificação contínua da qualidade do sistema durante e após a implementação.

No entanto, a automação de testes não é exclusiva dos métodos ágeis e nem depende significativamente de outras práticas, por isso é uma técnica de desenvolvimento independente que pode ser empregada por qualquer equipe utilizando qualquer metodologia, mesmo as mais tradicionais. Também é importante ressaltar que os métodos ágeis não se opõem a quaisquer revisões adicionais que sejam

feitas para aumentar a qualidade, apenas não é uma prática primária da filosofia.

2.4.1 Programação eXtrema

A Programação eXtrema, também conhecida como XP (de *Extreme Programming*), foi criada por Kent Beck em 1996, o mesmo criador do arcabouço de testes `SUnit` [13], que serviu de referência para muitos outros arcabouços de testes automatizados. XP surgiu de um desafio de reverter a situação de um projeto de folha de pagamento da empresa *Chrysler*, que já havia estourado os limites de custos e prazos. Para isso, o projeto adotou uma nova metodologia, que aplicava, ao extremo, um conjunto de práticas recomendadas de programação [19, 6] com disciplina e organização. Devido ao sucesso do projeto, Kent Beck reuniu as práticas que trouxeram os méritos da metodologia e a oficializou como Programação eXtrema, através do primeiro livro de XP [15].

Dentre as práticas recomendadas por XP temos os Testes Automatizados [44], que estão diretamente relacionados a outras práticas da metodologia. Algumas das práticas de XP dependem fortemente dos testes para que sejam executadas com sucesso. Por isso, para aplicar XP apropriadamente, é fundamental o emprego efetivo de testes automatizados. A seguir, descrevemos as principais práticas de XP que se relacionam diretamente com testes automatizados.

Refatoração

Refatoração é o processo de alterar um sistema de software para aperfeiçoar a estrutura interna do código-fonte sem alterar seu comportamento externo [59, 110]. Este processo, realizado através de **passos pequenos** e **sistematizados**, é um artifício poderoso para aprimorar o *design* da aplicação e melhorar a legibilidade e clareza do código. Existem ferramentas que auxiliam na automatização dessa tarefa [14, 122, 79] e estudos de refatorações em outras área do projeto, tais como banco de dados [4].

No entanto, como toda manutenção de código, refatoração também está sujeita a introduzir erros no projeto, seja através do descuido ou do manuseio incorreto de ferramentas. Por isso, é essencial que exista uma boa bateria de testes automatizados que assegure que os comportamentos não foram modificados indevidamente. Do ponto de vista do cliente, um erro introduzido em uma parte do sistema que estava funcionando corretamente pode ser frustrante, por isso o uso de testes automatizados em conjunto com a refatoração é uma prática fundamental.

Propriedade coletiva do código

Propriedade coletiva do código é a prática que propõe que todos os membros da equipe são responsáveis de alguma maneira por todo o código-fonte e, portanto, todos têm total liberdade para trabalhar em cima do código criado por outro membro. Essa prática é fundamental para não tornar um projeto dependente de um programador específico, assim como ajuda na velocidade do desenvolvimento, dado que qualquer trecho do código pode ser modificado a qualquer momento, aumentando a disponibilidade de trabalho e direcionando os esforços em algo que agrega valor diretamente ao produto final.

No entanto, esta prática traz riscos já que cada desenvolvedor possui ideias e maneiras próprias de solucionar problemas de computação. O conflito de soluções pode desorganizar o código-fonte e estragar o que estava funcionando. Portanto, esta é uma operação muito suscetível a erros e merece um controle de qualidade com alta cobertura de casos de testes que possam ser executados a qualquer momento e de forma ágil, como é possível com testes automatizados.

Design incremental

É uma das práticas mais conflitantes com as metodologias tradicionais, pois ela incentiva a **não** planejar todo o esqueleto da aplicação de uma só vez, e sugere que o *design* seja construído gradativamente de

acordo com o aprendizado da equipe e as necessidades prioritárias do cliente.

Entretanto, alterações de *design* e de arquitetura no sistema podem ser muito perigosas já que podem afetar diversos módulos do sistema, ou seja, muitas classes poderão ter de ser refatoradas. Essas alterações são ainda mais críticas quando utilizadas linguagens com tipagem dinâmica, que não possuem ajuda do compilador para verificação de tipos de variáveis, pois as interfaces de muitas classes podem ser alteradas, o que possibilita a inserção de erros de integração.

Por isso, a extrema importância de testes que sejam abrangentes, como os testes automatizados de integração e aceitação. Técnicas de escrita de testes, como Desenvolvimento Dirigido por Testes e por Comportamento (vide Capítulo 9), também influenciam diretamente no *design*, dado que ele emerge à medida que novos casos de testes são adicionados.

Integração Contínua

Integração Contínua é uma premissa do desenvolvimento incremental e das entregas frequentes ao cliente. Ela tem como objetivo integrar rotineiramente o sistema e todas suas dependências para verificar que nenhuma modificação tenha danificado o sistema, sejam elas alterações no código-fonte, em configurações ou mesmo em dependências e outros fatores externos [52].

Parte do processo de verificação é feito pelo próprio compilador que verifica erros estáticos do código-fonte e de mapeamentos de dependências. Já os erros de lógica, de configuração e integração de componentes só podem ser verificados em tempo de execução, por exemplo, através de testes automatizados. Muitas ferramentas para automação de testes já possuem artifícios que facilitam a execução automática dos casos de testes, o que facilita a configuração nos ambientes de integração contínua.

Entregas Frequentes

O ciclo de entrega de versões para o cliente deve ser curto, assim a equipe foca seu tempo nas tarefas mais prioritárias e o cliente consegue dar *feedback* rápido a respeito do software produzido. Segundo as metodologias ágeis, é dessa aproximação, entre cliente e equipe de desenvolvimento, que o software evolui da melhor forma para atender às principais necessidades do cliente.

Entregas frequentes implicam alterações rotineiras no código do sistema, tornando o software altamente vulnerável a erros de regressão, que é um dos principais tipos de erro que os testes automatizados ajudam a prevenir. Essas alterações vão se tornando cada vez mais perigosas à medida que o sistema fica mais extenso e mais complexo, por isso fazer entregas frequentes só são interessantes quando há segurança para fazer as modificações.

Tracking

Acompanhamento do projeto ou *tracking* é uma das atividades propostas em XP para ajudar a gerenciar o desenvolvimento do software. Esta atividade se dá através da coleta, observação e interpretação de métricas (Capítulo 10). As métricas que serão coletadas e analisadas dependem do contexto atual do sistema e das decisões tomadas pela equipe, isto é, a metodologia não possui regras ou métricas obrigatórias que devem ser monitoradas.

Entretanto, o acompanhamento da qualidade do produto final é natural em projetos sérios que valorizam a criação de bons produtos. Todavia, qualidade é um aspecto subjetivo, então é necessário utilizar diversas métricas que consigam representar satisfatoriamente a qualidade do produto final. Entre uma infinidade de aspectos que podem ser acompanhados, estão o *design*, elegância e simplicidade do código-fonte, assim como as métricas de testes automatizados. Como os testes influenciam diretamente na qualidade do produto final, as métricas são fundamentais para o acompanhamento e gerenciamento da qualidade do projeto.

Metáfora, Envolvimento real com o Cliente e Testes de Aceitação

A comunicação efetiva é fundamental para o sucesso de um sistema de software, contudo ela não é trivial. Mesmo um texto sem ambiguidades pode ter interpretações diferentes por seres humanos, já que a discrepância de conhecimento e de vocabulário podem levar a múltiplas interpretações. É natural que pessoas de comunidades específicas tenham um modo de falar e escrever peculiar, utilizando termos próprios ou mesmo um vocabulário formal que não é rotineiro para outras comunidades.

É por isso que XP incentiva um envolvimento real com o cliente, para que a equipe e os clientes eliminem problemas de má interpretação e criem um vocabulário próprio a partir de metáforas que todos consigam interpretar da mesma maneira. Uma forma de facilitar a criação de metáforas é através dos testes de aceitação que criam uma ponte de comunicação entre cliente e desenvolvedores por meio de documentos úteis que ajudam a encontrar defeitos e a certificar que o sistema é válido, isto é, faz o que deveria ser feito.

Como foi descrito no decorrer de toda essa Seção, testes automatizados estão fortemente relacionado com as principais práticas de métodos ágeis. Para algumas delas, a escrita de testes automatizados é um pré-requisito, enquanto, para outras, a automação dos testes traz muitas vantagens.

2.5 Software Livre

O movimento de Software Livre (também conhecido como Software Aberto ou *Open Source*) juntamente com a Internet explicitou a natureza e o potencial das redes colaborativas [22, 23], que possibilitam a criação de sistemas imensos como o GNU Linux[3] e com uma quantidade enorme de dados como a Wikipedia[4]. Hoje existem diversas frentes de incentivo e de estudo de Software Livre[5] que visam aumentar o uso de todo seu potencial, além buscar formas de melhorias para as contribuições e para o gerenciamento dos projetos. O projeto QualiPSo[6] faz parte de uma dessas frentes, que possui, como um de seus objetivos, estudar formas para aumentar a qualidade dos sistemas de software livre, como, por exemplo, através de testes automatizados.

Para os criadores e mantenedores de projetos livres, o resultado esperado é que o projeto tenha uma boa repercussão e que tenha contribuições de qualidade. Para uma boa repercussão, a qualidade do sistema é essencial e pode ser conseguida mediante os testes automatizados ou de outra prática que está no controle e no domínio dos mantenedores. No entanto, as colaborações nem sempre seguem os mesmos padrões de qualidade, elas podem ser muito heterogêneas, não padronizadas e de pessoas distantes e desconhecidas, o que pode dificultar e atrasar a comunicação. Dessa forma, cada contribuição precisa de um estudo cuidadoso para certificar que ela está agregando algum valor ao sistema e não está adicionando defeitos em outros módulos do projeto.

Essa falta de agilidade na aceitação de contribuições e o medo que pode existir na inclusão de código de colaboradores podem ser melhorados ou mesmo sanados com ajuda de testes automatizados. Os testes documentam a quais situações o sistema foi submetido e trazem segurança para modificações do código, pois permitem avaliar se erros de regressão foram adicionados a qualquer momento.

Essa insegurança é recíproca para os colaboradores que também têm como objetivo obter boa aceitação das suas contribuições, além de querer que o projeto evolua da melhor maneira possível. Além disso, se o colaborador evidencia aos mantenedores do projeto que sua contribuição é de qualidade, então a chance dela ser incluída no projeto aumenta. Por isso, é importante que todas as contribuições sejam acompanhadas de uma boa bateria de testes automatizados.

Por fim, a prática de testes automatizados traz benefícios para todas as pessoas envolvidas com

[3] `www.linux.org`
[4] `wikipedia.org`
[5] `ccsl.ime.usp.br`
[6] `www.qualipso.org`

o software livre. Os usuários que obtêm um produto de melhor qualidade, além de ganharem com a velocidade e tornarem menos burocrático o processo de melhorias do projeto.

2.6 Qualidade

O principal objetivo dos testes automatizados é melhorar a qualidade do software, mas qualidade é um termo muito amplo, podendo indicar uma infinidade de aspectos em um contexto específico, como, por exemplo, desempenho, segurança, flexibilidade, estruturação interna do código, usabilidade, entre outros [136, 148]. Apesar da qualidade ser um conceito elusivo e difícil de ser medido [48], os testes automatizados podem contribuir para melhoria dos sistemas. Por isso, nas subseções seguintes, há a descrição dos possíveis elos entre determinada característica de qualidade e testes automatizados. Alguns dos aspectos citados estão na norma internacional ISO 9126, outros são termos rotineiros de Engenharia de Software.

Correção

Correção é a característica de um software fazer corretamente o que foi proposto [77] e é o aspecto de qualidade mais básico e fundamental de qualquer sistema de software. Alguns erros de correção são supérfluos e nem chegam a incomodar os usuários, já outros são intoleráveis, podendo tornar um sistema inutilizável e acabar com sua reputação.

Existem diversos tipos de testes que verificam a correção dos sistemas, tais como os testes de unidade, integração, interface e de aceitação, os quais serão descritos na Seção 3.3. Para criação destes testes é necessário definir os dados de entrada de um módulo do sistema e quais devem ser os dados de saída desejados. Os resultados esperados são comparados com os resultados obtidos através da análise dos efeitos colaterais (valor de retorno, alteração do valor de variáveis etc.) causados pela execução daquele módulo com os dados de entrada correspondentes.

A correção dos sistemas depende de cada uma de suas unidades, assim como da integração correta das camadas e módulos do sistema. As unidades podem ser compostas de algoritmos bem coesos, que não dependem de outros módulos para processar as informações, ou de algoritmos que precisam de objetos colaboradores para produzir um resultado coerente. Em outros casos, especialmente em classes abstratas, podem existir trechos de código que apenas definem parte da estrutura da solução de um problema. No entanto, é possível definir cenários de testes que verifiquem a correção dessas classes, assim como pode ser verificado se a sua estrutura atende às necessidades.

Em outras situações, os algoritmos podem ter muitas possibilidades de combinações de dados de entrada e saída, o que torna inviável ou até impossível fazer testes para cada uma das combinações. Para esses casos, pode ser criada uma bateria de testes de correção baseada em certas características do algoritmo e outra bateria de testes mais abrangente e menos específica (testes aleatórios e de sanidade) que pode verificar uma quantidade maior de casos que podem dar mais segurança de que não foram cometidos erros desastrosos.

Contudo, raros são os casos onde é possível provar a correção do sistema apenas com baterias de testes. Como dizia Dijkstra, "Testes são muito eficazes para mostrar a presença de erros, não sua ausência" [50]. Para provar a correção de um módulo com testes é necessário verificar todos os casos possíveis, isto é, deve haver uma quantidade finita de combinações de dados de entrada e saída.

Outra forma de se provar a correção de um sistema é através de modelos matemáticos e formais, que geralmente são processos de alto custo, pois exigem pessoas muito especializadas. Um malefício dessa alternativa é a falta de praticidade para alteração do código do sistema, pois para cada alteração é preciso refazer o modelo matemático. Essa rigidez de desenvolvimento torna impraticável o emprego de outras técnicas recomendadas por métodos ágeis, tais como entregas frequentes e refatoração.

Para sistemas críticos, como os espaciais, de aviação e sistemas médicos, é imprescindível a prova e/ou uma bateria completa e minuciosa de cenários de testes. Para esses casos, uma possível solução é desenvolver o sistema com práticas ágeis, incluindo os testes e, no final de uma entrega, podem ser feita avaliações matemáticas para certificar que o sistema está correto.

Robustez

Meyer define robustez como "a habilidade de sistemas de software reagirem apropriadamente sob condições adversas" [100, 135], tais como entradas inválidas de usuários, infraestrutura lenta ou concorrência de processos. Também segundo Meyer, "a robustez complementa a correção. A correção trata do comportamento descrito pela especificação; a robustez caracteriza o que não foi especificado."

Normalmente, as histórias ou requisitos do cliente não contêm, de maneira explícita, que o sistema deve reagir corretamente sob certas condições não triviais que um software pode estar sujeito. Muitas vezes, essas informações são responsabilidade do desenvolvedor bem capacitado para tomar as devidas precauções no momento oportuno.

Programaticamente é possível simular erros de software e de hardware. Muitas ferramentas possuem artifícios para facilitar a escrita de casos de testes de situações de erro, facilitando a verificação e tornando o código dos testes mais legível. Também é possível criar objetos falsos que simulam erros de hardware e infraestrutura, lançando as exceções ou os códigos de erro relacionados.

Flexibilidade

Uma boa arquitetura não deve ser apenas robusta, é preciso que ela seja flexível para aceitar a adição de novas funcionalidades com pouco trabalho e sem requerer um conhecimento profundo da arquitetura já existente. A orientação a objeto e Padrões de Projeto [61] fornecem boas alternativas que facilitam a criação de software flexível.

Quando pensamos em testes automatizados, também temos de considerar que a arquitetura do sistema seja testável (Capítulo 10), para que o custo-benefício dos testes automatizados seja alto. Esta característica está diretamente ligada com a simplicidade e boa modularização do *design* do sistema, que é a proposta da orientação a objetos e Padrões de Projeto. Quando a implementação de um software é dirigida por casos de teste (Seção 9.3), o *design* emerge com simplicidade e com alta testabilidade, e, portanto, o *design* tende a ficar altamente flexível.

Eficiência

A missão de otimizar sistemas é uma preocupação de todo bom programador, mas a realização não criteriosa dessa tarefa pode resultar na criação de um projeto mal arquitetado e com código-fonte pouco legível. Esta otimização realizada de forma aleatória é bem ilustrada pela frase de Donald Knuth[7]: "Otimização prematura é a raiz de todo mal" [81]. A recomendação básica é otimizar os gargalos do sistema, que geralmente consomem uma grande porcentagem de todo o tempo gasto na execução do software. Comumente estes gargalos são encontrados através da ajuda de *Profilers*[8], que executam módulos do sistema medindo seu desempenho. Otimizar partes do sistema que não são os gargalos trazem um benefício desprezível, potencialmente, a um alto custo em termos de clareza de código e dispêndio de recursos humanos.

[7] Donald Knuth é professor emérito da Universidade de Stanford e é autor de livros importantes como a série *The Art of Computer Programming* e criador de programas mundialmente conhecidos como o Tex.

[8] *Profiler* é uma ferramenta para a análise de desempenho por meio da medição do tempo e frequência da chamada de funções.

Os testes automatizados também exercitam trechos específicos do sistema e podem disponibilizar o tempo de execução de cada teste. Mas é importante notar que o objetivo das baterias de testes que buscam erros não é medir o desempenho do sistema. Primeiro, porque as ferramentas de testes não são específicas para isso e, portanto, não substituem o uso de *Profilers*. Segundo, porque queremos que as baterias de testes sejam executadas muitas vezes por minuto.

Consequentemente, é importante que o tempo de execução seja muito pequeno, o que nem sempre acontece quando queremos encontrar gargalos. Por último, muitas vezes os casos de testes não ilustram com fidelidade a situação real de uso do sistema, por exemplo quando é utilizado Objetos Dublês (Seção 6.2), que são implementações falsas para facilitar a escrita dos testes. Por isso, os valores não têm credibilidade para serem utilizados para otimizar o sistema. Contudo, testes lentos até podem ser indicativos de gargalos do sistema.

Existem diversos tipos de testes que servem especificamente para identificar pontos de ineficiência do sistema, como testes de desempenho, de estresse, carga e longevidade. Todos eles geralmente exigem uma grande quantidade de dados, usuários ou tempo no decorrer das simulações, por isso, é inviável a realização manual dos testes e sem o auxílio de ferramentas adequadas.

Segurança

A segurança de sistemas de software é fundamental principalmente para os programas mais visados por pessoas mal intencionadas, como os que envolvem dinheiro ou informações sigilosas que podem trazer grandes prejuízos para empresas e pessoas. No caso de aplicações Web, a atenção deve ser redobrada porque elas ficam expostas a usuários anônimos e mal intencionados e, portanto, ficam mais suscetíveis a ataques [145].

Um sistema de software pode possuir diversas fontes para possíveis vulnerabilidades de segurança. Hoje, mesmo os sistemas pequenos possuem em geral muitas dependências de bibliotecas, *middleware* e arcabouços, todas com possibilidade de conter falhas de segurança, além do seu próprio código-fonte e do sistema operacional. Servidores de aplicações também são críticos neste aspecto, já que qualquer versão insegura pode trazer consequências negativas para milhares de usuários.

Portanto, idealmente todos os testes de segurança devem ser executados a cada atualização de software do servidor ou das dependências do projeto para averiguar que vulnerabilidades de aplicações, do servidor ou mesmo de conflitos entre versões, tenham sido inseridas no processo. Por isso, a automação dos testes de segurança é fundamental devido a facilidade da repetição de todos os casos de testes.

Durabilidade

Produtos de software podem estar sempre em manutenção, pois podem ter de incorporar novas funcionalidades, adaptar-se a novas plataformas e atingir novos mercados. Além disso, correções e melhorias colaboraram com o sucesso do software. A durabilidade de um software se caracteriza pelo tempo em que é utilizado [7].

Um software pode ficar obsoleto por diversas razões, tais como concorrência de outros produtos e surgimento de novas gerações de tecnologias. Uma razão que geralmente é acompanhada por prejuízos é quando o custo de manutenção do sistema se torna tão grande que é preferível reconstruí-lo do início. A dificuldade de manutenção pode estar associada às tecnologias envolvidas, falhas de arquitetura do software e também ao ciclo de erros de regressão, que é o processo de adicionar novos erros durante uma manutenção de outra falha.

Os erros de regressão podem diminuir drasticamente o tempo de vida útil de um sistema, isso porque o tempo gasto com essas manutenções não agrega novas funcionalidades ao sistema, podendo deixar a evolução do software estagnada. Ainda, o sistema pode perder sua credibilidade com os usuários e

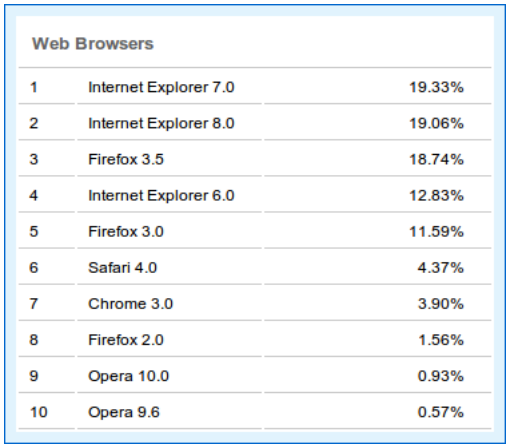
clientes devido aos erros, além de tornar o trabalho cada vez mais desgastante para a equipe de desenvolvimento.

Um sistema não seguro para mudanças, ou seja, sem uma bateria de testes que possa ser executada a qualquer momento de maneira ágil, está sempre sujeito aos erros de regressão quando o software é modificado. Os métodos ágeis ressaltam que mudanças em um sistema de software são normais e devem ser encaradas com naturalidade, sejam elas grandes mudanças arquiteturais ou simples correções de leiaute, mas elas precisam ser feitas sem riscos de prejudicar a qualidade do sistema.

Portabilidade

Atualmente, existe uma grande diversidade de ambientes distintos que um programa precisa se adaptar, não só em relação a sistemas operacionais, como também navegadores Web e dispositivos móveis. Por isso, é fundamental para o sucesso e lucratividade de um software que ele seja portável para diversos ambientes para que alcance o maior público possível.

Segundo a W3Counter[9], em 2009 duas famílias de navegadores dominavam grande parte do mercado: `Mozilla Firefox` e `Microsoft Internet Explorer` (Figura 2.1). Além disso, dentro de uma mesma família de navegadores existem diversas versões intensamente utilizadas, como é o caso das versões 7 e 8 do `Internet Explorer` e 3 e 3.5 do `Firefox` que possuem diferenças importantes que podem trazer problemas de incompatibilidade para as páginas Web, assim como pode acontecer com uma mesma versão do navegador sobre sistemas operacionais diferentes. Problemas comuns estão relacionados com a não padronização de tecnologias e falhas na interpretação de *tags* HTML, comandos JavaScript e descrições CSS[10].



Web Browsers

1	Internet Explorer 7.0	19.33%
2	Internet Explorer 8.0	19.06%
3	Firefox 3.5	18.74%
4	Internet Explorer 6.0	12.83%
5	Firefox 3.0	11.59%
6	Safari 4.0	4.37%
7	Chrome 3.0	3.90%
8	Firefox 2.0	1.56%
9	Opera 10.0	0.93%
10	Opera 9.6	0.57%

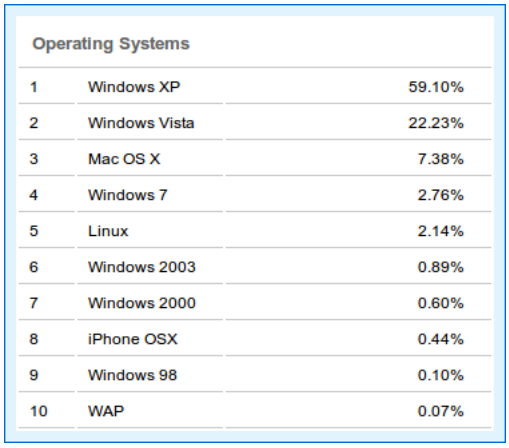
Figura 2.1: Popularidade de Navegadores Web em novembro de 2009 (Fonte: W3Counter).

Em relação aos sistemas operacionais o equilíbrio ainda está muito distante, pois a família `Windows` domina cerca de 85% do mercado (Figura 2.2). Contudo, 15% é uma porcentagem grande pensando no total de usuários da Internet, portanto a portabilidade entre sistemas operacionais também é fundamental para boa repercussão e lucratividade de um software.

A popularização dos dispositivos móveis, tais como celulares, computadores de mão e *smart phones* também tem trazido a tona a importância da portabilidade. Para cada tipo de aparelho existe uma grande diversidade de marcas e arquiteturas de hardware, que possuem infraestruturas específicas de programação. Ainda, com as tecnologias de transmissão de dados entre aparelhos como *Bluetooth*, *Wi-Fi* e infravermelho, a necessidade de compatibilidade entre diferentes sistemas é acentuada.

[9] W3Counter é uma página Web que agrupa informações de tráfego de milhares de páginas populares da Web.

[10] Cascading Style Sheets ou CSS são folhas de estilo que são aplicadas em componentes Web para alterar o *design*.



Operating Systems

1	Windows XP	59.10%
2	Windows Vista	22.23%
3	Mac OS X	7.38%
4	Windows 7	2.76%
5	Linux	2.14%
6	Windows 2003	0.89%
7	Windows 2000	0.60%
8	iPhone OSX	0.44%
9	Windows 98	0.10%
10	WAP	0.07%

Figura 2.2: Popularidade de Sistemas Operacionais em novembro de 2009 (Fonte: W3Counter).

As linguagens portáveis, que usam máquinas virtuais para interpretar códigos compilados em um formato específico (*byte codes*), tais como Java, Python e Ruby, estão cada vez mais populares (Figura 2.3, obtida em 2011 da página Web *Programming Language Popularity*[11] e Figura 2.4, obtida em 2009 da página Web *TIOBE Software*[12]). Uma das razões é que elas facilitam muito a escrita de programas portáveis, já que elas abstraem as diferenças do hardware das máquinas e criam uma API comum para todas elas. No entanto, não é garantido que a linguagem ou outras ferramentas sejam portáveis para todas as funcionalidades de um hardware porque podem haver diferenças enormes de tecnologias que torna impossível definir uma API padrão para todas as máquinas.

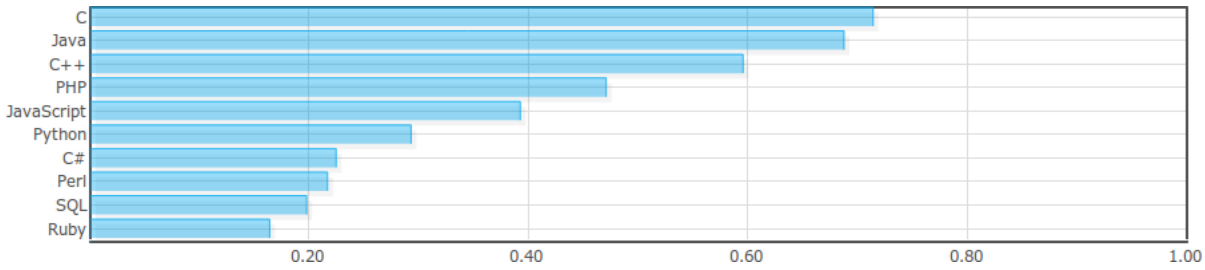


Figura 2.3: Indicação normalizada de popularidade de linguagens de programação no começo de 2011.

Uma maneira de assegurar que um sistema de software funciona em um determinado ambiente é ser submetido a pelo menos uma bateria de testes. Mas é inviável que pessoas realizem este trabalho repetidamente para cada ambiente, pois o número de casos de teste é multiplicado pelo número de plataformas a serem testadas. Por exemplo, se temos uma bateria simples de testes para uma aplicação Web contendo 500 casos de teste e existe uma exigência da aplicação ser executada pelo menos nos navegadores mais populares (Firefox 2 e 3 e Internet Explorer 6 e 7) sob o sistema operacional mais popular (Windows), teremos um total de 2000 casos de testes a serem executados.

No caso dos testes automatizados, basta executar a mesma bateria de testes em diferentes sistemas operacionais e plataformas para se certificar que nenhum erro de incompatibilidade ocorreu. Já no caso de aplicações Web, existem hoje ferramentas, como a Selenium[13], que auxiliam a criação de testes

[11] A página Web *Programming Language Popularity* (`http://www.langpop.com`) faz pesquisas em alguns grandes portais de buscas e de incubadoras de sistemas de software para coletar informações que são agrupadas e analisadas estatisticamente para tentar definir a popularidade das linguagens de programação.

[12] A ordenação por popularidade da página Web *TIOBE Software* (`www.tiobe.com`) é feita com base em dados coletados em grandes portais de buscas.

[13] Ferramenta para testes de aplicações de interface Web.

Position May 2009	Position May 2008	Delta in Position	Programming Language	Ratings May 2009	Delta May 2008	Status
1	1	=	Java	19.537%	-1.35%	A
2	2	=	C	16.128%	+0.62%	A
3	3	=	C++	11.068%	+0.26%	A
4	4	=	PHP	9.921%	-0.28%	A
5	5	=	(Visual) Basic	8.631%	-1.16%	A
6	7	↑	Python	5.548%	+0.65%	A
7	8	↑	C#	4.266%	+0.21%	A
8	9	↑	JavaScript	3.548%	+0.62%	A
9	6	↓↓↓	Perl	3.525%	-2.02%	A
10	10	=	Ruby	2.692%	+0.05%	A

Figura 2.4: Outra indicação de popularidade de linguagens de programação: 2008 e 2009.

automatizados portáveis entre navegadores, que abstraem as diferenças internas como as de JavaScript e CSS.

Usabilidade

A usabilidade dos sistemas é fundamental para facilitar seu aprendizado e para agilizar o acesso às informações pertinentes [133, 98]. Quanto mais intuitiva for a interface, menor será o tempo gasto com o estudo e entendimento do sistema. Atualmente, esta característica está cada vez mais em destaque, por causa do uso intenso do computador, da grande variedade de programas e da imensa quantidade de conteúdo disponível nas páginas Web.

Uma das estratégias de testes de usabilidade se dá através da observação do uso do sistema por usuários selecionados de acordo com um perfil desejado. As ações dos usuários podem ser filmadas ou documentadas para futura avaliação. A partir das informações coletadas é feita análise da facilidade de uso do sistema.

Na área de testes automatizados ainda há a necessidade de mais ferramentas especializadas que facilitem a coleta e interpretação das informações de usabilidade, mas muito já pode ser feito através de ferramentas de gravação de ações do usuário, de interceptadores de requisições ou ainda com auxílio de programação orientada a aspectos. A interpretação das ações coletadas é a tarefa mais complicada, pois deve ser baseada em definições subjetivas ou em heurísticas de Interação Humano-Computador.

Acessibilidade

Acessibilidade em software é a característica de usabilidade de possibilitar que mesmo usuários com necessidades especiais tenham acesso à utilização de um software. Os usuários especiais são indivíduos que possuem alguma peculiaridade física ou pscicológica que dificulta ou impede o uso de sistemas de software pelas interfaces de usuário tradicionais [42]. Dentre esses usuários especiais estão idosos, deficientes visuais e auditivos, tetraplégicos, entre outros.

Com as ferramentas de testes automatizados de interface Web e de *Desktop*, é possível criar testes para verificar a acessibilidade através da criação de eventos originários de dispositivos específicos. Por exemplo, podemos criar um caso de teste que tenta executar uma determinada ação apenas com eventos do *mouse* ou apenas com eventos do teclado. Também é possível fazer testes que captam informações do leiaute, tais como tamanho das letras dos componentes de texto.

Beleza

Dependendo do tipo de sistema e do produto vinculado, beleza é um atrativo fundamental para atrair novos clientes e usuários. Por isso, muitas aplicações, principalmente as voltadas para Web, possuem *designers*, profissionais especializados em arte eletrônica que trabalham com a geração de mídias e com a organização da interface de usuário para torná-la atraente e agradável.

Esse trabalho artístico, que pode ser realizado antes, depois ou paralelamente ao desenvolvimento do sistema, pode embutir diferentes tipos de não conformidades no sistema. A interface possui muitas responsabilidades, por isso ela está sujeita a erros de correção, robustez, portabilidade e falhas específicas da interface gráfica, como usabilidade e acessibilidade ruim. Por isso, toda alteração do *design* precisa ser encarada como uma mudança do código-fonte, que exige que todos os testes sejam executados novamente. Como esta tarefa pode ser rotineira, principalmente no caso das páginas Web, é necessário a agilidade dos testes automatizados para a certificação de que a qualidade do sistema não foi prejudicada.

2.7 Conclusões

A prática de testes automatizados está cada dia mais popular e disseminada entre as comunidades de desenvolvimento de software devido, principalmente, ao incentivo dessa prática por parte dos métodos ágeis e por linguagens e tecnologias que facilitam a criação e manutenção do código dos testes.

A automação de testes é uma prática de baixo custo de criação e manutenção que ajuda a garantir a qualidade dos sistemas de software se comparado com outras estratégias, tais como análises formais que exigem um alto grau de especialização de profissionais e também com testes manuais que não trazem seguranças para mudanças, não eliminam o trabalho de depuração e de documentação dos testes.

Uma das premissas dos métodos ágeis é tornar o desenvolvimento de software adaptável a mudanças, já que é difícil conhecer todos os requisitos e o melhor *design* desde o começo. É impossível prever com exatidão e certeza todas as possíveis dificuldades e fatores externos que podem afetar o andamento de um projeto, tais como flutuações do mercado e da concorrência que podem mudar as prioridades dos requisitos.

O trabalho de depuração, que geralmente é acompanhado de testes manuais, não agrega valor diretamente ao produto final, ele apenas ajuda a localizar o motivo de um erro que já foi detectado para futura correção. Esta tarefa pode ser demorada e estressante para os desenvolvedores, principalmente quando o erro é por uma falha simples que podia ser evitada com um pouco mais de atenção durante o desenvolvimento.

No caso dos testes automatizados, que evitam o trabalho de depuração, todo o tempo gasto está vinculado diretamente com a verificação e validade do sistema, mesmo nos casos de fracasso. Por isso, é um tempo de investimento real em qualidade. Além disso, todo o tempo dedicado para escrita dos testes traz benefícios durante o decorrer do desenvolvimento do software.

Por fim, é importante estar ciente que todas as abordagens para melhoria da qualidade dos sistemas de software exigem intervenção humana no processo de criação, sendo assim, todas estão sujeitas a imperfeições. Por isso, como nenhuma das abordagens são conflitantes entre si, elas podem ser combinadas com o intuito de agregar os benefícios de cada uma para tornar o processo de controle de qualidade mais eficaz. Mas o que é fundamental é dar ênfase à abordagem que mais trará benefícios dentro do contexto do sistema.

Capítulo 3

Definições e Terminologia

A área de testes de software possui um linguajar próprio, extenso e com muitas nomenclaturas [125], sendo que algumas possuem interpretações diferentes nas diversas comunidades, ferramentas e também de autores importantes de um mesmo grupo de pesquisa [99]. Testes de software são estudados pela comunidade de desenvolvimento ágil, pela comunidade tradicional de engenharia de software e também por grupos de testadores e de controle de qualidade, todas com um dialeto próprio, principalmente porque as abordagens são muito diferentes.

Durante o decorrer desta dissertação, o dialeto utilizado será primordialmente aquele da comunidade de métodos ágeis, mas com o uso de alguns termos das outras áreas de estudo. Neste capítulo estão as principais definições e terminologias do tema testes de software e automação de testes para tornar a leitura do resto do trabalho mais produtiva.

3.1 Abordagens de Controle de Qualidade

Existem muitas abordagens de teste de software e de controle de qualidade, cada uma com um linguajar peculiar o que dificulta a interpretação e torna o estudo mais lento. Isto pode ser observado não só em livros e artigos como também na Internet, através das postagens em *blogs*, listas e fóruns de discussão e também em mini-debates, onde não há uma padronização rigorosa.

As abordagens de qualidade podem ser divididas em dois grupos: testes estáticos que são verificações do código-fonte e da documentação; e testes dinâmicos que incluem verificações do software em tempo de execução. Nas abordagens dos testes estáticos estão as práticas de revisões de código [149], inspeção [63] e *walkthroughs* [26]. Já no grupo dos testes dinâmicos estão os testes manuais e automatizados, que exercitam o software em tempo de execução. A seguir há a uma definição simplificada de cada uma das abordagens mais comuns das comunidades tradicionais e ágil de desenvolvimento de software.

Abordagens tradicionais

Os processos tradicionais de gerenciamento de software propõem diversas técnicas e artefatos para aumentar a qualidade do produto final, entre eles estão documentos de requisitos e práticas como Revisão, Inspeção e Testes Manuais. Do estudo destas técnicas podemos tirar conhecimento para melhorar a qualidade dos testes automatizados.

Revisão: É a prática da averiguação minuciosa em busca de erros de artefatos produzidos pelo desenvolvimento de software, como código-fonte, e documentação [72]. Existem revisões técnicas, de código-fonte e *walkthroughs* que são revisões realizadas em pares ou em grupos de pessoas [151, 26].

Inspeção: Proposta por Tom Gilb, inspeção é um tipo específico de revisão que visa planejar casos de testes e fazer uma revisão formal da especificação e de documentos para capturar erros, ambiguidades e dúvidas antes da implementação [63, 137].

Testes Manuais: São realizados por seres humanos que utilizam o sistema em busca de não conformidades e de aspectos de qualidade que podem ser melhorados [100].

Abordagens ágeis

Os métodos ágeis propõem novas abordagens de controle de qualidade no desenvolvimento de software, todas relativamente recentes, mas com fundamentação prática, teórica e empírica. Essas abordagens, incluindo testes automatizados, possuem propósitos distintos para solucionar problemas diferentes e recorrentes. A seguir são destacadas três destas abordagens.

Comunicação Efetiva: Métodos ágeis substituem especificações rigorosas e detalhadas pela prática de comunicação frequente e efetiva, idealmente realizada diretamente com o cliente ou com usuários finais do produto para esclarecer dúvidas em relação às histórias (requisitos). Essa abordagem depende de acordos e contratos específicos entre times de desenvolvimento e clientes para que os trabalhos sejam realizados sem desentendimentos [3, 40, 1].

Programação em Pares: É a prática onde duas pessoas participam ativamente do desenvolvimento de uma mesma funcionalidade ao mesmo tempo. A programação em pares tem como principal objetivo a revisão do código-fonte em tempo de implementação, realizada através da redundância de ideias e da observação de dois programadores distintos [146, 39, 140].

Testes Automatizados: É a abordagem de criar *scripts* ou programas simples de computador que exercitam o sistema em teste, capturam os efeitos colaterais e fazem verificações, tudo isso automaticamente e dinamicamente [99].

3.2 Termos e Siglas

Existem muitas convenções e padronizações para definir certos termos de testes de software, mas para alguns desses termos a padronização não é seguida rigidamente por todas as comunidades. Alguns deles são usados rotineiramente em nossas conversas diárias com significados diferentes dependendo do contexto, o que dificulta a fixação dos padrões e consequentemente a interpretação dos estudos técnicos. Abaixo está a descrição de alguns desses termos e algumas observações referente à terminologia utilizada por este trabalho.

Engano/Defeito/Falha/Erro/Não conformidade: Existem padrões que diferenciam esses termos [111, 114], onde cada um possui um significado específico. No entanto, este texto interpreta todos como sinônimos, e para distinguir os diferentes tipos de erros é utilizado explicitamente um texto adicional autoexplicativo, por exemplo: "Erro de distração do programador." Apesar do termo ficar mais longo, evita problemas de ambiguidades e também não fere convenções definidas por ferramentas e outras comunidades.

Verificação: Atividade que verifica se o produto está sendo desenvolvido corretamente [29]. A maioria dos tipos de testes é de verificação, tais como os testes de unidade, de interface, de sanidade entre outros.

Validação: Tarefa para avaliar se o software que está sendo desenvolvido é o esperado, isto é, se atende à especificação [29]. Os testes de aceitação, ou testes do cliente, apontam para os desenvolvedores quais os requisitos que devem ser implementados e quais cenários devem ser satisfeitos, facilitando a validação do sistema por parte do cliente e indicando para os programadores quando uma interação do desenvolvimento foi finalizada.

Testes Caixa Branca ou Caixa Transparente ou Caixa de Vidro: Também conhecido como teste estrutural, são os testes que fazem verificações baseadas na implementação [95].

Testes Caixa Preta: Também conhecido como testes funcionais, são aqueles que verificam funcionalidades sem conhecer a implementação [21].

Testes Caixa Cinza: É uma mistura dos testes caixa branca e preta, isto é, envolve os dois conceitos dentro de um mesmo cenário de verificação.

Erros e Testes de Regressão: Quando um erro é identificado em um trecho de código que estava funcionando corretamente em versões anteriores do sistema, dizemos que é um erro de regressão, isto é, um erro que foi adicionado durante alguma manutenção. Pensando em verificações manuais, dizemos que testes de regressão são aqueles que buscam encontrar este tipo de erro. Contudo, com testes automatizados esses termos são raramente utilizados, pois todos os testes passam a ser testes de regressão, já que todos ajudam a evitar esse tipo de erro.

Testes de Correção: São tipos de verificações que buscam erros no sistema, tais como os testes de unidade, integração, interface de usuário e aceitação. O teste de longevidade também é verifica a correção, mas é específico para verificar erros que só tendem a ocorrer depois de um certo tempo de execução do sistema.

Versões Alfa e Beta: Quando uma versão do software está finalizada e aparentemente estável, dizemos que sua versão é alfa, pois ele é liberado para uso por um grupo específico de usuários de homologação. Já o software que está em ambiente de produção real e permite que qualquer usuário o utilize para testes, dizemos que a sua versão é beta.

Caso de Falso Positivo: Caso de teste que teve sucesso apesar do que está sendo verificado conter erros. Normalmente isso ocorre por uma falha do teste que não fez todas as verificações esperadas, mas também pode ser por erros no próprio código da bateria de testes.

Caso de Falso Negativo: Caso de teste que falha apesar do que está sendo verificado estar correto. Qualquer erro de implementação nos testes pode criar este cenário.

Padrões: Descrevem soluções para problemas recorrentes no desenvolvimento de sistemas de software. Este termo surgiu dos Padrões de Projeto (*Design Patterns*) de sistemas orientados a objetos [61]. Também existem padrões relacionados com testes automatizados [99], que são soluções comuns para organizar os casos de testes, para criar verificações mais robustas etc.

Antipadrões: Antipadrões são erros recorrentes, comumente realizados em diferentes contextos. Um Antipadrão descreve o erro e formas de evitá-lo.

Indícios de Problemas: Também conhecido como "cheiros", são características do software, do código-fonte ou até mesmo da forma de desenvolvimento que dão indícios que alguma solução equivocada foi utilizada.

Testabilidade: É uma característica que indica o quão fácil de ser testado é um sistema. Um sistema com alta testabilidade é fácil de ser testado.

Os termos testes funcionais e estruturais (testes de caixa branca, preta e cinza) não serão utilizados neste trabalho, e sempre que necessários serão substituídos por termos que explicitem o objetivo do teste. Isso por dois motivos: não faz parte dos objetivos do trabalho estudar e comparar essas diferentes técnicas de testes pois já existem outros trabalhos sobre isso [95]; e também para evitar problemas de má interpretação durante a leitura desta dissertação ou durante o uso de ferramentas de testes que utilizam convenções diferentes. Por exemplo, o famoso arcabouço para desenvolvimento Web Ruby on Rails define testes funcionais como os de unidade referente a camada dos controladores, do padrão arquitetural MVC [32].

Será recorrente neste trabalho o uso de siglas que são habitualmente utilizadas pelas comunidades ágeis de testes automatizados. Algumas vezes, existe mais de uma sigla para a mesma ideia, ou duas muito parecidas que podem ser confundidas durante a leitura. Por isso, abaixo segue uma breve descrição daquelas mais comuns, sendo que algumas delas serão mais aprofundadas pelo trabalho enquanto outras só serão citadas.

SUT *System Under Test*, ou Sistema em Teste: É a aplicação que está sendo exercitada e verificada pelos testes automatizados.

AUT *Application Under Test*, ou Aplicação em Teste: Análogo SUT.

DDD *Domain-Driven Design*, ou *Design* Dirigido pelo Domínio: simplificadamente, é uma abordagem de desenvolvimento de software que propõe que o *design* do sistema seja fortemente conectado ao modelo de negócios, não simplesmente em arquiteturas definidas exclusivamente por desenvolvedores ou por ferramentas [8].

TAD *Test After Development*, ou Testes Após a Implementação: Técnica de escrever testes automatizados depois da implementação (Seção 9.1).

TFD *Test First Development*, ou Testes Antes da Implementação ou Testes a priori: Técnica de escrever testes automatizados antes da implementação (Seção 9.2).

TDD *Test-Driven Development*, ou Desenvolvimento Dirigido por Testes: Técnica de desenvolvimento baseada em iterações curtas do ciclo: escrita de um teste de unidade, implementação da mínima porção de código de produção que seja suficiente para o teste passar e, por último, refatoração do código caso seja necessário. Apesar de os testes serem escritos antes da implementação, esta técnica não é o mesmo que TFD (Seção 9.3).

BDD *Behavior-Driven Development*, ou Desenvolvimento Dirigido por Comportamento: É um aprimoramento da linguagem de TDD para criar uma linguagem ubíqua entre cliente e equipe de desenvolvimento, que segue o mesmo princípio dos testes de aceitação. BDD sugere o uso dos conceitos de DDD para substituir o vocabulário tradicional de testes de TDD por um vocabulário que seja próximo de uma especificação. Dessa forma, BDD pode ser utilizado tanto para os testes de unidade quanto para os testes de aceitação (Seção 9.4).

ATDD *Acceptance-Test Driven Development*, ou Desenvolvimento Dirigido por Testes de Aceitação: Técnica que sugere que o desenvolvimento siga um ciclo semelhante ao de TDD, mas que envolve os testes de aceitação e os de unidade. O ciclo sugere que sejam escritos primeiramente os testes de aceitação de uma história (conjunto de requisitos), que guiarão a implementação com TDD dos primeiros testes de unidade para a história correspondente. Após os testes de unidade estarem finalizados, deve ser feito os ajustes finais do teste de aceitação e então executá-los. Quando o teste de aceitação obtém sucesso, a história pode ser finalizada. Dessa forma, os testes de aceitação também são indicadores de quando uma história foi finalizada.

STDD *Story Test-Driven Development*, ou Desenvolvimento Dirigido por Testes da História: Mesmo que ATDD e que *Customer Test Driven Development*.

TDDD *Test-Driven Database Design*, ou Modelagem do Banco de Dados Dirigido por Testes: A ideia é aplicar as práticas de TDD na criação de um esquema de banco de dados [2].

CT *Continuous Testing*: É a prática de executar os testes automatizados continuamente, através de um programa que detecta qualquer alteração de código [124]. Ela foi originada a partir de TDD, que requer que os testes sejam executados muitas vezes em poucos minutos. Uma ferramenta de CT fica observando os arquivos de código-fonte do sistema e dos testes, se algum deles for modificado e salvo, a ferramenta automaticamente executa uma bateria de testes que ela julgar pertinente. Este processo diminui a necessidade de intervenção humana no manuseio de ferramentas e acelera a obtenção de *feedback*.

3.3 Tipos de Testes Automatizados

Por causa da popularização dos computadores e da Internet, o número de usuários de sistemas de software está crescendo, incluindo o número de usuários crianças, idosos e especiais. Não obstante, a sociedade está cada vez mais dependente dos sistemas de software, pois eles estão sendo utilizados cada vez mais para gerenciar processos importantes. Por isso, a exigência de qualidade cresce a cada dia e algumas falhas de software estão cada vez mais em evidência, como problemas de usabilidade, segurança e desempenho, além de diversos outros que podem ser desde enganos simples por desatenção do programador até mau entendimento dos requisitos.

Para muitos tipos de erros existem testes específicos, que fazem verificações seguindo certas premissas e padrões. Por isso, é fundamental organizar os casos de teste por tipo devido a várias razões: 1) facilita a manutenção dos testes e a adição de novos cenários para correção de erros; 2) eles podem utilizar ferramentas próprias e seguir padronizações diferentes; 3) o tempo de execução pode ser variado, então baterias de testes lentas não afetarão o *feedback* rápido que as baterias velozes proporcionam; 4) facilita a coleta das métricas por tipo de teste, que pode ser útil para identificar pontos de verificação que precisam de mais esforço.

Cada tipo de teste também possui ferramentas especializadas, justamente para facilitar a escrita e tornar o código mais enxuto e legível, logo, mais fácil de manter. Um erro comum é exercitar ou utilizar uma ferramenta de um tipo de teste específico para objetivos que não são de sua responsabilidade natural, o que pode levar à criação de testes de baixa qualidade. Portanto, é necessário conhecer o leque de opções de ferramentas para optar pelas mais apropriadas para o que se está querendo verificar.

Também é importante notar que alguns tipos de testes só fazem sentido com o auxílio da automação, devido principalmente a sua complexidade de implementar ou de executar os testes, tais como os de desempenho, carga e longevidade. Os testes de unidade também são mais interessantes quando automatizados, porque, apesar da fácil implementação e execução, eles geralmente não possuem interface de usuário intuitiva para possibilitar os testes manuais. Nas seções seguintes serão descritos em mais detalhes alguns desses tipos de testes assim como será abordado o tema de automação de testes para cada um deles. No final da seção também será apresentado uma figura e uma tabela criadas pelo autor com o objetivo de agrupar de forma resumida algumas das principais características de cada tipo de teste.

3.3.1 Teste de Unidade

Teste de correção responsável por testar os menores trechos de código de um sistema que possui um comportamento definido e nomeado. Normalmente, ele é associado a funções para linguagens procedimentais e métodos em linguagens orientadas a objetos.

```
# Referência do PyUnit
import unittest

# Referência do sistema em teste
from mathutils import is_primo

class PrimoNaturalTests(unittest.TestCase):

  def test_numeros_0_e_1_nao_sao_primos_por_convencao(self):
    self.assertFalse(is_primo(0))
    self.assertFalse(is_primo(1))

  def test_numero_2_e_primo(self):
    self.assertTrue(is_primo(2))

  def test_numero_par_diferente_de_2_nao_eh_primo(self):
    self.assertFalse(is_primo(4))
    self.assertFalse(is_primo(6))
    self.assertFalse(is_primo(1000))

  def test_numero_com_apenas_2_divisores_eh_primo(self):
    self.assertTrue(is_primo(3))
    self.assertTrue(is_primo(5))
    self.assertTrue(is_primo(7))

  def test_numero_com_mais_de_2_divisores_nao_eh_primo(self):
    self.assertFalse(is_primo(9))
    self.assertFalse(is_primo(15))
    self.assertFalse(is_primo(21))
```

Figura 3.1: Exemplo de teste de unidade.

A Figura 3.1 apresenta um exemplo de teste de unidade em Python com auxílio da ferramenta `unittest`, que já vem inclusa na biblioteca padrão da linguagem. Neste exemplo, há diversos casos de testes (métodos com nomes iniciados com a palavra test) encapsulados em uma classe `PrimoNaturalTests` para um método chamado `is_primo` que devolve um valor booleano, indicando se o número natural é primo. Os métodos `assertTrue` e `assertFalse` são herdados de `unittest.TestCase`, que é a classe base para a escrita de testes com essa ferramenta. Na execução dessa classe, se o teste espera que o método `is_primo` devolva um valor verdadeiro (`self.assertTrue`), mas o método devolve um valor falso, então é lançado uma exceção automaticamente que é armazenada por objetos de *log*, contendo qual teste falhou e o motivo. Se o valor esperado é obtido, então o *log* armazena que o caso de teste foi executado com sucesso.

3.3.2 Teste de Integração

Teste de integração é uma denominação ampla que representa a busca de erros de relacionamento entre quaisquer módulos de um software, incluindo desde a integração de pequenas unidades até a integração de bibliotecas das quais um sistema depende, servidores e gerenciadores de banco de dados. Um trecho ou módulo do sistema pode estar completamente correto, mas não há garantias de que as camadas superiores, que fazem chamadas a esse trecho de código, estão implementadas corretamente.

Os erros de integração são frequentes, principalmente quando vamos utilizar bibliotecas de terceiros que não conhecemos por completo. Além disso, quanto mais complexa for a arquitetura do sistema, maior a chance de existirem erros de integração, já que aumenta a quantidade de troca de mensagens entre os módulos. Nos sistemas mal modelados, a situação fica ainda mais crítica. Falhas de *design* como intimidade inapropriada entre objetos e o excesso de responsabilidades em um módulo do sistema [59] tornam o software difícil de entender e manter, portanto ele fica muito suscetível a adição de erros de integração.

Existem várias ferramentas que fornecem utensílios para facilitar a escrita dos testes de integração. O Maven, ferramenta de gerenciamento de projetos Java, disponibiliza uma arquitetura padronizada para armazenar os testes de integração. Já o arcabouço Grails, utilizado para desenvolvimento Web com Groovy, fornece um diretório para os testes de integração onde automaticamente os testes são carregados com os métodos dinâmicos de acesso ao banco de dados, o que facilita os testes de *queries* e mapeamentos objeto-relacional, que requerem a integração dos gerenciadores de banco de dados.

Na Parte II serão discutidos dois subtipos de testes de integração: os testes com persistência de dados (Capítulo 7), que podem depender de gerenciadores de banco de dados e de sistemas de arquivos; e testes de interface de usuário (Capítulo 8), os quais podem envolver arcabouços de interfaces gráficas e Web.

3.3.3 Teste de Interface de Usuário

A interface de usuário (*Graphical User Interface* (GUI) ou *Web User Interface* (WUI)) é um módulo peculiar de um sistema de software, pois é o único que mantém contato direto e constante com os usuários finais do sistema e é a partir dela que o usuário normalmente julga se o software é de qualidade. Por isso, qualquer não conformidade dessa camada pode ofuscar todo interior do software, que pode ser de boa qualidade.

Os testes automatizados de interface verificam a correção por meio da simulação de eventos de usuários, como se uma pessoa estivesse controlando dispositivos como *mouse* e teclado. A partir dos efeitos colaterais dos eventos, são feitas verificações na interface e em outras camadas para se certificar que a interface está funcionando apropriadamente.

A Figura 3.2 contém um exemplo de teste de interface Web, uma página que contém um formulário de login. O exemplo é escrito em Java e utiliza as ferramentas TestNG para definir os casos de testes (`Test`), o JUnit para fazer as verificações (`assertEquals`) e o WebDriver para criação dos eventos de

usuário. `FirefoxDriver` cria um *driver* que representa o navegador, e é ele que cria os eventos de usuário, tais como cliques do *mouse* (linha 17) e digitação de teclas do teclado (linhas 15 e 16). A linha 18 faz uma verificação para certificar se a interface foi atualizada corretamente após a autenticação.

```
1  // referências do WebDriver
2  import org.openqa.selenium.By;
3  import org.openqa.selenium.WebDriver;
4  // referências do TestNG
5  import org.testng.annotations.Test;
6  // referências do JUnit
7  import static org.junit.Assert.assertEquals;
8
9  public class LoginInterfaceTests {
10
11   @Test
12   public void loginComSucessoHabilitaLinkDeLogout() {
13     WebDriver browser = new FirefoxDriver();
14     browser.get("http://umsite.net");
15     browser.findElement(By.id("username_field")).sendKeys("fulano");
16     browser.findElement(By.id("password_field")).sendKeys("abracadabra");
17     browser.findElement(By.name("login_button")).click();
18     assertEquals("Logout", browser.findElement(By.id("logout_link")).getText());
19   }
20
21 }
```

Figura 3.2: Exemplo de teste de interface Web com Java.

Como a interface de usuário é o "cartão de visita" do programa, não basta apenas ela estar correta, é necessário que ela seja organizada, atraente e que permita que diferentes grupos de usuários consigam utilizá-la, por isso é importante os testes de leiaute, usabilidade e acessibilidade.

Teste de Leiaute

Os testes de leiaute buscam avaliar a beleza da interface, assim como verificar a presença de erros após a renderização, que são erros difíceis de se identificar com testes comuns de interface. Por exemplo, um teste de interface básico pode facilmente verificar que um módulo da interface foi carregado corretamente, mas não é garantido que o módulo está bem desenhado ou mesmo visível para o usuário final.

Além disso, o leiaute pode variar significativamente entre diversas plataformas, tais como em diferentes gerenciadores de janelas e navegadores Web. Cada plataforma pode possuir suas particularidades, já que as funcionalidades e algoritmos de renderização podem variar consideravelmente, por isso uma bateria de testes de leiaute também é importante para verificar a portabilidade do sistema.

Ainda há uma carência de ferramentas que facilitem a automação para estes tipos de testes, principalmente porque a grande subjetividade na interpretação do que é certo ou errado dificulta a criação de ferramentas completas e flexíveis. No entanto, muito pode ser feito com auxílio das ferramentas dos testes de interface, nem que seja para facilitar os testes manuais.

A Figura 3.3 apresenta um exemplo de teste de leiaute semiautomatizado, escrito em Java com auxílio das ferramentas Selenium e Util4Selenium. O *script* automatizado de testes contém os métodos `visitaPaginaPrincipal`, `visitaPaginaInformacoes` e `visitaPaginaResultadosDeBusca` que têm instruções da ferramenta Selenium que enviam comandos para o navegador para iteragir com determinadas páginas do sistema.

```java
// Classe base pra todas clases de teste de layout

// referências do TestNG
import org.testng.annotations.AfterSuite;
import org.testng.annotations.BeforeSuite;
// referências do Util4Selenium
import utilities.util4selenium.annotations.Screenshot;
import utilities.util4selenium.annotations.Screenshot.ScreenshotPolicy;
// referências do Selenium-RC/Java
import com.thoughtworks.selenium.DefaultSelenium;
import com.thoughtworks.selenium.Selenium;

@Screenshot(policy = ScreenshotPolicy.ALWAYS)
public class SeleniumTestCase {
  public Selenium browser;

  @BeforeSuite
  public void abreNavegador() {
    browser = new DefaultSelenium("localhost", 4444, "*chrome", "http://localhost
        :8080");
    browser.start();
  }

  @AfterSuite
  public void fechaNavegador() {
    browser.stop();
  }
}

// Classe de teste de layout

import org.testng.annotations.Test;

public class PaginasPublicasLayoutTests extends SeleniumTestCase {

  @Test public void visitaPaginaPrincipal() {
    browser.open("/");
  }

  @Test public void visitaPaginaInformacoes() {
    browser.open("/about");
  }

  @Test public void visitaPaginaResultadosDeBusca() {
    browser.open("/search");
    browser.type("word", "open source");
    browser.submit("search_form");
  }

}
```

Figura 3.3: Exemplo de teste de leiaute Web com Java.

O resto do trabalho é realizado pela ferramenta Util4Selenium, através da anotação `Screenshot`, que captura a imagem da tela após o término da execução do método (a política `ScreenshotPolicy.ALWAYS` determina que sempre um *screenshot* será capturado, sem exceções).

No final da execução temos uma lista de imagens, que alguma pessoa pode observar visualmente em busca de falhas do leiaute, eliminando as tarefas de navegar entre as páginas e preencher formulários repetidamente para diferentes navegadores e a cada mudança de leiaute.

Usabilidade e Acessibilidade

Os testes de usabilidade verificam a facilidade de aprendizado e de uso de um sistema. O requisito básico para uma boa usabilidade é a correção do sistema e da interface mas, para uma melhor organização, a correção deve ser verificada em outras baterias de testes, como as de unidade e interface. Seguindo esse padrão organizacional, os testes de usabilidade devem verificar outras características, como organização, padronização e clareza da interface, assim como a documentação de ajuda para o usuário.

Para uma boa usabilidade, a organização da interface deve levar em consideração quais as funcionalidades mais importantes e as mais utilizadas, o tipo de programa e também o público alvo. Por isso, também é importante os testes de acessibilidade, que são um tipo de teste de usabilidade específico para usuários com alguma limitação da visão, audição ou motora. Geralmente, esse grupo de usuários é composto por idosos, deficientes ou pessoas com doenças específicas, tais como mal de Parkinson, síndrome do encarceramento etc. As limitações destes usuários os impedem de utilizar com êxito alguns dispositivos do computador, como o monitor, *mouse* e teclado.

Os testes de acessibilidade devem verificar se é possível utilizar o sistema com facilidade usando apenas alguns dos dispositivos. Eles podem verificar os tamanhos e cores e contrastes dos textos e das imagens, a existência de atalhos de teclado para os comandos, opções de áudio para componentes de CAPTCHA[1] etc.

Entretanto, apesar da automação também ser útil para verificar a usabilidade dos sistemas, muitas vezes eles precisam ser realizados manualmente por causa da grande dificuldade de automatizá-los. A relação custo-benefício da criação de algoritmos e heurísticas de verificação pode ser maior do que a repetição manual dos testes.

3.3.4 Teste de Aceitação

Também conhecido como teste funcional ou de história de usuário, são testes de correção e validação. Eles são idealmente especificados por clientes ou usuários finais do sistema para verificar se um módulo funciona como foi especificado [96, 87]. Por isso o termo "aceitação", pois ele verifica se o cliente aceita as funcionalidades que foram implementadas. Os testes de aceitação devem utilizar uma linguagem próxima da natural para evitar problemas de interpretação e de ambiguidades, e também devem ser facilmente conectados ao código do sistema para que os comandos e verificações sejam executados no sistema em teste [104].

Nas diretrizes da Programação eXtrema, uma história do cliente não é finalizada enquanto os testes de aceitação não certificarem que o sistema atende aos requisitos especificados. Sendo assim, os testes de aceitação não só são utilizados para identificar erros de programação e interpretação dos requisitos como também para identificar quando uma história foi finalizada.

[1] CAPTCHA ou *Completely Automated Public Turing test to tell Computers and Humans Apart*, é um teste de desafio cognitivo utilizado como ferramenta anti-spam. Na Internet é comum um componente que fornece um pequeno desafio como uma imagem ou um som que é facilmente decifrado por um humano, mas muito difícil de ser avaliado por um computador, assim é possível evitar que *scripts* automatizados indesejados utilizem o sistema.

3.3.5 Teste de Desempenho

Testes de desempenho executam trechos do sistema em teste e armazenam os tempos de duração obtidos, como um cronômetro. Os testes de desempenho não avaliam a complexidade computacional dos algoritmos, por isso os tempos obtidos estão intimamente relacionados à infraestrutura sobre a qual o teste está sendo executado, podendo variar drasticamente dependendo do hardware, da rede etc. [129, 51].

Os resultados dos testes de desempenho ajudam a identificar os gargalos que precisam de otimização para diminuir o tempo de resposta para o usuário [90]. A otimização pode ser feita através de mudanças em algoritmos e também pela adição de caches em pontos específicos do sistema, caso os algoritmos já sejam suficientemente rápidos.

3.3.6 Teste de Carga

O teste de carga exercita o sistema sob condições de uso intenso para avaliar se a infraestrutura é adequada para a expectativa de uso real do sistema [9, 10]. São criadas simulações com muitos usuários e requisições ao software realizadas simultaneamente ou dentro de um intervalo pequeno de tempo para medição de informações de desempenho. Dependendo do sistema e do que o teste quer avaliar, pode ser verificado o tempo de resposta de cada requisição ou informações relativas ao uso de recursos, como uso da CPU, cache, memória, espaço em disco e banda de rede. Também podem ser observados os programas que se relacionam com o sistema, como gerenciadores de banco de dados, máquinas virtuais e servidores de aplicação.

Após o término da simulação, é realizada a tarefa de interpretação subjetiva das informações coletadas, que pode ser feita manualmente ou através de *scripts* que seguem alguma heurística especializada para um contexto. Essas informações são úteis para identificar módulos do sistema que apresentem mau desempenho sob uso intenso, como, por exemplo, *queries* que são executadas repetidamente e que poderiam ser inseridas em um sistema de cache. As informações também podem indicar partes do hardware e da infraestrutura que são mais utilizadas pelo software, portanto quais são as mais importantes para a execução satisfatória do sistema.

Outra conclusão que pode ser obtida com os testes de carga é se o sistema e suas dependências são escaláveis. Se a vazão das requisições apresentar um aumento linear com o tempo, significa que o sistema é escalável, portanto, é possível melhorar o desempenho do sistema com o *upgrade* da infraestrutura. Se a vazão tiver um crescimento exponencial, ou seja, um crescimento relativamente rápido para poucos usuários e requisições, então é uma indicação que alguma configuração ou algoritmo precisa ser melhorado.

Uma ferramenta popular de testes de carga é a JMeter, que possibilita testar aplicações Web, bancos de dados e outros tipos de aplicações[2]. Com ela ainda é possível criar testes de estresse e longevidade, que serão descritos a seguir, além de testes de desempenho. Um exemplo pode ser visto no Apêndice A.

Teste de Estresse

Enquanto o teste de carga visa avaliar se a infraestrutura é adequada para as expectativas de uso do sistema, o teste de estresse, também conhecido como teste de carga máxima, visa descobrir os limites de uso da infraestrutura, isto é, qual a quantidade máxima de usuários e requisições que o sistema consegue atender corretamente e em um tempo aceitável. A análise dos resultados pode ser feita através de asserções, mas sempre é necessário uma última análise manual dos dados obtidos pelos testes.

Os valores limites obtidos da simulação de estresse são importantes para o gerenciamento e configuração do hardware e do software. Essa simulação aponta quais são os gargalos de hardware que precisam de *upgrade* e também orienta a configuração do hardware e do software para melhorar o desempenho

[2] O autor utilizou a ferramenta JMeter para realizar testes de carga no sistema Janus.

ou mesmo para criar barreiras que impeçam que a quantidade máxima de requisições extrapole um limite seguro. Por exemplo, podemos configurar um servidor de aplicações Web para barrar quantidades exageradas de requisições para impedir ataques de negação de serviço[3].

3.3.7 Teste de Longevidade

O teste de longevidade tem por objetivo encontrar erros que só são visíveis após um longo tempo de execução do sistema. Esse teste é importante porque o sistema pode se comportar de maneira errônea ou ineficiente após dias ou semanas de execução ininterrupta, mesmo que ele funcione corretamente sob uso intenso de usuários e requisições em um curto intervalo de tempo. Esses problemas são geralmente de cache, replicação, da execução de serviços agendados e, principalmente, de vazamento de memória.

A execução de serviços agendados e de replicação são muito suscetíveis a erros do ambiente, como configuração incorreta do hardware e do sistema operacional e também problemas de bloqueio e permissão em sistemas de arquivos. Além disso, é comum que uma infraestrutura de hardware seja compartilhada entre outros usuários e sistemas de software, os quais possuem comportamento desconhecido que pode afetar o sistema em teste, por isso é importante o teste de longevidade para verificar se existe em algum momento um impacto direto no desempenho e correção da aplicação.

Erros em sistema de cache também são mais facilmente observados com o passar do tempo. Os sistemas de cache podem ser configurados para serem atualizados depois de um longo período de tempo, por exemplo, uma vez ao dia ou depois de uma semana; dessa forma, só conseguiremos verificar se essa tarefa está sendo executada corretamente após este período. No entanto, esses problemas ainda podem ser verificados com o auxílio de testes de unidade e Objetos Dublês (vide Seção 6.2), o que não é trivial para os problemas de vazamento de memória.

Vazamento de memória é um problema comum a muitos sistemas de software, não só os que são implementados com linguagens de programação que exigem a criação de destrutores e que não possuem coleta de lixo automática (como C++), como também em linguagens mais modernas, como Java [30] e JavaScript. Mesmo um vazamento de memória aparentemente insignificante pode trazer problemas com o decorrer do tempo, porque a quantidade de memória gasta pode ser multiplicada pela quantidade de usuários e requisições que pode ser muito grande.

3.3.8 Testes de Segurança

Os testes de segurança servem para verificar se os dados ou funcionalidades confidenciais de um sistema estão protegidos de fraude ou de usuários não autorizados. A segurança de um software pode envolver aspectos de confidencialidade, integridade, autenticação, autorização, privacidade, entre outros, todos sujeitos a ter vulnerabilidades por erros de software ou de infraestrutura [145].

Existem diversas ferramentas especializadas em testes de segurança que simulam os ataques mais comuns de pessoas mal intencionadas. Contudo, as vulnerabilidades também podem ser verificadas com auxílio de ferramentas de testes de interface ou de unidade, que permitem simular o comportamento de usuários e inserir informações maliciosas no sistema. Entre os ataques que é possível simular estão os que se beneficiam da falha de conversão ou da fraca validação dos dados de entrada do sistema, assim como os ataques que injetam código malicioso nos servidores e causam estouro de memória.

3.4 Técnicas de Teste

Na maioria das vezes os testes de software não provam que o sistema está livre de defeitos, justamente porque os testes não conseguem cobrir a enorme quantidade de possibilidades de entrada e combinações

[3] Ataques de negação de serviço (*Denial of Service* ou simplesmente DoS) são feitos com o envio intenso de requisições para um mesmo servidor com o objetivo de estressar o sistema a tal ponto que o sistema pare de funcionar.

de resultados a que um software pode estar sujeito. Sendo assim, os testes são realizados até que o time, incluindo desenvolvedores e clientes, esteja satisfeito quanto à qualidade do sistema ou de um algoritmo. Após a realização dos testes, podem ser executadas outras tarefas para certificar que o software atende bem às necessidades, como análises formais e matemáticas ou mesmo a liberação do sistema em versões alfa e beta para testes em ambiente real com usuários reais.

A regra geral durante a criação dos testes automatizados é pensar em partições de domínio ou classes de equivalência de um algoritmo, isto é, tentar identificar tanto os casos rotineiros, como as situações especiais e excepcionais, e, então, criar um caso de teste para cada uma delas. No entanto, em alguns casos isso não basta para dar a confiança de que está tudo implementado corretamente, por isso existem outras técnicas de testes que podem ser utilizadas para complementá-los e reforçar a confiabilidade do sistema, aumentando as chances de encontrar erros durante a implementação do código, como através de testes aleatórios, fumaça e de sanidade.

Essas técnicas, as quais também são definidas como tipos de testes, não precisam possuir baterias exclusivas de casos de testes. Por exemplo, não é necessário organizar uma bateria com apenas casos de testes de sanidade. Se esses testes seguirem as padronizações e os princípios da bateria, tais como desempenho e uso de ferramentas específicas, então eles podem complementar as baterias de testes já existentes, como a de testes de unidade.

3.4.1 Testes Aleatórios (*Random Tests*)

Quando não é possível avaliar todas as combinações de entrada e saída de um algoritmo e há insegurança quanto à sua correção, pode-se realizar testes com dados aleatórios para tentar encontrar cenários que não foram previamente identificados. Para criação dos testes aleatórios é necessário que haja um gerador dos dados de entrada e também uma lógica simples de verificação de resultados, que não implemente o algoritmo identicamente e que faça comparações flexíveis para que os testes sejam consistentes, ou seja, a aleatoriedade dos dados não devem tornar os testes intermitentes.

Na Figura 3.4, há um exemplo de teste aleatório para as funções seno e cosseno da biblioteca padrão da linguagem Python. É impossível provar que algoritmos implementados para calcular os valores de seno e cosseno estão corretos com poucos testes e sem uma formulação matemática. Portanto, podemos criar um teste aleatório que faça verificações pouco específicas, mas que atinja um grande número de casos a cada vez que o teste é executado. Nesse exemplo, é utilizado uma propriedade trigonométrica como artifício para verificação.

```
1  import unittest
2  import random
3  import math
4
5  class SenoCossenoTests(unittest.TestCase):
6
7    def test_aleatorio_para_seno_e_cosseno(self):
8      for i in xrange(0, 100):
9          numero_aleatorio = random.randint(100, 100000)
10         resultado_seno = math.sin(numero_aleatorio)
11         resultado_cosseno = math.cos(numero_aleatorio)
12         # verifica a correção por meio de uma propriedade trigonométrica
13         resultado = math.pow(resultado_seno, 2) + math.pow(resultado_cosseno, 2)
14         self.assertAlmostEqual(1, resultado)
```

Figura 3.4: Exemplo de teste aleatório.

É importante notar que os testes aleatórios não são preditivos e podem possuir um comportamento

intermitente quando há falhas no sistema. Essas duas características são antipadrões de testes automatizados em testes rotineiros, por isso é importante documentar com nomes claros as variáveis e os métodos de casos de testes, destacando que o teste é aleatório, pois isso facilita a identificação e compreensão dos erros nos relatórios dos testes, evitando gastar tempo excessivo com depuração.

3.4.2 Teste de Fumaça (*Smoke Tests*)

Testes de fumaça são verificações simples para busca de grandes erros, que geralmente afetam muitas funcionalidades e são fáceis de se detectar. Eles são geralmente executados antes de outras baterias de testes mais especializadas, pois, se grandes erros forem encontrados, provavelmente uma grande parcela dos outros cenários de verificações também irá falhar. Por isso, os testes de fumaça podem economizar tempo gasto com a execução de outras baterias de testes e com a identificação das causas dos problemas.

O nome deste tipo de teste deve-se aos testes de fumaça realizados em *hardware*. Se uma placa de hardware não pega fogo ou não solta fumaça durante um teste, então é um primeiro indício que a placa pode ter sido corretamente produzida. Esse termo também é utilizado em outras áreas do conhecimento, por exemplo, existem testes de fumaça para verificar a vedação de encanamentos e de instrumentos de sopro. É pressionado uma fumaça atóxica dentro dos objetos para verificar se existem rachaduras.

Testes de fumaça são geralmente pouco específicos e muito abrangentes, além de rápidos de implementar e de executar. Eles são úteis para capturar erros grandes, principalmente alguns erros de configuração e de ambiente que são facilmente detectados após a instalação do software.

A Figura 3.5 mostra um exemplo de teste de fumaça de interface Web, utilizando a ferramenta Selenium [4]. Os testes navegam por páginas do sistema e verificam se elas foram encontradas, isto é, não foi exibido o erro 404 do protocolo HTTP[5].

```
1  // referências do TestNG
2  import org.testng.annotations.Test;
3
4  public class PaginasEncontradasTests extends SeleniumTestCase {
5
6    @Test public void visitaPaginaPrincipal() {
7      selenium.open("/");
8      assertFalse(selenium.isTextPresent("404"));
9    }
10
11   @Test public void visitaPaginaInformacoes() {
12     selenium.open("/about");
13     assertFalse(selenium.isTextPresent("404"));
14   }
15
16 }
```

Figura 3.5: Exemplo de teste de fumaça.

3.4.3 Teste de Sanidade (*Sanity Tests*)

Testes de sanidade possuem as mesmas características dos testes de fumaça, isto é, são rápidos, abrangentes e pouco específicos. No entanto, eles são geralmente desenvolvidos para testes de um mó-

[4] O sistema Janus da Pró-Reitoria de Pós-Graduação da USP, do qual o autor participou do desenvolvimento, possui uma bateria de testes de fumaça para verificar a existência de grandes erros no ambiente de produção.

[5] Hypertext Transfer Protocol (HTTP): `http://en.wikipedia.org/wiki/Hypertext\_Transfer\_Protocol`

dulo específico. É comum utilizar testes de sanidade para verificar algoritmos essencialmente matemáticos, onde é possível utilizar certas propriedades matemáticas nas verificações.

```
1  import unittest
2  import math
3
4  class RaizQuadradaTests(unittest.TestCase):
5
6    def test_sanidade_para_raiz_quadrada(self):
7      # 10^4 < 123456789 < 10^5
8      value = math.sqrt(123456789)
9      self.assertTrue(value > (10**4))
10     self.assertTrue(value < (10**5))
```

Figura 3.6: Exemplo de teste de sanidade.

O exemplo da Figura 3.6 apresenta um teste simples de sanidade para um método que calcula a raiz quadrada de um número (`math.sqrt()`). Por alguma razão[6] não sabemos a raiz de 123456789, então utilizamos uma propriedade matemática para fazer as verificações: se um número natural **A** é maior que um número natural **B**, então a raiz quadrada de **A** é maior que a raiz quadrada de **B**. As duas verificações feitas são bem abrangentes e não fornecem um indício forte que o método está correto, apenas informa que não foi cometido um erro enorme para este caso de teste.

O exemplo de teste aleatório citado na seção anterior (Figura 3.4) pode ser convertido em um teste de sanidade, pois ele também é um teste que faz verificações abrangentes através do uso de propriedades matemáticas. Basta trocar a geração aleatória dos dados de entrada por valores definidos, como é mostrado na Figura 3.7. Nesse caso, a vantagem sobre o teste aleatório é que ele é um teste reprodutível e com menos possibilidade de se tornar intermitente, pois há o total controle dos dados de entrada.

```
1  import unittest
2  import random
3  import math
4
5  class SenoCossenoTests(unittest.TestCase):
6
7    def test_aleatorio_para_seno_e_cosseno(self):
8      for numero_conhecido in xrange(0, 100):
9          resultado_seno = math.sin(numero_conhecido)
10         resultado_cosseno = math.cos(numero_conhecido)
11         # verifica a correção por meio de uma propriedade trigonométrica
12         resultado = math.pow(resultado_seno, 2) + math.pow(resultado_cosseno, 2)
13         self.assertAlmostEqual(1, resultado)
```

Figura 3.7: Exemplo de conversão do teste aleatório para teste de sanidade.

3.5 Considerações Finais

Um sistema de software está sujeito a uma infinidade de tipos de erros diferentes, sendo que muitos deles são recorrentes no desenvolvimento de software. A verificação destes erros deve ser organizada

[6]Poderíamos calcular em uma calculadora o resultado da raiz de 123456789 para escrever um teste mais específico, mas é só um exemplo simples para apresentar o conceito.

e planejada através de baterias de testes especializadas e padronizadas para que os testes sejam mais eficazes e para que a execução dos testes forneça o *feedback* apropriado.

A Figura 3.8, criada pelo autor, organiza os tipos de teste de acordo com sua definição e com as discussões feitas nas seções anteriores. Ela apenas mostra, de uma forma resumida, o relacionamento dos tipos de teste. Cada círculo representa um conjunto e as intersecções representam se existem relacionamentos entre os tipos de teste. Caso um círculo esteja inserido completamente em outro, significa que são subtipos de tipo de teste representado pelo círculo maior. Por exemplo, teste de carga não possui qualquer relacionamento com teste de unidade; e teste de estresse é um subtipo de teste de carga.

Já a Tabela 3.1, também criada pelo autor, aponta algumas características de cada tipo de teste para comparação e para ajudar no gerenciamento, assim como adaptar e organizar baterias já existentes que precisam ser refatoradas. No entanto, a tabela está organizada de modo geral, pois as características de cada tipo de teste podem variar de acordo com o contexto e com as ferramentas utilizadas.

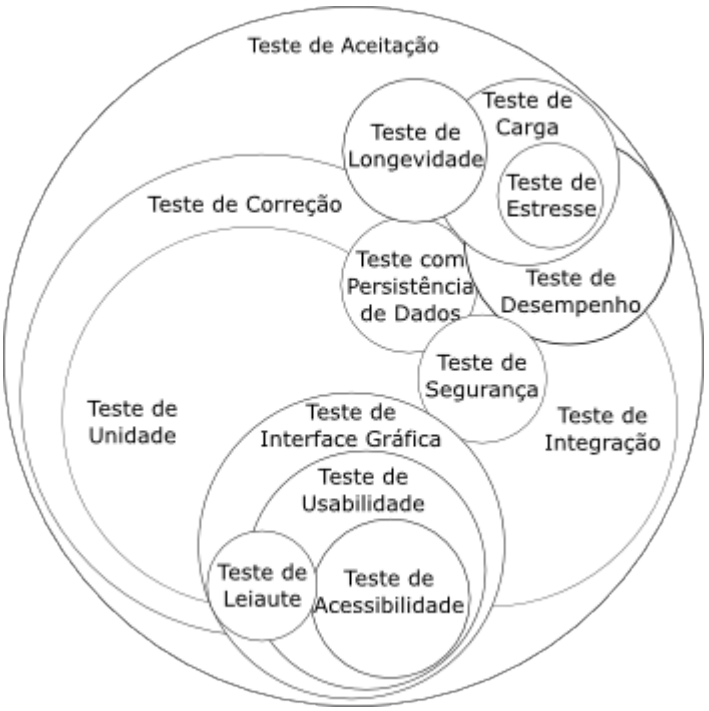


Figura 3.8: Tipos de testes de software.

Característica / Tipo de Teste	Específico	Abrangente	Isolado	Integrado	Lento
Unidade	o		o		
Integração		o		o	
Interface de Usuário	o	o		o	o
Aceitação		o		o	o
Desempenho	o			o	o
Carga		o		o	o
Estresse		o		o	o
Longevidade		o		o	o
Segurança	o			o	

Tabela 3.1: Características dos tipos de testes.

Capítulo 4

O Processo de Automação de Testes

Até o momento vimos que alguns dos problemas dos testes automatizados são intrínsicos de cada tipo de teste. Entretanto, outras dificuldades podem ser causadas simplesmente pelo desconhecimento de boas práticas de organização de um projeto com automação de testes. Por isso, esse capítulo discute e recomenda formas de se trabalhar para aumentar a produtividade.

4.1 Visão Geral

Os testes automatizados afetam diretamente a qualidade dos sistemas de software, portanto agregam valor ao produto final, mesmo que os artefatos adicionais produzidos não sejam visíveis para os usuários finais do sistema. Mas, como qualquer artefato, o código-fonte dos testes automatizados e os documentos e relatórios gerados requerem qualidade e capricho para que a evolução e manutenção seja viável, com custo de trabalho linear ou mesmo constante, ou seja, que não aumente radicalmente sua complexidade à medida que o projeto evolui.

É necessário que também haja preocupação com a qualidade do código-fonte dos testes, pois eles também estão sujeitos à problemas similares que podem ocorrer com o software em teste. Consequentemente, diversas tarefas de manutenção de código do sistema também são aplicadas ao dos testes, tais como evolução, refatoração e otimização [143]. Tudo isso para obter e manter a correção, legibilidade, modularização coesa e desempenho, que são algumas das características básicas para uma boa bateria de testes automatizados [76].

Parece não fazer sentido adicionar um artefato sujeito a erros para controlar a qualidade do sistema, e também dedicar uma grande porcentagem do trabalho para algo que também precisa de um esforço para garantir qualidade e que não será explicitamente visível para os usuários finais. Entretanto, o principal aspecto que traz coerência para a prática de testes automatizados é a simplicidade: testes automatizados devem possuir um código-fonte trivial, que seja fácil de implementar e de entender. Dessa forma, a chance de introduzir erros é relativamente pequena. Outro aspecto complementar é que um erro introduzido nos testes geralmente é fácil de ser identificado em tempo de implementação, pois a execução do teste tende a falhar, se ele não falhar é porque houve uma replicação do erro no código do sistema. Essa redundância de trabalho entre código do sistema e código dos testes cria duas frentes de busca de erros e traz mais segurança para escrita do sistema e dos testes.

4.2 Quem Deve Implementar

A criação dos testes automatizados pode exigir conhecimento de programação, de testes de software, da linguagem de negócio e de ferramentas específicas. Portanto, podem ser necessário profissionais qualificados para a realização dessa tarefa. Entre os profissionais com estas qualificações, geralmente

estão os desenvolvedores, testadores ou analistas de qualidade, mas também clientes e usuários finais podem colaborar com esta tarefa.

O que deve ser considerado em primeiro lugar é o tipo de teste a ser realizado (vide Figura 3.8), pois alguns deles requerem conhecimentos completamente específicos, entretanto, para todos eles é necessário um conhecimento geral de diversas áreas. Devido à abrangência de conhecimento necessário para a criação de testes, não é recomendado responsabilizar apenas um grupo de pessoas com conhecimentos específicos pela qualidade do sistema. Por isso, os métodos ágeis sugerem disseminar a preocupação pela qualidade entre todos os membros do time. Assim, todos devem trabalhar colaborativamente para agregar seus conhecimentos para o objetivo final que é comum a todos da equipe: produzir um sistema de software de qualidade.

No entanto, trabalhar colaborativamente não significa que todos os profissionais irão realizar tarefas semelhantes, apenas que deverá haver uma integração das ideias. Para que esta troca de informações seja efetiva, sempre é necessário uma boa comunicação, transparência e clareza das informações, que podem ser obtidas com a ajuda de reuniões frequentes, áreas de trabalho informativas e uso de metáforas comuns a todos membros da equipe.

De forma geral, os testes de unidade, mutação e de integração de módulos devem ser implementados com a ajuda principalmente de desenvolvedores e testadores. Os testes de interface gráfica (incluindo usabilidade, acessibilidade e leiaute) e de aceitação devem ter participação de todos os membros da equipe, principalmente dos clientes e dos usuários finais porque o sistema deve atender às suas necessidades apropriadamente. Já os testes de desempenho, carga, estresse, longevidade e segurança podem ser realizados por profissionais especializados, dado que as tarefas costumam ter uma certa independência e um padrão a ser seguido, mas também é fundamental a comunicação para concentrar os testes nas principais áreas do sistema.

4.3 Quando Implementar

Projetos que seguem metodologias tradicionais, derivadas do modelo de cascata, tendem a ter uma equipe de testadores especializados e analistas de qualidade que realizam testes manuais no fim de uma jornada de implementação. Como é sugerido pelos métodos ágeis, esta organização não é eficaz porque o *feedback* obtido dos testes não é facilmente e rapidamente aproveitado, pois, geralmente, é necessário um árduo trabalho de depuração e de reorganização do código.

Automatizar os testes no fim das iterações também não é a melhor estratégia, porque um dos principais benefícios dos testes automatizados, que é a segurança contra erros de regressão, não é aproveitado durante todo o desenvolvimento. Em suma, os testes finais passam a ter um custo de implementação maior que dos testes manuais e pode existir a compensação do tempo gasto com os benefícios da automação.

Para valer a pena, a automação dos testes, é necessário que eles sejam implementados o quanto antes para fornecer *feedback* em tempo de implementação, pois quanto mais cedo uma falha é detectada, mais fácil será corrigi-la. Fazer alterações pontuais nas funcionalidades que estão sendo implementadas é mais rápido porque estamos com grande parte do código-fonte em nossa memória recente, dispensando um estudo adicional para lembrar e entender o código já existente.

Contudo, é fundamental estabelecer as prioridades dos tipos de testes necessários, considerando o valor que será agregado ao produto final. A primeira premissa básica da qualidade de todo sistema de software é a aceitação do cliente, isto é, devemos implementar o que foi pedido. A segunda premissa básica é a correção, se o software não está correto, os resultados dos outros testes se tornam não confiáveis. Depois destas premissas estarem asseguradas podemos pensar na prioridade de outros aspectos de qualidade, como desempenho, segurança, beleza e usabilidade. Estas prioridades variam de sistema para sistema, pois devem ser consideradas detalhes das tecnologias utilizadas, o tipo de software, o contexto da aplicação e o público-alvo.

Considerando apenas as características fundamentais de qualidade (aceitação e correção), o ideal é criar os testes em fase de implementação, particularmente em alguns momentos especiais como antes mesmo da própria implementação do sistema, como é sugerido pelas técnicas de Testes *a Priori* e Desenvolvimento Dirigido por Testes que serão discutidas no Capítulo 9. De forma geral, é recomendado que os testes sejam escritos nas seguintes situações.

Antes de escrever o código: Evita que os testes sejam deixados de lado, favorece a escrita de testes simples e também influencia o *design* da aplicação para ser testável.

Durante a implementação: Os testes ajudam a refinar o *design* do sistema e até implementar algoritmos.

Quando um erro for encontrado: Se um erro foi encontrado quer dizer que o cenário não foi testado, então antes da correção pode ser criado um teste que o simule. Com esse teste teremos segurança que o problema não reaparecerá em futuras modificações do código.

Quando um teste de aceitação falha: É um erro de algum módulo que é propagado por entre as camadas do software, por isso é um indício de que falta um caso de teste de unidade ou de integração para algum módulo do sistema.

Quando um usuário ou cliente encontra um erro: Pode ser um sinal que estejam faltando testes de unidade e de aceitação. A criação desses testes é especialmente útil para evitar que ele reapareça, o que pode tornar a situação ainda mais constrangedora.

4.4 Onde Executar

Com os testes implementados, o próximo passo é definir onde eles serão executados. Cada tipo de teste possui restrições próprias relacionadas ao ambiente, mas o que todos têm em comum é que podem depender da flexibilidade e portabilidade das tecnologias utilizadas e também podem ser influenciados por fatores externos, sejam eles de software ou hardware.

Os testes de desempenho, carga, estresse e longevidade requerem idealmente que a infraestrutura seja a mais parecida com o ambiente real de produção, incluindo configurações de hardware, sistema operacional e também os processos que são regularmente executados na infraestrutura, isto porque os resultados obtidos estão diretamente relacionados com o ambiente. Os testes de segurança também dependem da infraestrutura, pois pode haver vulnerabilidades no ambiente que expõem módulos restritos do software, assim como os testes de usabilidade e leiaute que dependem do desempenho do sistema.

Já os testes de correção (unidade, integração, aceitação etc.) não devem possuir uma dependência tão rígida quanto ao tipo e configuração do hardware e da infraestrutura. Essa flexibilidade é necessária para que os testes sejam executados tanto em máquinas de desenvolvimento como em servidores de integração contínua, assim como em outros ambientes que sejam necessários.

Contudo, nem sempre é possível criar baterias de testes completamente compatíveis com diferentes tipos de ambientes. Ambientes compartilhados, controlados (de homologação ou de controle de qualidade) e isolados requerem mudanças significativas na implementação do código dos testes.

Ambiente compartilhado de testes é utilizado por diversos projetos e times de desenvolvimento, além de diferentes áreas de uma organização. Ele geralmente é semelhante ao ambiente real de produção para tornar o ambiente de teste o mais próximo do real. Devido ao alto custo destes ambientes, pode ser inviável possuir um por projeto, por isso a solução é compartilhar um entre todas as equipes.

A escolha destes ambientes é favorecida principalmente quando a infraestrutura é complexa de ser criada e configurada, pois minimiza a quantidade de ambientes que precisarão de manutenção. Por exemplo, é trabalhoso lidar com ambientes onde há um grande conjunto de servidores de aplicações e gerenciadores de banco de dados distribuídos.

Portanto, estes ambientes compartilhados devem ser evitados porque a falta de controle do ambiente por parte dos testes acarreta em diversos antipadrões nas baterias de testes, como desempenho ruim, código pouco legível e difícil de manter.

O que favorece a escrita de testes de qualidade é ter um ambiente de teste controlado, sobre o qual eles podem fazer diversas suposições de forma a diminuir o código-fonte e melhorar o seu desempenho. Estes ambientes podem até ser compartilhados, mas deve existir uma organização entre os projetos que o utilizam a fim de criar um certo nível de isolamento para os testes. No entanto, muitas vezes isso nem sempre é possível porque os testes e projetos podem ser conflitantes dentro de um mesmo ambiente, a tal ponto que se torna impossível organizar de maneira efetiva todas as equipes. Por exemplo, se um projeto executar um teste de longevidade no ambiente, a infraestrutura pode ficar inadequada por um longo período de tempo para os outros projetos. Outro problema é a falta de autonomia dos testes para configurar o ambiente apropriadamente, o que pode impossibilitar a criação de casos de testes importantes.

Assim, mesmo que apenas um projeto utilize o ambiente, ele também será compartilhado entre os desenvolvedores de um mesmo time, portanto, a execução dos testes de um desenvolvedor pode influenciar a execução de outro. Logo, o mais adequado para a realização rotineira dos testes de correção é aquele completamente isolado e independente, pois facilita a escrita dos testes e dispensa a negociação do ambiente com outros desenvolvedores e times que podem ter prazos e objetivos conflitantes.

Um ambiente isolado pode ser o próprio computador de cada desenvolvedor ou testador, isto é, a máquina local, pois geralmente existe mais autonomia para instalar e configurar programas da forma que se desejar. Também pode haver máquinas dedicadas para isso sem a exigência de imitar o ambiente de produção, onde podem ser instalados programas de integração contínua [52] para executar as baterias de testes desejadas, inclusive com agendamentos de horários. Para verificar os casos de testes que requerem o ambiente parecido com o de produção, pode ser criado uma bateria de testes exclusiva e isolada que serão executados em ambientes compartilhados.

Para quando houver problemas com desempenho devido à grande quantidade de testes, pode ser necessário paralelizar a execução dos testes, por exemplo através de grades computacionais. A ferramenta Selenium Grid, por exemplo, facilita a criação de uma grade computacional para execução dos testes Web criados com a ferramenta Selenium.

Quando houver necessidade de testar um programa em diversos sistemas operacionais, navegadores Web ou ainda com diferentes configurações de hardware, será necessário criar ambientes com cada uma das exigências. Uma alternativa com baixo custo é a criação e configuração de máquinas virtuais com o auxílio de emuladores próprios como o VMware, o VirtualPC ou o VirtualBox que permitem a emulação de máquinas com características diferentes de hardware e com sistemas operacionais desejados, além da possibilidade de instalação de qualquer sistema de software compatível.

4.5 Quando Executar

Uma grande vantagem da automação de testes em relação a fazer testes manuais é que eles podem ser reproduzidos identicamente a qualquer momento e sem custos adicionais relevantes. Essa possibilidade proporciona rápido *feedback* a todo momento que novos resultados são obtidos. Por isso, o recomendado é executar os testes o mais rápido possível após qualquer alteração que possa influenciar a execução do programa, como as de código e de configuração, atualizações de dependências ou mesmo mudanças na infraestrutura.

No entanto, alguns tipos de testes não podem ser executados a todo momento porque são intrinsecamente lentos, como os testes de desempenho e longevidade. Testes de segurança e usabilidade também podem ser indesejavelmente lentos, dependendo das verificações e das ferramentas utilizadas. Outros tipos de teste não precisam ser executados a todo momento, como os de carga e estresse, pois avaliam a infraestrutura que geralmente não é alterada com frequência. Dessa forma, esses tipos de testes podem

ser agendados para serem executados em dias e horários específicos, definidos de acordo com prazos de entregas e outras necessidades.

Os tipos de testes com maior prioridade são os de aceitação e de unidade, que são focados nos aspectos básicos de qualidade: aceitação e correção. O *feedback* destes tipos de testes são os mais valiosos em tempo de implementação porque eles permitem revelar com rapidez erros nas novas funcionalidades e erros de regressão. A frequência de alterações no código-fonte pode ser alta, podendo chegar a dezenas de linhas modificadas por minuto, todas sujeitas a falhas.

Idealmente, os testes de unidade devem ser rápidos porque a sua quantidade pode ser muito grande. O código-fonte dos testes de unidade pode até ser mais extenso que o próprio sistema em teste, isto porque para cada método da aplicação podem haver dezenas de métodos de teste. Com cada caso de teste demorando milésimos de segundo para ser executado, torna-se viável a execução de todos os testes a cada alteração do código-fonte.

O malefício de não executar os testes a cada alteração do código é que os erros se acumulam, o que torna mais lenta a compreensão e identificação dos erros, podendo até atrasar o projeto como um todo. Para evitar que isso aconteça e para poupar o trabalho repetido de executar a bateria de testes, foram criadas ferramentas de Testes Contínuos (*Continuous Testing*) [124], que detectam quando o código-fonte foi alterado e executam a bateria de testes automaticamente.

Os testes de aceitação são geralmente demorados, porque eles devem integrar todas as camadas do sistema, incluindo algoritmos e processos lentos e camadas de persistência, como banco de dados e sistemas de arquivos. Por isso, pode ser inviável executá-los com a frequência dos testes de unidade, mas devem ser realizados pelo menos uma vez ao dia, por exemplo, em um ambiente de integração contínua [52].

Os ambientes de integração contínua são fundamentais para os projetos desenvolvidos com métodos ágeis porque possibilitam a criação de *releases* diários, também conhecidos como *snapshots* ou *nightly builds*. Com estes programas é possível automatizar o processo inteiro de construção do software, incluindo a compilação, ligação de dependências e execução de diversas baterias de testes. A construção pode ser programada para ser executada em horários específicos, preferencialmente quando os ambientes não estão sendo utilizados, dessa forma, há a segurança de que as baterias de testes serão executadas frequentemente. Existem ferramentas que facilitam a criação desses ambientes e dessas configurações, tais como CruiseControl e Continuum.

O que é importante notar é que executar os testes com frequência é fundamental. É mais vantajoso executar frequentemente uma bateria pequena de testes do que executar raramente uma bateria grande de testes. Baterias de testes que não são executadas tendem a ficar desatualizadas, o que pode reduzir drasticamente o custo-benefício da automação de testes.

O custo da criação dos testes automatizados geralmente é maior do que o dos testes manuais, justamente por causa do tempo utilizado com a implementação. Por isso, é necessário que este custo adicional seja compensado com os benefícios da automação, como a possibilidade de execução de baixo custo da bateria completa de testes, que auxilia o desenvolvimento e evita o desperdício de tempo gasto com a depuração dos sistemas.

Além disso, as baterias de testes que não são executadas regularmente tendem a se tornar obsoletas, caso o sistema em teste continue em desenvolvimento. A manutenção do sistema pode alterar o *design* do código-fonte e até mesmo o comportamento do sistema. Dessa forma, os testes podem não verificar mais o comportamento esperado do sistema e até mesmo não compilar. Também é importante notar que quanto maior for o acúmulo de casos de testes obsoletos, mais desgastante será o trabalho de identificação dos erros, já que cada um deles pode influenciar de diferentes maneiras os casos de teste.

4.6 Documentação

Metodologias tradicionais de desenvolvimento dão grande ênfase a documentos, tanto referentes ao software quanto ao processo de desenvolvimento. Por outro lado, os métodos ágeis recomendam priorizar a colaboração com o cliente em vez de organizar o processo de desenvolvimento através de documentos burocráticos. A comunicação rápida e efetiva dispensa documentos longos e detalhados, que requerem muito esforço para se manterem atualizados e não agregam mais valor que um software de qualidade. Com os métodos ágeis, os requisitos são organizados em pequenas histórias, contendo informações enxutas que direcionam o desenvolvimento.

Os testes de aceitação visam averiguar se as histórias estão implementadas como o esperado. Através de uma linguagem próxima da natural, o cliente descreve exemplos de uso das funcionalidades do sistema e os comportamentos esperados. Exemplos compõem uma documentação efetiva que podem acelerar o aprendizado do sistema, pois ele apresenta de forma prática e objetiva como o sistema deve ser usado.

Por isso, os testes de aceitação podem e devem ser utilizados como um artefato de *documentação executável* do sistema, tanto para os próprios clientes e usuários finais quanto para os desenvolvedores. A maior vantagem dessa documentação é que ela se autoverifica dinamicamente, se está atualizada, a cada vez que a bateria de testes é executada, ou seja, se um teste falhar é um indício de que ou aquele trecho da documentação está obsoleto ou então o sistema não satisfaz um certo requisito.

Outro tipo de documentação de sistemas de software são os artefatos destinados ao usuário final, tais como *screencasts*, tutoriais, FAQs[1], manuais de usuário e documentação de APIs[2], no caso de bibliotecas de software. Esses artefatos são indispensáveis em qualquer metodologia de desenvolvimento, pois estão diretamente relacionados com a usabilidade do sistema e com aspectos de mercado e divulgação. No entanto, trabalhando com métodos ágeis, esses artefatos podem ser gerenciados como as funcionalidades, isto é, cada tarefa relacionada com esse tipo de documentação pode entrar em uma pilha de tarefas para serem priorizadas. Não obstante, as tarefas de documentação podem até serem priorizadas na mesma pilha de tarefas de funcionalidades, já que este tipo de documentação também agrega valor diretamente ao produto final.

Também pode ser interessante utilizar os testes de interface como parte da documentação do usuário final, já que muitas ferramentas permitem visualizar os eventos simulados de dispositivos como *mouse* e teclado, ou seja, os testes podem servir de demonstração de uso do sistema. É semelhante a um *screencast*, com as vantagens de ser facilmente editado e de consumir consideravelmente menos memória de armazenamento. A desvantagem é que não é viável fazer edições mais elaboradas, como adição de áudio, narração e efeitos.

Mais um tipo de documentação típica em sistemas de software é a do código-fonte, direcionada aos desenvolvedores e outros membros da equipe de desenvolvimento. Esta documentação pode conter diagramas, textos explicativos e também comentários diretamente no código-fonte, sendo que alguns deles podem até ser interpretados por ferramentas que geram relatórios organizados para documentação, tais como o *Javadoc*[3].

O que pode complementar a documentação são os relatórios de alguns tipos de testes, como os de unidade e integração, pois eles fornecem diversos exemplos de casos de sucesso e de fracasso que determinados trechos do código estariam sujeitos. Em algumas situações, pode até ser mais fácil entender um algoritmo a partir de exemplos do que a partir de uma descrição detalhada. Contudo, para que os relatórios e até o código dos testes possam ser utilizados como documentação, é necessário que o código esteja muito bem escrito, modularizado e com ótima legibilidade, por isso é importante dar nomes coerentes e autoexplicativos para classes, métodos e variáveis.

[1] FAQ: *Frequently Asked Questions* contém uma lista de respostas das perguntas mais frequentes.
[2] API: *Application Programming Interface* ou Interface de Programação de Aplicações.
[3] Ferramenta para gerar documentação de APIs em HTML.

Os relatórios dos testes automatizados, que são gerados automaticamente, também têm outra utilidade: eles podem ser usados como uma certificação de qualidade, já que é um documento que apresenta todos os casos de testes dos quais o software foi submetido. Com os testes manuais é necessário criar ou alimentar com dados estes documentos manualmente, ou seja, mais um trabalho que pode exigir grande esforço e que é difícil de ser reaproveitado e mantido atualizado.

4.7 Considerações Finais

Testes automatizados é uma prática muito efetiva para aumentar a qualidade dos sistemas de software, mas desde que eles também sejam de qualidade. O código do teste precisa ser organizado, legível e ter um bom desempenho. Para isso, eles devem ser implementados com o mesmo cuidado e atenção do código do sistema principal.

A qualidade de um projeto não deve ser responsabilidade apenas de alguns membros da equipe. A qualidade precisa estar disseminada entre todas as pessoas envolvidas; todas devem ser responsáveis por ajudar no que estiver a seu alcance para a produção de um software de excelente nível. Dessa forma, gerentes, desenvolvedores, *designers*, testadores, analistas, clientes e usuários finais devem contribuir para a criação de baterias de testes boas e completas.

Os testes devem ter alta prioridade durante o desenvolvimento para que eles sejam implementados o mais breve possível, pois quanto o *feedback* rápido facilita e agiliza a tarefa de melhoria do sistema. Os testes de correção (unidade e aceitação) devem ser implementados concorrentemente com o código do sistema, enquanto outros tipos de testes podem ser implementados quando houver necessidade, dependendo dos problemas do software e dos objetivos do projeto.

As baterias de testes devem ser executadas idealmente em ambientes isolados e controlados para facilitar a escrita e manutenção dos testes. Quando não existe total autonomia do ambiente, os testes precisam prever diferentes situações que aumentam a complexidade do código-fonte e podem atrapalhar a execução, diminuindo o desempenho ou até mesmo quebrando os casos de teste, impossibilitando interpretar com fidelidade os resultados obtidos.

Para facilitar o desenvolvimento do sistema e dos testes, a execução das baterias de verificações deve ser a mais frequente possível, principalmente no que diz respeito aos testes de unidade e de aceitação, que são os mais especializados para garantir a correção do sistema. Se a frequência de execução das baterias de testes é pequena, os erros se acumulam, o que pode dificultar a compreensão dos relatórios e tornar o processo de manutenção muito desgastante.

Ainda, tanto o código dos testes quanto os relatórios gerados nas execuções são artefatos úteis para documentação do sistema, tanto para desenvolvedores, quanto para clientes e usuários finais. Os testes são exemplos de uso do sistema, que é um tipo prático e eficaz de documentação.

Parte II

Práticas, Padrões e Técnicas para Testes de Correção

Capítulo 5

Introdução da Parte II

Como foi discutido na Parte I, testes automatizados fazem parte de uma abordagem de controle de qualidade simples e que pode ser eficaz e de baixo custo. Por estas características, ela é indicada para complementar o controle de qualidade de qualquer projeto de software, desde simples *scripts* até sistemas onde não são toleráveis erros, como sistemas médicos, pilotos automáticos de aviões e de controladores de voo.

No entanto, assim como qualquer trecho de código de computador, os testes automatizados também estão sujeitos a erros, desde pequenas imperfeições até falhas graves. O código dos testes pode ser definido segundo alguma linguagem de programação ou mesmo descrito através de linguagens de marcação, como HTML. Geralmente ele é criado por um ser humano, mas há situações onde o código é gerado por alguma ferramenta, o que também é suscetível a erros.

Por isso, a escrita e a manutenção dos testes automatizados requer muitos cuidados, pois testes não eficazes ou que exigem alta manutenção resultam em desperdício de tempo e, consequentemente, podem acarretar grandes prejuízos. Nesta parte da dissertação, será discutido o conceito de qualidade na automação de testes, destacando boas práticas, padrões e técnicas para criação de bons testes automatizados de correção.

Primeiramente, esse capítulo discutirá o que é um bom teste de correção, mostrando as principais características que um teste precisa ter para que ele seja considerado de qualidade. Na sequência, serão apresentados indícios de problemas, descritos por Meszaros como *cheiros* (*smells*) [99]. Para completar, será feito um paralelo entre os indícios de problemas e as características de qualidade. Ainda nesse capítulo, serão definidos os esqueletos dos padrões e antipadrões que aparecerão no decorrer desta parte da dissertação.

Nos três capítulos seguintes, Capítulos 6, 7 e 8, serão apresentados padrões e boas práticas de implementação de **Testes de Unidade**, **Testes com Persistência de Dados** e **Testes de Interface de Usuário**, respectivamente. Cada um desses capítulos possui uma introdução que discute as principais práticas que devem ser consideradas para a automação. Posteriormente, serão apresentados padrões específicos para tornarem os testes mais organizados e robustos, além de favorecer a escrita de sistemas com maior testabilidade e qualidade. Por último, serão descrito antipadrões, com o objetivo de serem evitados.

Para finalizar essa parte, o Capítulo 9 irá discutir as **Técnicas de Desenvolvimento de Software com Testes Automatizados**, tais como TFD, TDD, e BDD. Cada técnica pode influenciar diretamente na qualidade dos testes e do sistema.

5.1 Testes de Correção de Qualidade

No decorrer desta parte da dissertação serão apresentadas boas práticas para criação de bons testes automatizados de correção. Contudo, para um bom entendimento dos capítulos a seguir, é necessário conhecer as principais características que definem a qualidade desse tipo de testes.

Sendo assim, a seguir serão descritas as principais propriedades de um bom conjunto de testes automatizados de correção. Elas foram reunidas a partir de trabalhos de diferentes autores [71, 99, 97] referente principalmente a testes de unidade. No entanto, a argumentação a seguir foi generalizada para qualquer tipo de testes de correção, incluindo testes com persistência de dados e com interface de usuário.

Automático: Deve ser possível executar as baterias de testes sem qualquer intervenção humana. Essa propriedade é fundamental para viabilizar o uso de ferramentas de integração contínua e de testes contínuos.

Repetitível: Métodos de *set up* e *tear down*, ou outras funcionalidades dos arcabouços, devem garantir que os testes possam ser executados quantas vezes forem necessárias, de tal forma que sempre produzam os mesmos resultados. Em outras palavras, o número de vezes que um teste é executado não deve interferir nos resultados dos testes. Essa característica pode ser difícil de ser assegurada quando os algoritmos trabalham com datas e horários ou com persistência de informações, tais como bancos de dados e objetos *Singleton*.

Útil: O propósito dos testes deve ser verificar algo de importante no sistema. Testes realizados irresponsavelmente apenas para melhoria de métricas do projeto devem ser rejeitados. Ainda, o custo-benefício de um caso de teste deve ser baixo, o que não acontece com testes replicados, semi-automatizados, frágeis ou que exigem muita manutenção.

Único: Cada caso de teste deve verificar um comportamento específico do sistema ou utilizar uma partição de domínio de dados de entrada. Testes que não agregam valor ao processo de controle de qualidade prejudicam as baterias de testes como um todo, pois deixam-nas mais lentas e mais extensas, o que piora a legibilidade e a manutenibilidade.

Preciso: O resultado do teste não pode ser ambíguo, ou seja, não deve haver falso positivo ou falso negativo (Capítulo 3.2). Para assegurar esse aspecto, devem ser evitados testes com lógica condicional ou com muitos pontos de verificação.

Profissional: A qualidade e a organização do código-fonte dos testes deve ser a mesma do código-fonte de produção para prevenir erros e facilitar a manutenção. Refatoração e Padrões de Projeto também podem ser aplicado ao código dos testes.

Legível e Claro: O código dos testes deve ser o mais próximo da linguagem natural quanto possível, dessa forma, eles poderão ser utilizados como uma boa documentação do sistema. Para isso, é essencial utilizar nomes auto-explicativos de variáveis, métodos e classes.

Simples: A criação e manutenção de um caso de teste deve ser uma tarefa sem grandes dificuldades. Para isso, é fundamental o uso de arcabouços e ferramentas de testes adequadas para que o teste se torne o mais simples possível. Não obstante, essa característica também depende da testabilidade (Seção 10.3) do sistema em teste.

Independente: Os casos de testes devem ser independentes uns dos outros. A ordem de execução dos casos de testes não deve interferir no resultado dos testes. Ainda, deve ser possível executar dois ou mais testes em paralelo sem qualquer intervenção. A execução em paralelo se torna mais complicada quando são utilizados recursos compartilhados, tais como gerenciadores de banco de dados.

Isolado: Cada caso de teste deve verificar apenas um aspecto do sistema, para isso, quanto mais isolado ele estiver de outros módulos do sistema e do ambiente, mais fácil será sua escrita e manutenção. O ambiente em questão pode ser a infraestrutura de rede, dispositivos de *hardware*, Internet,

serviços externos etc. Quando um caso de teste depende do ambiente, a execução do teste fica mais suscetível a erros, pois ele depende de configurações externas que estão fora de seu controle. A configuração do ambiente se torna especialmente complexa quando os serviços são de responsabilidade de outras equipes de desenvolvimento ou até mesmo de outras empresas. A solução é simular os serviços externos com auxílio de Objetos Dublês (Seção 6.2), dessa forma, se um teste falhar, o motivo do erro tende a ser facilmente identificado.

Rápido: O desempenho dos testes é importante, contudo, não é necessário otimizações exageradas no código dos testes. O recomendável é separar os casos de testes que são intrinsicamente lentos em baterias de testes específicas, dessa forma fica mais fácil organizar quando as baterias de testes deverão ser executadas. Comumente, testes de unidade devem ser executados em poucos milissegundos, enquanto testes que envolvem persistência ou interfaces de usuários são normalmente mais lentas, podendo durar até alguns poucos segundos.

5.2 Indícios de Problemas

Testes automatizados que não possuem as características descritas na seção anterior podem trazer grandes prejuízos para um projeto, como perda de tempo e dinheiro. Em casos extremos, o tempo gasto para a automatização dos testes pode não ser recompensada pela melhoria da qualidade do sistema em teste. Por isso, é importante identificar o mais breve possível os problemas das baterias de testes, caso existam.

Os tipos de problemas das baterias dos testes são comuns a diferentes projetos, assim como os problemas rotineiros dos sistemas de software. Da mesma forma que Martin Fowler nomeou e descreveu alguns dos principais indícios de problemas de software (*smells*), que podem ser solucionados através de refatorações [59], Meszaros [99] fez o mesmo relacionado com os problemas da automação dos testes.

Abaixo seguem os indícios de problemas descritos por Meszaros, que os organizou em três categorias: indícios de problemas de código-fonte, de comportamento e de projeto. Os indícios apenas serão citados com uma breve descrição, já que o próprio nome dos indícios são intuitivos.

Indícios de Problemas de Código-Fonte (*Code Smells*)

Teste Obscuro (*Obscure Test*) Testes muito complexos, pouco legíveis ou que verificam muitos aspectos do sistema de uma só vez. Testes obscuros dificultam o entendimento e a manutenção.

Testes com Lógica Condicional (*Conditional Test Logic*) Testes que possuem comandos condicionais para conseguirem lidar com diversas situações do ambiente. Normalmente, isso deve-se à incorreta utilização de métodos de *set up* e *tear down*, ou a classe de testes está muito extensa.

Código Difícil de Testar (*Hard-to-Test Code*) Sistema com baixa testabilidade tornam os testes difíceis de serem automatizados (Seção 10.3), consequentemente, os testes tendem a ficar obscuros.

Replicação do Códito dos Testes (*Test Code Duplication*) Replicação do código dos testes pode significar que o código de produção também está replicado ou que é preciso refatorar o código dos testes.

Lógica dos Testes em Produção (*Test Logic in Production*) Ocorre quando parte do código dos testes é incorporada ao código do sistema em teste. O objetivo dos arcabouços de testes é justamente facilitar a escrita dos testes de forma completamente isolada.

Indícios de Problemas de Comportamento (*Behavior Smells*)

Erros não Claros (*Assertion Roulette*) Ocorre quando é necessário muita depuração para identificar o motivo da falha de um teste. Isso é rotineiro quando um teste faz muitas verificações ou quando não são precisos, ou seja, possuem resultado falso positivo ou falso negativo.

Testes Intermitentes (*Erratic Test*) Quando um teste não é completamente independente dos outros ou de fatores externos, eles podem produzir resultados inesperados, dessa forma não são testes reprodutíveis.

Testes Frágeis (*Fragile Test*) Testes que param de funcionar por causa de mudanças insignificantes no sistema em teste, tais como adição de novos casos de testes ou pequenas mudanças no leiaute da interface de usuário.

Depuração Frequente (*Frequent Debugging*) Se um desenvolvedor está perdendo muito tempo com depuração, é um sinal de que testes automatizados estão faltando ou então que os testes não estão claros.

Intervenção Humana (*Manual Intervention*) Testes que são semi-automatizados, isto é, depende de um ser humano para enviar informações ao sistema para que então o teste possa proceder.

Testes Lentos (*Slow Tests*) Testes que são intrinsicamente lentos ou que não são bem isolados.

Indícios de Problemas de Projeto (*Project Smells*)

Testes com Erro (*Buggy Tests*) Assim como o código do sistema em teste, o código dos testes também podem ter erros. Normalmente isso ocorre quando a criação dos testes não é uma tarefa trivial.

Desenvolvedores não Escrevem Testes (*Developers Not Writing Tests*) Quando o prazo de entrega está curto, o mais natural é sacrificar o tempo gasto com a escrita dos testes automatizados, consequemente, o tempo gasto com a qualidade do sistema.

Alto-custo de Manutenção (*High Test Maintenance Cost*) Baterias de testes de baixa qualidade exigem um alto-custo de manutenção.

Erros em Produção (*Production Bugs*) Erros em produção é um indício de que alguns comportamentos do sistema não foram verificados, portanto, há falta de testes automatizados.

Indícios de Problemas *vs.* Características de Qualidade dos Testes

Tendo o conhecimento das principais características de qualidade dos testes e dos indícios de problemas mais rotineiros no desenvolvimento com testes automatizados, é possível fazer um paralelo entre eles para ajudar na manutenção e gerenciamento das baterias de testes.

A Tabela 5.1, criada pelo autor, apresenta quais aspectos de qualidade devem ser primeiramente questionados quando são encontrados alguns dos indícios de problemas citados. Ainda, a tabela pode ser usada de maneira inversa, dado que queremos assegurar determinada característica de qualidade dos testes, podemos verificar prioritariamente os indícios de problemas mais prováveis.

5.3 Definição de Padrão

Padrões descrevem soluções comuns para problemas recorrentes no desenvolvimento de sistemas de software. Esse termo foi aplicado em Ciência da Computação pela primeira vez com os Padrões de Projeto de sistemas orientados a objetos [61], mas, hoje, ele também é utilizado em inúmeros outros

Indícios de Problema \ Aspectos de Qualidade	Automático	Repetitível	Útil	Único	Preciso	Profissional	Claro	Simples	Independente	Isolado	Rápido
Obscure Test						x	x	x		x	
Conditional Test Logic		x		x	x	x	x	x			
Hard-to-Test Code	x						x	x	x	x	
Test Code Duplication			x	x		x	x				x
Test Logic in Production		x				x	x		x		
Assertion Roulette		x			x				x	x	
Erratic Test		x	x		x				x	x	
Fragile Test		x	x			x			x	x	
Frequent Debugging		x	x		x	x	x	x	x	x	
Manual Intervention	x	x	x				x			x	x
Slow Tests									x	x	x
Buggy Tests					x	x		x	x	x	
Developers Not Writing Tests	x					x	x				
High Test Maintenance Cost	x	x	x	x	x	x	x	x	x	x	x
Production Bugs	x					x					

Tabela 5.1: Indícios de Problemas *vs.* Aspectos de Qualidade.

contextos, incluindo os testes automatizados [99]. Dessa forma, esses padrões ajudam a criar testes de qualidade típicos para cenários rotineiros em sistemas de software.

A organização de um texto em padrões o torna facilmente extensível, ou seja, novos padrões podem ser adicionados sem a necessidade da reformulação de muitos outros trechos do trabalho. A descrição de um padrão segue uma estrutura enxuta e padronizada de tópicos pré-definidos, o que tende a facilitar o estudo, principalmente para os leitores que já conhecem parte da teoria, pois os tópicos podem ser lidos em uma ordem arbitrária.

Alguns dos padrões que serão descritos nos próximos capítulos já foram identificados por outros autores e nomeados em inglês, mas devido à sua grande importância para a criação de testes automatizados de qualidade, eles também foram incluídos nessa dissertação. Os nomes desses padrões foram traduzidos para o português, mas seus nomes originais em inglês foram mantidos para fácil identificação. Além disso, para esses casos, haverá um tópico com as principais referências da solução. Quanto aos outros padrões, eles foram identificados e nomeados pelo autor desta dissertação. No entanto, independente da autoria, todos serão descritos de acordo com a seguinte estrutura:

Nome: Nome do padrão. Caso o nome possua a tradução para o inglês, significa que ele já foi identificado por outros autores.

Tipo: Neste trabalho, um padrão poderá ser dos tipos Testabilidade, Organizacional, Robustez, Qualidade ou Desempenho. Testabilidade indica que o padrão ajuda a melhorar essa característica do sistema em teste; Organizacional aponta que é um padrão utilizado para ajudar a organizar os casos de teste; já os padrões de Robustez servem para tornar os testes menos frágeis; os de Qualidade são úteis para fazer análises interessantes do sistema em teste em busca de erros; por último, Desempenho para aumentar a velocidade de execução das baterias de testes.

Quando utilizar: Contém sugestões de situações típicas que o padrão deve ser utilizado.

Intenção: Contém uma breve descrição dos objetivos do padrão.

Motivação: Apresenta os problemas que o padrão ajuda a resolver.

Solução: Contém a descrição da solução proposta pelo padrão.

Consequências: Discute como o padrão ajuda a solucionar os problemas descritos no tópico anterior.

Implementação: Contém informações de como o padrão pode ser implementado. Se a implementação for específica para um nicho de sistemas, o nome do tópico irá indicá-lo, por exemplo, Implementação - WUI. Um mesmo padrão poderá ter tópicos de implementação para mais de um nicho de sistemas.

Exemplo: Contém exemplos de código-fonte demonstrando o padrão. Os exemplos neste trabalho utilizarão as linguagens C, Java, Scala ou Python. O nome do tópico irá informar a linguagem e as tecnologias utilizadas, por exemplo, Exemplo - Python/UnitTest/QAssertions. Um mesmo padrão poderá ter exemplos em mais de uma linguagem e as referências das tecnologias estão nas páginas iniciais da dissertação.

Padrões Relacionados: Outros padrões que possuem algum relacionamento importante com o padrão correspondente.

Usos Conhecidos: Sistemas ou ferramentas que utilizaram a solução proposta e que serviram de amostra para nomeação do padrão. As referências estão nas páginas iniciais da dissertação.

Referências: Tópico não obrigatório. Contém outras referências para o padrão, caso ele já tenha sido nomeado por outro autor.

5.4 Definição de Antipadrão

Antipadrões também descrevem soluções para problemas recorrentes, mas são propostas equivocadas, pois causam problemas para o desenvolvimento dos sistemas e dos testes. Contudo, a descrição dos antipadrões será mais simples do que dos padrões.

Os antipadrões, com raras exceções, nunca devem ser utilizados. Em relação à Implementação, os programadores não precisam conhecer todos os detalhes das soluções. Quanto aos Usos Conhecidos, não seria elegante expor negativamente projetos sem uma boa causa. Já a Intenção e a Motivação foram agrupados em um único item, Contexto, para simplificar e substituir a conotação positiva dos termos. Os tópicos Consequências e Padrões Relacionados foram substituídos pelo tópico Indícios de Problemas Relacionados. Sendo assim, a organização de um antipadrão segue a estrutura abaixo:

Nome: Nome do antipadrão. Se o nome tiver a tradução para o inglês, significa que ele já foi identificado por outros autores como antipadrão ou até mesmo como padrão. Em alguns dos casos, o autor desse trabalho interpretou padrões nomeados por outros trabalhos como soluções ruins e, portanto, devem ser entendidas como antipadrões.

Tipo: Os antipadrões poderão ser dos tipos Organizacional, Robustez ou Testabilidade. A explicação de cada um dos tipos é análoga à descrita na seção anterior, com a diferença de que as soluções indicadas são prejudiciais.

Contexto: Apresenta, de forma resumida, as situações típicas onde aparecem esses antipadrões.

Exemplo: Similar ao tópico de um padrão.

Indícios de Problemas Relacionados: Mostra as ligações dos indícios de problemas (Seção 5.2) que podem ser causados pelo antipadrão, ou seja, as possíveis consequências causadas pela solução equivocada.

Referências: Similar ao tópico de um padrão.

Capítulo 6

Testes de Unidade

Como foi descrito na Seção 3.3.1, um teste de unidade verifica a correção dos menores trechos de código que possuem um comportamento definido e nomeado. Normalmente associamos uma unidade a um método ou função do sistema, mas em certas situações podemos entender unidade como blocos, classes que geralmente são muito simples, módulos ou até mesmo constantes (variáveis com valores fixos) [99].

Os testes de unidade são, via de regra, os mais importantes dentre os demais tipos de teste, pois são os mais apropriados para verificação da correção, que é a premissa básica de qualquer sistema de software de qualidade, além de ser pré-requisito dos outros aspectos de qualidade. Apesar de existirem erros de correção insignificantes, que não impedem o bom funcionamento do sistema ou que trazem pouco prejuízo se comparado com outras irregularidades, falhas na correção do sistema são, geralmente, críticas e mais intoleráveis do que problemas de desempenho, segurança ou usabilidade [70].

Independentemente da gravidade dos erros de correção, sempre é desejável que o sistema esteja completamente correto, isto porque falhas na correção do sistema podem ocultar outros defeitos do software e até mesmo desvalorizar os resultados de outras baterias de testes. Por exemplo, um algoritmo que não trata todas as situações esperadas pode ser muito mais rápido do que a versão correta do algoritmo, sendo assim, os testes de desempenho realizados sobre o sistema trazem resultados que podem ser descartados. Por isso, é recomendado que outras baterias de testes só sejam executadas depois que as baterias de testes de correção estejam completas e sem falhas.

De qualquer maneira, todos os tipos de testes e ferramentas podem ajudar a melhorar a correção dos sistemas, principalmente em relação a situações raras e pequenos detalhes que passam despercebidos durante a implementação. Além disso, outros tipos de testes submetem o sistema a situações peculiares que podem exercitar módulos e dependências que não foram previamente testados. Contudo, é importante não atribuir a responsabilidade de encontrar erros de correção a baterias de outros tipos de testes, principalmente por causa do uso de ferramentas que não são apropriadas e, portanto, induzem à ocorrência de diversos antipadrões que tornam os testes difíceis de escrever e manter. As ferramentas para testes de unidade são geralmente bibliotecas de código-fonte que disponibilizam funcionalidades para facilitar o manuseio e verificação de trechos do sistema, e essas ferramentas já são suficientes para a automação produtiva dos testes de unidade durante o desenvolvimento.

Ao contrário de outros tipos de testes que precisam que determinados módulos do sistema estejam finalizados para serem executados com êxito, os testes de unidade devem ser escritos e executados sobre pequenas porções de código que não constituem isoladamente uma funcionalidade do sistema do ponto de vista dos usuários finais. Essa característica é fundamental para encontrar erros de correção nos estágios iniciais de desenvolvimento e para evitar desperdício de tempo com depuração, pois os testes de unidade podem ser bem especializados e a falhas nos testes tendem a ser facilmente localizadas [68].

6.1 Arcabouços para Testes de Unidade

Executar e testar um sistema são tarefas intrínsecas ao desenvolvimento de sistemas de software. Durante a implementação é comum executar o sistema periodicamente, seja para averiguar seu progresso, encontrar erros ou até mesmo para incentivo pessoal pela visualização do trabalho que está sendo realizado. No entanto, só é possível executar certas funcionalidades do sistema quando um conjunto determinado de módulos está pelo menos parcialmente implementado. Por exemplo, pode ser necessário que a interface de usuário, assim como o ponto de partida da execução do sistema (método *main*), que é obrigatório para muitas linguagens de programação, estejam implementados para que o usuário possa fazer a chamada de uma determinada funcionalidade.

Uma solução trivial para executar pequenos trechos do sistema sem depender de outros módulos é a criação de métodos *main* (geralmente implementados no próprio arquivo de código da funcionalidade) exclusivos para a execução do trecho de código desejado juntamente com a utilização de comandos *print*, que são utilizados para imprimir valores de variáveis. Esta solução nada mais é que um teste de unidade manual, ou seja, é executado um pequeno trecho do sistema e as verificações são feitas por um ser humano que compara o valor impresso com o valor esperado.

Além de todos os problemas da abordagem manual que já foram discutidos na Parte I, essa solução mistura o código dos testes com o do sistema, prejudicando a legibilidade, depuração e manutenção do código, que pode implicar piora da correção do sistema a médio e longo prazo. Desses padrões e antipadrões de testes manuais de unidade surgiram os arcabouços que deram origem à automação de testes [20].

Por volta de 1994, Kent Beck criou o arcabouço `SUnit` [13] para fazer testes automatizados em seus projetos que utilizavam a linguagem SmallTalk. Até hoje o `SUnit` é utilizado, além de servir como referência para implementação de arcabouços semelhantes para outras linguagens de programação. Em 1998, por exemplo, Kent Beck e Erich Gamma desenvolveram o arcabouço `JUnit` para a linguagem Java com base no `SUnit`. Kent Beck e colaboradores também escreveram na página Web oficial do JUnit um artigo descrevendo passo a passo a sua implementação para referência[1].

O conjunto de arcabouços que são inspirados na arquitetura do SUnit são conhecidos como família *xUnit*. Entre eles estão o *PyUnit* para Python, CppUnit para C++, JsUnit para JavaScript etc. Todos eles facilitam a escrita do código dos testes evitando replicação de trabalho e sem prejudicar o código do sistema principal.

Cada arcabouço da família *xUnit* possui suas particularidades, devido, principalmente, às diferenças das linguagens, mas todos seguem uma mesma estrutura básica e extensível que facilita a escrita e acompanhamento dos testes automatizados. Esses arcabouços possuem basicamente três responsabilidades: 1) possibilitar a criação e organização de casos de testes com pouco esforço e sem replicação de código; 2) facilitar a comparação dos valores obtidos com os valores esperados e 3) gerar um relatório claro dos resultados obtidos.

Um caso de teste é representado por um método quando são utilizadas linguagens orientadas a objetos, ou por uma função no caso de linguagens estruturadas. Contudo, nem todo método ou função do código-fonte é um caso de teste, por isso as ferramentas utilizam metadados[2] (Figura 6.2) ou convenções[3] (Figura 6.1) para identificar quais deles devem ser interpretados como casos de teste. Quando a ferramenta xUnit é executada, ela processa apenas os métodos/funções que são definidos como casos de testes, o que dispensa escrever métodos *main* e também permite rodar todos casos de teste com apenas um comando.

Para comparar os valores obtidos com os valores esperados automaticamente, basta substituir os

[1] `http://junit.sourceforge.net/doc/cookstour/cookstour.htm`

[2] A versão 4 ou superior do JUnit associa métodos a casos de teste através da anotação Java `@Test`.

[3] Muitas ferramentas seguem a convenção de que se o nome do método começa com a palavra `test`, então ele é um caso de teste.

```java
// Referências do JUnit
import junit.framework.TestCase;

// Após a execução desta classe de teste, será impresso no console:
// Será invocado pelo JUnit
public class JUnit35Exemplo extends TestCase {

  // Convenção: método começa com a palavra "test"
  public void testUmMetodoDeTeste() {
    System.out.println("Será invocado pelo JUnit");
  }

  public void umMetodoAuxiliar() {
    System.out.println("Não será invocado pelo JUnit");
  }
}
```

Figura 6.1: Definindo métodos de teste com JUnit 3.5.

```java
// Referências do JUnit
import org.junit.Test;

// Após a execução desta classe de teste, será impresso no console:
// Será invocado pelo JUnit
public class JUnit4Exemplo {

  @Test // Metadado informando que este é um método de teste
  public void umMetodoDeTeste() {
    System.out.println("Será invocado pelo JUnit");
  }

  public void umMetodoAuxiliar() {
    System.out.println("Não será invocado pelo JUnit");
  }

}
```

Figura 6.2: Definindo métodos de teste com JUnit 4 ou superior.

comandos `print` da abordagem manual por comparações (comandos `if`). Contudo, os arcabouços disponibilizam funcionalidades que fazem as verificações (asserções), substituindo esses comandos com as vantagens de minimizar a replicação de código de comparação e também de armazenar as informações pertinentes do caso de teste, como o resultado e a causa dos erros. A Figura 6.3 apresenta exemplos de verificações comuns utilizando os comandos básicos de verificação do JUnit e, também, a biblioteca Hamcrest, que torna o código dos testes mais próximo da linguagem natural se ignorarmos os parenteses, vírgulas e pontos da linguagem Java. Essa biblioteca tem como base o método de verificação `assertThat` e *Matchers* que são objetos que fazem as comparações de forma conveniente.

Na Figura 6.4 há um exemplo de um teste escrito em Java com JUnit e Hamcrest. Já na Figura 3.1 da Parte I podemos ver um exemplo de testes em Python de uma função que verifica se um número é primo.

Todas as informações armazenadas são expostas no relatório final da bateria de testes, que são fundamentais para identificar os erros e diminuir o tempo perdido com depuração. Os relatórios também são úteis para o acompanhamento da automação dos testes e para documentação do sistema. Existem diversas ferramentas que armazenam o histórico de resultados dos testes para gerar gráficos e representações que ajudam a gerenciar os projetos.

6.1.1 *Set up* e *Tear down*

Muitos cenários de testes (de qualquer tipo) só podem ser realizados segundo configurações específicas do ambiente, de dados e de estados de objetos. Por isso, é comum a prática de preparar um ambiente propício para um único ou para um conjunto de casos de testes, sejam eles automatizados ou manuais.

Tendo o ambiente configurado, um caso de teste o manipula indiscriminadamente de acordo com seus objetivos e faz as verificações necessárias. Após o término do teste, o ambiente alterado não é mais necessário, portanto, é uma boa prática da automação de testes descartá-lo, seja simplesmente para liberar memória ou até mesmo para facilitar a criação do ambiente dos próximos cenários de testes.

Por isso, outra característica comum aos arcabouços da família *xUnit* é a chamada implícita a métodos próprios para prepararem e destruirem os ambientes dos testes, com os objetivos de padronizar, simplificar e reduzir o código-fonte dos testes. Como o próprio arcabouço faz a chamada desses métodos, o desenvolvedor não precisa fazer as chamadas, apenas saber em que momento eles serão chamados.

Da mesma forma que os métodos de testes, os métodos de preparação e destruição do ambiente são definidos através de convenções ou metadados. Geralmente, o método de criação do ambiente é identificado pelo nome `set up`, enquanto os de limpeza do ambiente de *tear down*. Quando são utilizados metadados, é possível dar nomes aos métodos mais coerentes com o que está sendo realizado.

Não obstante, muitos arcabouços fornecem chamadas implícitas para esses métodos para diversos escopos: escopo de caso de teste, onde a chamada implícita é realizada antes e depois de cada caso de teste; escopo de grupos de testes, os quais as chamadas são feitas apenas antes e depois da execução de todos os casos de testes de um determinado conjunto; e, também, escopo da bateria de testes, onde apenas uma chamada é feita antes da execução da bateria inteira de todos os testes e uma após o término da execução dos mesmos.

O arcabouço TestNG, para Java, é bastante flexível em relação aos escopos de *set up* e *tear down*. A Figura 6.5 apresenta um esqueleto de teste para demonstrar o fluxo das chamadas implícitas do arcabouço.

Como os testes de unidade são isolados, é muito comum o uso dos métodos de *set up* apenas para a inicialização dos objetos necessários. Essa responsabilidade é mesma de um construtor de um objeto, entretanto, deve ser evitado sobrescrever os construtores das classes de testes para evitar interferência no fluxo de controle dos arcabouços.

Já os métodos de *tear down* são utilizados principalmente para destruir os objetos, dados e liberar a memória. No caso das linguagens que possuem coletor de lixo automático, os métodos de *tear down*

```java
// Referências Java
import java.util.*;
// Referências do JUnit
import org.junit.Test;
import static org.junit.Assert.*;
// Referências do Hamcrest
import static org.hamcrest.Matchers.*;

// Todos as verificações dos testes a seguir são válidas
public class ExemploDeVerificacoesTest {
  @Test public void comparandoInstanciaDosObjetos() {
    Object o1 = new Object();
    Object o2 = o1;
    assertSame(o1, o2);
    assertThat(o1, is(sameInstance(o2)));
    o2 = new Object();
    assertNotSame(o1, o2);
    assertThat(o1, is(not(sameInstance(o2))));
  }

  @Test public void comparandoStrings() {
    assertTrue("a".equalsIgnoreCase("A"));
    assertThat("a", is(equalToIgnoringCase("A")));

    assertTrue("..zzz!!".contains("zzz"));
    assertThat("..zzz!!", containsString("zzz"));

    assertTrue("..zzz!!".startsWith(".."));
    assertThat("..zzz!!", startsWith(".."));
  }

  @Test public void comparandoNumeros() {
    assertEquals(10, 10);
    assertThat(10, is(equalTo(10)));

    assertTrue(11 >= 11);
    assertThat(11, is(greaterThanOrEqualTo(11)));

    assertTrue(10 < 11);
    assertThat(10, is(lessThan(11)));
  }

  @Test public void comparandoPontosFlutuantes() {
    assertEquals(10.493, 10.5, 0.1); // Precisão: 1 décimo de diferença
    assertThat(10.493, is(closeTo(10.5, 0.1)));
  }

  @Test public void comparandoListas() {
    List<Integer> list = Arrays.asList(1, 2, 3, 4);
    assertTrue(list.contains(1));
    assertThat(list, hasItem(1));
  }
}
```

Figura 6.3: Exemplos de verificações com JUnit e Hamcrest.

```java
// Referências do JUnit
import org.junit.Test;
import static org.junit.Assert.*;
// Referências do Hamcrest
import static org.hamcrest.Matchers.*;

public class MathTests {
  final static double PRECISAO = 0.01;

  @Test
  public void testaValoresMuitoConhecidosDaFuncaoCosseno() {
    assertThat(Math.cos(0), is(closeTo(1.0, PRECISAO)));
    assertThat(Math.cos(90), is(closeTo(0.0, PRECISAO)));
    assertThat(Math.cos(180), is(closeTo(-1.0, PRECISAO)));
    assertThat(Math.cos(270), is(closeTo(0.0, PRECISAO)));
    assertThat(Math.cos(360), is(closeTo(1.0, PRECISAO)));
  }

  @Test
  public void testaValoresDeAngulosComunsDaFuncaoCosseno() {
    assertThat(Math.cos(30), is(closeTo(Math.sqrt(2)/2, PRECISAO)));
    assertThat(Math.cos(45), is(closeTo(0.5, PRECISAO)));
    assertThat(Math.cos(60), is(closeTo(Math.sqrt(2)/2, PRECISAO)));
  }

}
```

Figura 6.4: Exemplo de Teste em Java com JUnit e Hamcrest.

são dispensáveis nos testes de unidade, pois após a realização de todos os testes de uma classe, todas as variáveis de instância serão coletadas. Nesse caso, o uso dessa funcionalidade é uma micro-otimização desnecessária, pois o aumento da complexidade do código dos testes por causa do código adicional não é recompensada por milissegundos de velocidade.

Um caso excepcional é quando um único método de teste consome muita memória, então pode ser interessante liberá-la antes de realizar os outros casos de testes da mesma classe. Ainda, caso sejam utilizados variáveis globais, pode haver a preocupação de vazamento de memória, onde o *tear down* também será útil. A Figura 6.6 mostra um exemplo de teste de unidade em Java onde é interessante preparar e destruir o ambiente de teste.

Entretanto, os métodos de *set up* e *tear down* são mais importantes para testes de integração. Os testes com persistência de dados e de interface de usuário podem depender de ambientes complexos e propícios para tornarem os testes frágeis. Por exemplo, é comum popular um banco de dados para realização de um teste e remover os dados adicionados após a conclusão do mesmo. Assim, dados criados por um teste não afetam os outros. Já para interfaces de usuário, a preparação e destruição do ambiente refere-se comumente à abertura e fechamento dos navegadores, páginas e janelas.

6.2 Objetos Dublês (*Test Doubles*)

Algumas vezes é difícil testar um sistema porque ele pode depender de componentes que são difíceis de serem utilizados em um ambiente de teste [93, 138]. Tais componentes podem ser bancos de dados, sistemas de arquivos, redes, serviços Web, bibliotecas e até mesmo do relógio do computador, no caso de funcionalidades que envolvem datas e instantes.

Para essas situações, é mais produtivo verificar a correção do sistema através de testes de unidade

```java
// Referências do TestNG
import org.testng.annotations.*;

// Após a execução desta classe de teste, será impresso no console:
// @BeforeSuite => @BeforeTest => @BeforeClass
// @BeforeMethod => @Test: teste 1 => @AfterMethod
// @BeforeMethod => @Test: teste 2 => @AfterMethod
// @AfterClass => @AfterTest => @AfterSuite
public class TestNGExemplo {
  /* Métodos de set up */
  @BeforeSuite public void antesDeTodasAsClassesDeTestes() {
    System.out.print("@BeforeSuite => ");
  }

  // Ao contrário do BeforeClass, BeforeTest roda mesmo que não tenha teste
  @BeforeTest public void antesDessaClasseDeTestes() {
    System.out.print("@BeforeTest => ");
  }

  @BeforeClass public void antesDoPrimeiroMetodoDeTesteDessaClasse() {
    System.out.println("@BeforeClass");
  }

  @BeforeMethod public void antesDeCadaMetodoDeTesteDessaClasse() {
    System.out.print("@BeforeMethod => ");
  }

  /* Métodos de tear down */
  @AfterMethod public void depoisDeCadaMetodoDeTesteDessaClasse() {
    System.out.println("@AfterMethod");
  }

  @AfterClass public void depoisDoPrimeiroMetodoDeTesteDessaClasse() {
    System.out.print("@AfterClass => ");
  }

  @AfterTest public void depoisDessaClasseDeTestes() {
    System.out.print("@AfterTest => ");
  }

  @AfterSuite public void depoisDeTodasAsClassesDeTestes() {
    System.out.print("@AfterSuite");
  }

  /* Métodos de teste */
  @Test public void metodoDeTeste1() {
    System.out.print("@Test: teste 1 => ");
  }

  @Test public void metodoDeTeste2() {
    System.out.print("@Test: teste 2 => ");
  }
}
```

Figura 6.5: Métodos de *set up* e *tear down* do arcabouço TestNG para Java.

```java
// Referências do TestNG
import org.testng.annotations.*;
// Referências do sistema em teste ocultas

public class PilhaComTamanhoLimitadoTests {
  Pilha pilha; // variável utilizada em todos os testes

  /* Set up */
  @Before public void inicializaObjetos() {
    pilha = new Pilha();
  }

  /* Tear down */
  @After public void liberaMemoria() {
    pilha.esvazia();
    pilha = null;
    System.gc(); // Agilizando a execução do Garbage Collector
  }

  // Teste que consome bastante memória
  @Test public void pilhaNaoAceitaMaisElementosDoQueLimiteEstipulado() {
    pilha.setAlturaMaxima(1000);
    for(int i = 0; i < 1000; i++)
      pilha.coloca(i);
    assertEquals(1000, pilha.altura());
    pilha.coloca(i);
    assertEquals(1000, pilha.altura());
  }
}
```

Figura 6.6: Exemplo típico de uso dos métodos *set up* e *tear down*.

em vez de testes de integração. Primeiramente, os testes de unidade solucionam os problemas de baixa testabilidade do sistema em teste. Além disso, todas as outras características de qualidade, descritas na Seção 5.1, são mais facilmente asseguradas quando um cenário de teste é isolado, ou seja, ele tende a ficar mais rápido, independente e repetitível.

O que caracteriza um teste de unidade é justamente o isolamento de um trecho de código do resto do sistema e do ambiente. Isolar um trecho de código significa substituir todas as suas dependências, que podem ter implementações lentas, incompletas ou que prejudicam a testabilidade, por dependências controladas. Dessa maneira, o trecho de código em teste trabalha sob situações ideais, supondo que todas suas dependências estão corretas. Inclusive, essa característica ajuda a dispensar a prática de depuração, pois, se algum teste falhar, fica explícito o trecho de código que está o problema.

No entanto, isolar um trecho de código pode ser uma tarefa complicada. A dificuldade deve-se principalmente à testabilidade do sistema (Seção 10.3). Quanto mais entrelaçado estiverem os módulos em teste, mais difícil será para substituir as dependências por objetos controlados [113].

Dado que um módulo é suficientemente coeso para isolar seu comportamento, é possível que isso seja feito comumente de duas maneiras: a primeira é inserir, errôneamente, lógica de teste no código de produção, que é um indício de problema (Seção 5.2); a segunda, mais elegante, é fazer com que os testes substituam, apenas dentro do seu contexto e durante sua execução, as dependências da funcionalidade em teste por módulos e objetos que apenas respondam o que o cenário de teste espera para poder fazer suas verificações.

No caso de linguagens orientada a objetos, os testes podem substituir os objetos dependentes por objetos equivalentes, mas que possuem o comportamento mínimo desejado para realização do teste. Por exemplo, através de herança, pode-se criar subclasses que sobrescrevem a implementação original por uma simplificada. Nas linguagens em que até os métodos são objetos, é possível, simplesmente, substituí-los por implementações temporárias durante a execução do cenário de teste. Ainda, existem bibliotecas, tais como Mockito, EasyMock e JMock para Java, que criam objetos que seguem o comportamento desejado, sem a necessidade de implementá-los.

Esses objetos que são criados no escopo dos testes para substituir dependências são chamados genericamente de Objetos Dublês, em analogia aos dublês de cinema [99]. Contudo, há diversos tipos de Objetos Dublês (Figura 6.7), que são apropriados para situações específicas. Nas seções 6.4.3 até 6.4.7 há a descrição detalhada dos cinco tipos de Objetos Dublês já descritos por Meszaros: Objeto Tolo, Stub, Falsificado, Emulado e Espião, respectivamente. Na Seção 6.4.8, é descrito um novo tipo que foi identificado pelo autor, o Objeto Protótipo.

A escolha do tipo de Objeto Dublê a ser utilizado depende prioritariamente do que se está querendo verificar, pois nem todos eles podem ser utilizados dentro de um contexto. Por exemplo, somente os Objetos Falsificados e os Espiões são capazes de imitar um algoritmo, ou seja, de fornecer dados dinâmicos para a funcionalidade em teste.

A Tabela 6.1 faz uma comparação dos objetos dublês de acordo com quatro características importantes: (1) se o dublê é exercitado pelo teste, ou seja, se ele influencia diretamente no resultado do teste; (2) se o dublê fornece informações enganosas que influenciam no resultado gerado pela funcionalidade em teste, sejam elas dados estáticos ou gerados dinamicamente por algoritmos simplificados; (3) a capacidade do dublê de armazenar informações sobre o que foi executado, o que permite fazer verificaçoes na forma que um algoritmo é executado e (4) caso o objeto dublê precise ou não seguir uma interface definida, o que pode ser importante para realizar testes de algoritmos reflexivos ou com programação a aspectos.

Depois que se sabe quais padrões são viáveis de serem utilizados, a escolha deve-se basear na simplicidade, ou seja, qual deles torna a implementação do teste mais enxuta e legível. Por exemplo, Se o objeto servirá apenas para evitar erros de compilação ou erros indiretos de execução, ou seja, ele não será exercitado diretamente pelo teste, então Objeto Tolo é o que deve ser escolhido porque é o mais fácil de implementar. No entanto, não existe uma regra para isso, pois varia de acordo com as particularidades

de cada contexto e das ferramentas disponíveis de objetos dublês. Mais detalhes e exemplos são podem ser encontrados na Seção 6.4.

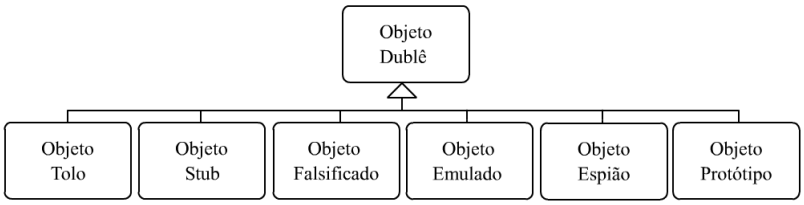


Figura 6.7: Tipos de Objetos Dublês.

Características / Objeto Dublê	Exercitado pelo Teste	Fornecimento de Dados para o SUT	Armazena Informações da Execução	Interface Predefinida
Objeto Tolo (Dummy Object)	Não	Não Fornece	Não	Sim
Objeto Stub (Test Stub)	Sim	Estático ou Dinâmico	Não	Sim
Objeto Falsificado (Fake Object)	Sim	Estático ou Dinâmico	Não	Sim
Objeto Emulado (Mock Object)	Sim	Estático	Sim	Sim
Objeto Espião (Spy Object)	Sim	Estático ou Dinâmico	Sim	Sim
Objeto Protótipo	Sim	Estático	Não	Não

Tabela 6.1: Comparação de algumas características dos Objetos Dublês.

6.3 Boas Práticas de Automação

A automação de testes pode ter diversos problemas de código-fonte, comportamento e projeto, como foi discutidos na Seção 5.2. Entretanto, muitos desses problemas podem ser facilmente evitados através da utilização de boas práticas de automação de testes. Algumas das boas práticas mais gerais, úteis para quaisquer tipos de teste, já foram discutidas no Capítulo 4. Agora, serão abordadas boas práticas direcionadas para testes de unidade.

6.3.1 Código-Fonte

Apesar de que o código-fonte dos testes precisa receber a mesma atenção dedicada ao código-fonte do sistema em teste (Seção 5.1), não é necessário seguir, rigorosamente, todas as boas práticas de programação conhecidas pela equipe de desenvolvimento. Por exemplo, algumas práticas de otimizações ou de modularização podem trazer muitos benefícios para o sistema, tornando-o mais rápido e flexível, mas elas podem prejudicar outras características, como a clareza e simplicidade.

Para os testes, o ideal é que seu código seja o mais simples, enxuto e legível possível. Deve-se encontrar o equilíbrio entre o uso de todo o poder das linguagens de programação com a clareza e simplicidade de um texto em linguagem natural. Para encontrar esse equilíbrio, algumas boas práticas podem ajudar, como serão descritas abaixo.

Sem Rigores das Linguagens de Programação: Muitas linguagens permitem atribuir muitas propriedades a uma variável, método ou classe. Por exemplo, em Java, é possível definir uma variável como pública (*public*), privada (*private*), protegida (*protected*), de classe (*static*), constante (*final*) etc. Todas essas propriedades podem ser muito importantes para a arquitetura do sistema, mas, para os testes, elas são dispensáveis, pois elas apenas poluem o código-fonte com complexidade desnecessária. O *design* do código-fonte dos testes deve ser tão simples a ponto de não necessitar desses rigores de arquitetura. Os casos excepcionais são quando as ferramentas utilizadas necessitarem de alguma propriedade específica. Por exemplo, o JUnit requer que todos os métodos de teste sejam públicos.

Ainda, as linguagens podem oferecer diferentes convenções de código-fonte, sendo que algumas são mais rígidas do que as outras. Por exemplo, em C o padrão ANSI é mais rígido do que o POSIX. No caso de Java, é possível configurar ferramentas auxiliares e IDEs para definir quais são as convenções que devem ser seguidas. Para os testes, deve-se sempre optar pelas convenções menos controladoras, que possibilitem a criação de código-fonte menos rigoroso.

Nomes Auto-Explicativos: Os nomes das variáveis, métodos e classes podem seguir convenções, desde que eles sejam claros e auto-explicativos. Deve-se dar preferência a nomes completos e longos do que siglas e abreviações que tornem o significado não-óbvio.

Linguagem Ubíqua da Equipe: Além dos nomes deverem ser auto-explicativos, é preferível que eles utilizem uma linguagem ubíqua entre clientes, programadores, testadores etc. Ainda, deve ser evitado o uso de sinônimos para diminuir o vocabulário utilizado pelo projeto.

Sem Variáveis Globais Mutáveis: Variáveis globais, compartilhadas e de classe devem ser evitadas tanto no sistema como nos testes, mas há situações em que elas trazem muitos benefícios para as aplicações. No entanto, para os testes, elas devem sempre ser evitadas, pois elas não só aumentam a complexidade dos testes, como também favorecem à criação de casos de testes dependentes, o que os tornam mais difíceis de serem paralelizados.

Arquitetura Simples: Por mais que um *design* de arquitetura possa tornar o código-fonte mais flexível e diminuir a replicação de código, ele também pode tornar o código-fonte mais difícil de ser entendido. Flexibilidade e não-replicação de código também são importantes para os testes, mas deve haver um equilíbrio com a simplicidade. Boas técnicas de orientação a objetos, *design* e arquitetura também devem ser utilizadas ao escrever os cenários de testes. Contudo, devido à própria simplicidade do código dos testes automatizados e do uso de arcabouços, não deve ser grande o esforço para criar baterias organizadas de testes automatizados. De maneira geral, o *design* das classes de testes não deve ser muito mais complexo do que simples relacionamentos de herança e colaboração.

Sem Otimizações Desnecessárias: Se até para os sistemas as otimizações são recomendadas apenas para as funcionalidades com os gargalos de desempenho, para os testes essa recomendação é ainda mais enfática. De maneira geral, otimização dos testes deve ser feita apenas por ferramentas auxiliares e de maneira transparente, ou seja, sem a necessidade de alterar o código-fonte dos testes. Na prática, essas ferramentas podem executar os testes paralelamente ou com alguma estratégia mais elaborada.

6.3.2 Refatorações Comuns

Como discutido na seção anterior, o código dos testes deve ser organizado, claro, legível e sem replicação. Quando a implementação não atende a esses requisitos, ainda é possível melhorá-la através de refatoração, que deve ser uma tarefa rotineira da automação dos testes.

Existem diversos estudos sobre refatoração de código de teste com o intuito de melhorar a qualidade de testes já implementados [53, 143]. Entretanto, mesmo durante a criação de novos cenários de testes é comum a realização de refatorações, principalmente para facilitar a adição de novos cenários de verificação. A seguir há breves comentários das refatorações que são mais frequentemente utilizadas durante a automação de testes. Por causa da modelagem simples do código dos testes, as refatorações mais usadas estão entre as mais simples [59, 122].

Extrair Método: Muitos métodos de testes são parecidos, então é comum extração de fragmentos para métodos auxiliares ou até mesmo para os métodos de *set up* e *tear down*.

Extrair Classe: Quando uma classe de teste começa a ficar muito extensa, com muitos métodos auxiliares e de teste, pode ser um sinal que ela precisa ser dividida, ou, até mesmo, que as classes do sistema que estão sendo testadas possuem muitas responsabilidades. O mesmo raciocínio pode ser empregado quando os métodos de *set up* e *tear down* estão muito extensos e complexos.

Extrair Superclasse: Quando diversas classes de testes possuem métodos de *set up* ou *tear down* semelhantes, pode ser um indício de que uma classe base para os testes pode ser criada para evitar replicação de código.

Renomear Classes, Métodos e Variáveis: As refatorações anteriores de extração sempre produzem novas variáveis e classes, logo, os nomes podem não mais fazer sentido, logo, precisam ser renomeados. Não obstante, os nomes do sistema em teste que forem renomeados também precisam ser refletidos nos testes, no entanto, as ferramentas de refatoração não conseguem automatizar essa tarefa, portanto, é necessário uma preocupação adicional quanto a isso.

Introduzir Variável Explicativa: Alguns cenários de testes podem criar cenários complexos e não intuitivos. Para melhorar a legibilidade do código dos testes, pode-se adicionar variáveis temporárias com nomes auto-explicativos para substituir expressões complexas.

6.3.3 Orientação a Objetos

Programas orientados a objetos possuem um bom potencial para alta testabilidade [117, 97, 27], principalmente por causa das facilidades de modularização através de classes, heranças e relacionamentos. Entretanto, vários detalhes de implementação dos objetos podem influenciar no modo que os testes são realizados.

Basicamente, o que precisa ser testado em um objeto é sua interface, ou seja, tudo que é exposto para o resto da aplicação. A interface pública de um objeto pode ser composta por variáveis, métodos e classes internas, todas sujeitas a erros de implementação. Entretanto, o mais comum é testar os métodos do objeto e das classes internas. Testes de variáveis são apenas interessantes para o caso de constantes ou para verificar se elas foram inicializadas corretamente.

Quando as funcionalidades públicas de um objeto estão corretas, há bons indícios que toda a implementação interna do objeto também está. Já quando o que está querendo ser testado é alguma funcionalidade interna (privada), é preciso ficar atento, pois isso é um indício de que um conjunto de objetos não possuem um bom *design*. Essa situação é um exemplo típico de como os testes automatizados provêm informações para ajudar com a criação do *design* do sistema.

Nessa situação, o primeiro passo é avaliar se o trecho em teste pode ser movido para algum outro objeto mais coerente, de modo que a funcionalidade se torne pública. Dado que o *design* está de acordo com as necessidades do sistema, então basta realizar os testes desejados. O empecilho é que, devido às limitações das linguagens, pode ser impossível os testes conseguirem fazer as chamadas das funcionalidades privadas. Em alguns casos, o uso de reflexão pode ser suficiente para chamada da funcionalidade. Se ainda assim não for possível, a solução é alterar a visibilidade da funcionalidade ou então tentar testá-la de forma indireta, através de uma funcionalidade pública que a utilize.

Para classes anônimas, o pensamento é análogo ao de métodos privados. Primeiro deve-se avaliar se a classe precisa ser nomeada. Caso a classe precise mesmo ser anônima, então a solução é testá-la de forma indireta, ou seja, através da funcionalidade que a utiliza.

Ja para as classes abstratas que não podem ser instanciadas, a solução é mais simples, basta criar uma subclasse concreta que não sobrescreva as funcionalidades da classe em teste. Para não poluir o código do sistema, essa classe deve ser acessível apenas aos testes. No entanto, essa abordagem só é coerente se o sistema respeita o Princípio de Substituição de Liskov[4] [89]. Através dessa classe concreta

[4]Esse princípio diz que uma instância de uma subclasse deve ser capaz de substituir qualquer instância da classe pai sem qualquer dano ao sistema.

é possível verificar o código implementado na classe pai, mas não é possível verificar se os métodos abstratos estão sendo chamadas corretamente. Para fazer essas verificações, o recomendado é utilizar Objetos Dublês, tais como Objeto Emulado (*Mock Object*, Seção 6.4.6) ou Objeto Espião (*Spy Object*, Seção 6.4.7).

Quanto aos recursos globais e mutáveis, eles devem ser evitados sempre que possível, assim como também é recomendado em programas procedimentais. No caso dos objetos, dois exemplos comuns de recursos compartilhados são variáveis de classe ou objetos *Singleton*. Todos recursos compartilhados dificultam a automação dos testes e impedem a execução paralela dos testes. Quando não é possível refatorar o sistema em teste, então as baterias de testes devem ser executadas sequencialmente e, para evitar testes intermitentes, cada conjunto de teste deve ser responsável por atualizar o recurso compartilhado de acordo com suas necessidades.

Não obstante, as implementações dos Padrões de Projetos [61] também devem ser testadas, mesmo que os programadores estejam bem familiarizados com o *design* dos objetos. Inclusive, também é possível pensar em recomendações para os testes das implementações dos padrões.

Por exemplo, para os padrões *Singleton* e *Flyweight* é recomendado que sejam verificadas que as instâncias dos objetos gerados sejam sempre as mesmas (`assertSame`), em oposição às instâncias obtidas através dos padrões *Prototype*, *Builder* e *Factory Method* (`assertNotSame`). Já os padrões *Chain of Responsability* e *Composite* lidam com coleções de objetos, então é importante realizar testes com coleções vazias e com elementos.

Ainda, podem ser utilizados Objetos Dublês (6.2) para testar alguns Padrões de Projeto. Por exemplo, *Mediator*, *Interpreter*, *Adapter* e *Decorator* podem exigir o uso de Objetos Espiões para verificarem que as saídas indiretas de dados estão corretas. Não obstante, *Template Method* e *Abstracty Factory* podem utilizar Objetos Falsificados para ajudar a testar o trecho concreto das implementações. Outros padrões podem ser mais simples de serem testados por serem intrinsicamente mais coesos, tais como *Strategy*, *Command* e *State*.

6.3.4 Orientação a Aspectos

Programação Orientada a Aspectos (POA) é um novo paradigma de desenvolvimento que serve de complemento à Orientação a Objetos. Os Aspectos fornecem uma nova maneira de encapsular as funcionalidades que são comuns a diversos objetos e difíceis de serem isoladas [150]. O código-fonte da funcionalidade que foi modularizada em um Aspecto é inserido em objetos pré-definidos, sendo que esse processo pode ser feito em tempo de compilação ou execução, dependendo das ferramentas utilizadas.

A POA é uma técnica poderosa, mas a criação de testes automatizados, tanto para os Aspectos como para os sistemas que os utilizam, é uma tarefa que requer novas abordagens. Um Aspecto não possui uma identidade independente, pois ele depende de um objeto para existir; ou seja, ele não pode ser instanciado diretamente. Entretanto, esse novo tipo de estrutura de dado pode conter diversos tipos de erros, inclusive erros graves como de laços infinitos. Por isso, é fundamental que eles sejam muito bem testados.

Existem muitos estudos e estratégias para se testar programas orientados a aspectos, sendo que alguns deles sugerem testes de unidade [86, 91] e outros de integração [102, 83, 24, 106]. O teste de unidade define o próprio Aspecto como sendo uma unidade, e com a ajuda de ferramentas apropriadas (Jaml-Unit), é criado uma maneira de fazer as verificações diretamente. Já os testes de integração verificam o comportamento dos objetos que receberam código-fonte de Aspectos [150]. Contudo, o termo integração é confuso para esse caso, pois é possível realizar testes de unidade para as classes que possuem código de Aspectos, que será a estratégia abordada nessa seção.

Essa estratégia nada mais é do que testar sistemas orientados a aspectos como se fossem simplesmente orientado a objetos [152]. Dessa maneira, todas as práticas, padrões e técnicas conhecidas de testes OO também podem ser utilizadas, inclusive o uso do Objeto Protótipo (Seção 6.4.8). Essa estraté-

gia é coerente, pois um bom teste automatizado não testa como uma funcionalidade foi implementada, mas sim o que ela deve fazer.

Um Aspecto é composto de Pontos de Atuação (*Pointcuts*) e Adendos (*Advices*), sendo que ambos precisam ser testados. As duas subseções seguintes discutirão as estratégias de cada um deles.

Pontos de Atuação (*Pointcuts*)

Os Pontos de Atuação são expressões que mapeiam os locais do sistema onde serão inseridos os trechos de código-fonte do aspecto (Adendos). Essas expressões são análogas às expressões regulares, que possuem padrões de caracteres que devem ser encontrados em *strings*. Um padrão definido de forma errada pode adicionar ou remover pontos de atuação importantes [85]. Por isso, por mais que existam IDEs que ajudam na criação dos Pontos de Atuação, é fundamental ter uma bateria de testes automatizados para evitar erros de distração e de regressão.

Para testar esses pontos, pode ser utilizado Objetos Protótipo (Seção 6.4.8) que contenham trechos de código que devem e que não devem ser encontrados pelas expressões do Ponto de Atuação, justamente para verificar os cenários positivos e negativos. Para fazer as verificações, pode ser utilizado Objetos Espiões ou Emulados, que possibilitam verificar se eles foram ou não exercitados. Consequentemente, deve ser possível injetar os Objetos Dublês para que os testes possam ser realizados.

Outra solução é a utilização de arcabouços que ajudam a testar os Pontos de Atuação, como o APTE (*Automated Pointcut Testing for AspectJ Programs*) [5]. Ele recebe um conjunto de aspectos e classes e devolve duas listas: uma dos Pontos de Junção que satisfazem os Pontos de Atuação dos aspectos; e outra contendo os Pontos de Junção que quase satisfazem os Pontos de Atuação, que são casos interessantes de serem analisados em busca de erros. Dessa forma, basta criar um teste automatizado que mande executar essa ferramenta e que verifique se as listas contém ou não os métodos esperados. Essa solução dispensa o uso de Objetos Dublês, mas ao mesmo tempo requer o uso de uma ferramenta externa a dos testes.

Adendos (*Advices*)

Já os Adendos são os trechos de código que serão acrescentados em todos os Pontos de Junção (*Join points*) do sistema, que são os locais que satisfazem os Ponto de Atuação do Aspecto. Isso implica que erros de implementação em seu código são espalhados em diversos pontos da aplicação. Consequentemente, Aspectos mal implementados podem causar danos catastróficos ao sistema. Por isso, é fundamental a criação de uma bateria de testes automatizados muito cuidadosa.

A primeira recomendação para criação desses testes é a utilização do padrão Objeto Humilde (Seção 6.4.2) para separar a lógica principal de outros detalhes de implementação não testáveis. Por exemplo, em Java com AspectJ, parte do código do Adendo pode estar escrito em Java (testável) e parte na linguagem do AspectJ (não testável de forma direta sem a ajuda de arcabouços próprios).

O código em Java pode ser encapsulado em um objeto comum e testado como qualquer outro. Quanto ao restante da implementação, é recomendável que os testes sejam realizados, novamente, através de Objetos Protótipo que receberam o código do Adendo. A princípio, pode parecer que os testes estão verificando o Objeto Dublê e não o sistema em teste, o que seria um erro, mas essa estratégia é coerente, pois o código adicionado ao Objeto Dublê é idêntico ao código adicionado ao sistema em teste. Um exemplo de teste com Aspectos que engloba todos essas recomendações pode ser visto na Seção 6.4.8.

Ainda, caso o sistema possua diversos Aspectos que serão aplicados em um mesmo Ponto de Junção, pode-se criar situações de teste para verificar se existe incompatibilidade entre eles [119]. Para isso, basta criar um Objeto Protótipo que satisfaz os Pontos de Atuação correspondentes.

Também é importante notar que existem diversos tipos de Aspectos. Alguns apenas coletam informações do sistema e não influenciam o resto da implementação do objeto, enquanto outros podem

alterar o fluxo de controle ou até mesmo alterar os valores de variáveis dos objetos [121]. Contudo, essa estratégia permite criar testes para quaisquer dessas situações, pois pensando no objeto como uma caixa preta, ele não é diferente. Apenas é importante ressaltar que o código dos Adendos deve permitir que os objetos colaboradores possam ser injetados.

6.3.5 Reflexão

Reflexão é a capacidade de um programa observar ou alterar sua própria estrutura ou comportamento [58]. A técnica de programar utilizando essa capacidade tem sido vastamente utilizada, principalmente por bibliotecas, APIs e *engines*. Bons exemplos são os arcabouços para programas Web, tais como os populares Django (Python), Rails (Ruby), Grails (Groovy), Spring e Hibernate (Java).

No entanto, assim como a programação orientada a aspectos, as funcionalidades que utilizam reflexão processam parte do próprio código do sistema, por exemplo, por meio de *bytecodes*. Em alguns casos, as regras de uma funcionalidade se baseiam em algumas características do código, enquanto, em outras situações, novas funcionalidades são adicionadas a objetos em tempo de execução. Para ambos os casos podem haver diversos problemas, não só de correção, como também de segurança e desempenho.

Ainda, devem ser consideradas as inúmeras possibilidades de se implementar um mesma solução computacional, portanto, mesmo os algoritmos reflexivos mais simples devem se preocupar com muitos detalhes. Não obstante, o código-fonte de sistemas pode estar em constante evolução, seja através de refatorações, correções ou da adição de novas funcionalidades. Cada mudança do código-fonte pode quebrar os algoritmos reflexivos, o que os tornam funcionalidades muito suscetíveis a erros de regressão. Por causa desses fatores, a automação de testes é uma boa solução para garantir a qualidade dessas funcionalidades.

As recomendações para criação de bons testes automatizados para os algoritmos reflexivos são parecidas aos de programação orientada a aspectos. Primeiramente, por mais que esses algoritmos não tenham uma estrutura definida, é uma boa prática de programação separar as responsabilidades de reconhecimento de um padrão de código-fonte (análogo aos Pontos de Atuação dos Aspectos) das tarefas que serão realizadas no momento oportuno (análogo aos Adendos dos Aspectos).

Para os testes das funcionalidades que reconhecem padrões de código-fonte, podem ser utilizados Objetos Falsificados (Seção 6.2) ou Objetos Protótipo (Seção 6.4.8). Já para as tarefas que serão executadas, pode-se utilizar os padrões Objeto Humilde (Seção 6.4.2).

6.3.6 Módulos Assíncronos

Hoje, a importância de sistemas assíncronos é muito grande devido à Internet, à grande modularização dos sistemas e ao uso de serviços. Além disso, a tendência é tornar os sistemas cada vez mais independentes e velozes, devido à criação e popularização dos processadores com vários núcleos e das linguagens de programação que facilitam a escrita de sistemas altamente escaláveis, tais como Erlang, Haskell e Scala.

Trechos de código que envolvem programação paralela ou distribuída são complexos e, por isso, muito suscetíveis a erros. Há muitos pontos que precisam ser verificados quando um sistema utiliza outros processos, *threads* ou atores, tais como a sincronização e a comunicação através do compartilhamento de memória ou da troca de mensagens. Erros típicos de programação concorrente como os de sincronização, *deadlocks*, *livelocks*, *starvation* e *race conditions* podem quebrar regras de negócio, tornar dados inconsistentes e até mesmo deixar o sistema inteiro inutilizável.

No entanto, os testes de unidade tem como premissa básica a execução síncrona do sistema em teste. Apesar da grande importância dos módulos assíncronos em sistemas de software, a maioria das ferramentas e arcabouços de testes automatizados não possuem funcionalidades que facilitam a implementação de casos de teste de qualidade.

Contudo, criar testes automatizados para esses sistemas é possível, apesar de ser muito mais complexo. Ao contrário de uma bateria de testes sequenciais, a bateria de testes de módulos assíncronos é executada em uma thread diferente da do sistema em teste. Como a execução do sistema testado depende do sistema operacional e do escalonador de processos, não é possível prever, com exatidão, quando o sistema será executado. Dessa forma, os testes não recebem automaticamente os efeitos colaterais do sistema, logo, é preciso que eles sejam sincronizados, ou então, que observem quando o sistema foi alterado, para então fazer as verificações no seu tempo.

Para sincronizar os testes é necessário o mesmo conhecimento e cautela da sincronização do sistema em teste. Já para observar o sistema, os testes podem obter informações de seu estado em um momento oportuno, ou então, eles podem capturar os eventos emitidos pelo sistema. Todas essas alternativas são muito propícias a criarem testes com muitos antipadrões, que geram os problemas de testes difíceis de manter e escrever, falsos positivos e negativos, testes pouco legíveis, intermitentes e lentos.

A primeira boa prática para criar bons testes de módulos assíncronos é utilizar o padrão Objeto Humilde (Seção 6.4.2) para isolar a lógica de negócios da lógica computacional que gerencia a linha de execução da funcionalidade. A lógica de negócios deve ser testada como qualquer outra parte do sistema. Quanto ao restante da funcionalidade, deve ser utilizado o padrão *Assert Eventually* [60], que verifica periodicamente se a funcionalidade assíncrona terminou de ser executada. Quando o padrão identifica que o teste está pronto para ser testado, então são feitas as verificações da correção. Caso o sistema demore muito para ser executado, então o padrão devolve um erro de tempo esgotado.

6.4 Padrões

A seguir serão descritas soluções de testes automatizados de unidade que podem ser aplicadas em diferentes contextos. Entretanto, algumas das recomendações podem ser generalizadas para outros tipos de teste. Vale ressaltar que todos os padrões serão definidos segundo o esqueleto exibido na Seção 5.3.

6.4.1 Injeção de Dependência (*Dependency Injection*)

Tipo: Testabilidade

Quando utilizar: Em sistemas orientado a objetos. Idealmente, em todos objetos do sistema.

Intenção: Desacoplar os objetos e facilitar a inserção e substituição das suas dependências.

Motivação: Sistemas muito acoplados são mais difíceis de serem testados [142]. Mais especificamente, objetos que não permitem a substituição de seus objetos colaboradores inviabilizam o uso de Objetos Dublês para criação de testes isolados.

Solução: Um objeto deve estar desacoplado de suas dependências, de modo que haja mecanismos para que todas elas possam ser substituídas por outros objetos do mesmo tipo. Consequentemente, é possível testar o objeto isoladamente com o auxílio de Objetos Dublês.

Consequências: Todos as dependências podem ser inseridas através dos contrutores ou de métodos pertinentes. Ainda, a responsabilidade de instanciar os objetos colaboradores é passada para outros objetos do sistema, ou até mesmo para novos objetos que são criados especificamente para isolar esta responsabilidade, como ocorre quando utilizamos Padrões de Projeto de Criação, tais como *Builder*, *Factory* e *Prototype* [61].

Implementação: Para variáveis privadas de instância, a injeção das dependências pode ser feita mais comumente através do construtor, ou de métodos `set`. Se a variável for pública, dentro de um contexto, basta atualizá-la diretamente. Ainda, pode ser utilizado arcabouços que são responsáveis por vasculhar e inicializar as variáveis desejadas, como o Spring para Java. Já as dependências não associadas ao objeto em teste podem ser passadas como argumentos para os métodos necessários.

Exemplo - Java/JUnit/Hamcrest: A Figura 6.8 mostra um trecho de código do objeto `Compra` de uma loja que está acoplado às regras de desconto. Nesse exemplo, queremos testar apenas que o valor do subtotal com desconto é o valor do desconto subtraído do subtotal. Entretanto, da maneira que está implementada, não conseguimos realizar esse teste sem conhecer toda a regra de descontos aplicada na compra, isso porque a classe `Compra` e o método que faz o cálculo do subtotal com desconto possuem mais de uma responsabilidade.

Testar funcionalidades que possuem mais de uma responsabilidade é trabalhoso, além de resultar em diversos outros antipadrões de testes automatizados. Isso porque os testes se tornam presos à implementação e não ao comportamento do sistema. A Figura 6.9 mostra um exemplo de teste automatizado para o método `subTotalComDesconto`.

O primeiro passo para melhorar a implementação do objeto `Compra` é isolar as regras de desconto em um objeto própio para isso, como mostra a Figura 6.10. Contudo, essa melhoria não influencia em como os testes do objeto `Compra` serão implementados, isso porque o objeto colaborador `RegraDesconto` ainda não pode ser injetado, dessa maneira, ainda é necessário conhecer as regras de desconto para poder testar o método `subTotalComDesconto`.

Para finalizar a refatoração, é necessário remover a responsabilidade do objeto `Compra` de instanciar as regras de desconto (Figura 6.11), consequentemente, será possível injetá-la tanto através dos testes quanto pelo próprio sistema.

```java
public class Compra {
  Cliente cliente;
  Produtos produtos;
  FormaPagamento formaPagamento;
  Promocao promocao;

  public Compra(Cliente c, Produtos l, FormaPagamento f, Promocao p) {
    this.cliente = c;
    this.produtos = l;
    this.pagamento = f;
    this.promocao = p;
  }

  public Dinheiro subTotal() {
    return produtos.subTotal();
  }

  // Objeto Desconto e as regras estão acoplados ao objeto Compra.
  // Não é possível testar o objeto Compra independente das regras de desconto.
  public Dinheiro subTotalComDesconto() {
    Dinheiro valor = new Dinheiro(0);

    if(cliente.isVIP()) {
      Desconto d = new DescontoVIP();
      valor.add(d.valor(this));
    }
    else {
      Desconto d = new DescontoPelaFormaDePagamento();
      valor.add(d.valor(this));
      valor.add(promocao.desconto().valor(this));
    }

    return subTotal().subtract(valor);
  }
}
```

Figura 6.8: Objeto `Compra` com implementação acoplada ao objeto `Desconto`.

```java
//Dependências do JUnit + Hamcrest
import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.*;
import static org.hamcrest.Matchers.equalTo;

public class CompraTests {
  // Objetos pertinentes ao teste
  Produtos produtos;
  Cliente cliente;
  FormaPagamento pagamento;
  Promocao promocao;
  Compra compra;

  @Before
  public void setUp() {
    produtos = new Produtos();
  }

  // Para realizar este teste é preciso conhecer as regras de desconto.
  // Um teste simples fica extenso e difícil de entender.
  @Test public void totalComDescontoDeveSubtrairDescontoDoSubTotal() {
    // Detalhes de implementação das regras de desconto.
    boolean vip = true;
    cliente = new Cliente(vip);
    Dinheiro preco = new Dinheiro(100);
    produtos.add(new Produto(preco));
    pagamento = new BoletoFormaPagamento(); // 3% de desconto
    promocao = new PromocaoDeNatal(); // 10% de desconto

    compra = new Compra(cliente, produtos, pagamento, promocao);

    // 100 - (3 + 10 + 15) = 72
    assertThat(compra.subTotalComDesconto(), equalTo(new Dinheiro(72)));
  }

  // As regras de desconto deverão ser testadas a partir do objeto Compra.
  // Deverão ser feitos testes para cliente não-VIP, com pagamento por cartão etc.
}
```

Figura 6.9: Teste complicado do objeto `Compra`.

```java
public class Compra {
  Cliente cliente;
  Produtos produtos;
  FormaPagamento formaPagamento;

  public Compra(Cliente c, Produtos l, FormaPagamento f) {
    this.cliente = c;
    this.produtos = l;
    this.formaPagamento = f;
  }

  public Dinheiro subTotal() {
    return produtos.subTotal();
  }

  // As regras de desconto estão isoladas no objeto RegraDeDescontoPadrao,
  // mas ainda assim as regras estão implicitamente acopladas ao objeto Compra.
  public Dinheiro subTotalComDesconto() {
    RegraDesconto regraDesconto = RegraDescontoPadrao();
    Dinheiro valorDesconto = regraDesconto.calcula(this);

    return subTotal().subtract(valor);
  }
}
```

Figura 6.10: Objeto `Compra` com implementação mais organizada, mas ainda acoplada ao objeto `Desconto`.

A testabilidade está relacionada com a flexibilidade do sistema, portanto, se está complicado de testar, é um indício de que o sistema precisa ser refatorado. Com essa nova modelagem, o sistema poderá trabalhar com diversas regras de desconto simultâneamente e os testes ficam legíveis e fáceis de implementar (Figura 6.12).

Padrões Relacionados: O padrão Objeto Humilde (Seção 6.4.2), que também é utilizado para separar as responsabilidades de um objeto, pode ser utilizado para desacoplar os objetos e facilitar a injeção de dependência. Ainda, esse padrão é pré-requisito para os padrões Objeto Emulado (*Mock Object*, Seção 6.4.6), Objeto Falsificado (*Fake Object*, Seção 6.4.5) e Objeto Espião (*Test Spy*, Seção 6.4.7).

Usos Conhecidos: O arcabouço *Spring* para Java é uma das ferramentas mais populares de injeção de dependência. Outro uso conhecido que merece destaque é a API EJB para Java. Na Seção 10.3 há outros exemplos relacionados.

Referências: Existem livros específicos sobre injeção de dependência [113] e outros de catálogo de padrões que também o descrevem [99].

```java
public class Compra {
  Cliente cliente;
  Produtos produtos;
  FormaPagamento formaPagamento;
  RegraDesconto regraDesconto;

  // É possível injetar as dependências.
  // Objeto Compra e Desconto não estão mais acoplados.
  public Compra(Cliente c, Produtos p, FormaPagamento f, RegraDeDesconto r) {
    this.cliente = c;
    this.produtos = p;
    this.formaPagamento = f;
    this.regraDesconto = r;
  }

  public Dinheiro subTotal() {
    return produtos.subTotal();
  }

  public Dinheiro subTotalComDesconto() {
    Dinheiro valor = regraDesconto.calcula(compra);
    return subTotal().subtract(valor);
  }
}
```

Figura 6.11: Objeto `Compra` desacoplado de suas dependências.

```java
// Dependências do JUnit + Hamcrest
import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.*;
import static org.hamcrest.Matchers.*;

public class CompraTests {
  // Objetos Tolos
  Cliente cliente = new Cliente();
  FormaPagamento pagamento = new BoletoFormaPagamento();

  // Objetos pertinentes ao teste
  Produtos produtos;
  Compra compra;
  // Objeto Dublê: Objeto Falsificado
  RegraDesconto regraDesconto = new RegraDesconto() {
    public Dinheiro calcula(Compra compra) {
      return new Dinheiro(15);
    }
  }

  @Before
  public void setUp() {
    produtos = new Produtos();
  }

  // Valor total com desconto depende somente do valor dos produtos e
  // do valor total do desconto.
  @Test public void totalComDescontoDeveSubtrairDescontoDoSubTotal() {
    Dinheiro preco = new Dinheiro(100);
    produtos.add(new Produto(preco));

    compra = new Compra(cliente, produtos, pagamento, regraDesconto);

    // 100 - 15 = 85
    assertThat(compra.subTotalComDesconto(), equalTo(new Dinheiro(85)));
  }

  // ... As regras de desconto também devem ser testadas, mas isoladamente e
  // em outra classe de teste.
}
```

Figura 6.12: Teste do objeto `Compra` refatorado.

6.4.2 Objeto Humilde (*Humble Object*)

Tipo: Testabilidade

Quando utilizar: Sempre que o objeto em teste possui mais de uma responsabilidade, mas, principalmente, quando é difícil testá-lo devido ao seu acoplamento com arcabouços ou até mesmo a objetos complexos. São exemplos comuns os objetos que possuem processos assíncronos, ou que interagem com requisições Web e gerenciadores de bancos de dados.

Intenção: Uma boa prática de orientação a objetos é que cada objeto tenha apenas uma responsabilidade. A intenção desse padrão é justamente partir um objeto complexo em objetos simples e coesos (objetos humildes).

Motivação: Objetos com muitas responsabilidades são difíceis de serem testados. Primeiramente, porque a inicialização do objeto pode ficar mais complexa, assim como os métodos de preparação do teste (*set up*). Além disso, os próprios métodos de testes tendem a ficar mais extensos e difíceis de implementar, pois quanto mais acoplado um objeto está do resto do sistema, maior será o trabalho para executar um cenário de teste de modo isolado. Não obstante, quanto mais responsabilidades um objeto possui, mais verificações são necessárias para avaliar a correção da implementação.

Solução: Refatorar objetos para que eles tenham apenas uma responsabilidade. Em particular, é necessário separar a lógica testável de um objeto dos aspectos técnicos e complexos de arcabouços e do ambiente.

Consequências: Após a refatoração, os objetos ficam mais simples e coesos, pois possuem apenas uma responsabilidade. Além disso, a lógica de negócios fica desacoplada da lógica de infraestrutura, tal como lógica de programação assíncrona, persistência de dados etc. Isso resulta em um sistema mais flexível, com objetos coesos, desacoplados e com alta testabilidade.

Implementação: Esse padrão descreve simplesmente uma boa prática de orientação a objetos, independentemente do sistema ter ou não testes automatizados. Sendo assim, não há uma implementação sistematizada para esse padrão. Qualquer padrão arquitetural ou de projeto pode descrever a solução apropriada para melhorar a testabilidade dos objetos, assim como todas as técnicas de refatoração podem ser úteis. No entanto, como as responsabilidades de um objeto serão divididas entre objetos menores, é natural a utilização de refatorações de extração, tais como *Extract Class* e *Extract Method*.

Exemplo - Python/Django: A Figura 6.13 possui um trecho de código de uma aplicação Web com o arcabouço Django para buscar pessoas por parte do nome. Essa funcionalidade recebe uma requisição Web e devolve a resposta apropriada. No entanto, esse método também possui a responsabilidade de gerar a *query* que será executada no banco de dados para encontrar os resultados, ou seja, além do método ter mais de uma responsabilidade, ele não segue a arquitetura MVC proposta pelo arcabouço.

Essa falha de modelagem reflete na qualidade dos testes automatizados. Para testar a busca realizada no banco de dados é necessário lidar com objetos de requisição e de resposta Web. Para testar apenas a *query* gerada, pode-se separar as responsabilidades em objetos distintos, como é mostrado na Figura 6.14. Nesse caso, o objeto `PessoaManager` é o Objeto Humilde.

Padrões Relacionados: Caso sejam feitos alguns testes nos objetos que contêm os detalhes complexos de arcabouços, o padrão Injeção de Dependência (Seção 6.4.1) pode ser utilizado para substituir o Objeto Humilde por dublês.

```python
# Funcionalidade para buscar pessoas por parte do nome.
def busca_pessoa_pelo_nome(request):
    if request.method == 'POST':
        pessoas = Pessoa.objects.filter(nome__icontains=request.POST['texto_busca'
            ])
    else:
        pessoas = []
    return HttpResponse('/pessoas-encontradas', {'pessoas': pessoas})
```

Figura 6.13: Exemplo de funcionalidade com muitas responsabilidades.

```python
# Objeto Humilde: Para testar, basta chamar o método
# Pessoa.objects.com_parte_do_nome com uma string desejada.
# Os testes não precisam mais lidar com objetos de Request e Response.
class PessoaManager(models.Manager):
    def com_parte_do_nome(texto_busca):
        return self.filter(nome__icontains=texto_busca)


# Método refatorado
def busca_pessoa_pelo_nome(request):
    if request.method == 'POST':
        pessoas = Pessoa.objects.com_parte_do_nome(request.POST['texto_busca'])
    else:
        pessoas = []
    return HttpResponse('/pessoas-encontradas', {'pessoas': pessoas})
```

Figura 6.14: Funcionalidade de busca de pessoas refatorada, utilizando um Objeto Humilde.

Referências: Esse padrão foi identificado por Meszaros [99].

6.4.3 Objeto Tolo (*Dummy Object*)

Tipo: Testabilidade

Quando utilizar: Quando for necessário lidar com objetos que não são utilizados pelo cenário de teste, mas que são fundamentais para sua execução. Por exemplo, para evitar erros de compilação.

Intenção: Apenas viabilizar a execução de um teste.

Motivação: Para realização de um caso de teste, pode ser necessário a instanciação de vários objetos, mas nem sempre todos eles são utilizados diretamente pela funcionalidade em teste. Alguns desses objetos são necessários apenas para evitar erros de compilação ou de execução de outras funcionalidades que não estão sob verificação.

Solução: Substituir os objetos colaboradores que são necessários para a execução de um teste mas que não são processados por objetos nulos ou implementados da maneira mais simples possível.

Consequências: Torna viável a execução dos cenários de teste.

Implementação: Basta substituir os objetos colaboradores dispensáveis por valores nulos ou instâncias implementadas da maneira mais simples e legível possível.

Exemplo - Java/JUnit/Mockito: É mais comum que os Objetos Tolos sejam objetos de tipos primitivos das bibliotecas provenientes da linguagem de programação utilizada (strings, números etc), entretanto, também pode acontecer de serem de tipos definidos pelo próprio sistema em teste. A Figura 6.15 mostra três maneiras comentadas de substituir objetos colaboradores por Objetos Tolos.

Padrões Relacionados: Quando não for possível passar valores nulos e a inicialização do Objeto Tolo se torna complexa, então é recomendado o uso de Objetos Emulados (*Mock Objects*, Seção 6.4.6).

Usos Conhecidos: O uso desse padrão é natural durante a implementação de um caso de teste.

```java
class Pessoa {
  public Pessoa(String nome, Date nascimento) { /* ... */ }
  public int idade() { /* método em teste */ }
}

import java.util.Date;

// Dependências do JUnit + Hamcrest
import org.junit.Test;
import static org.junit.Assert.*;

// Dependências do Mockito
import static org.mockito.Mockito.*;

public class PessoaTests {

  @Test public void idadeDeUmaPessoaQueNasceuHojeRetornaZero_versao1() {
    // A String nome recebe um valor nulo.
    // Se o nome receber algum processamento, como validação de dados,
    // essa abordagem se torna inviável.
    Pessoa pessoa = new Pessoa(null, new Date());
    assertEquals(0, pessoa.idade());
  }

  @Test public void idadeDeUmaPessoaQueNasceuHojeRetornaZero_versao2() {
    // A String "Um nome qualquer" é um Objeto Tolo.
    // O cálculo da idade não deve depender do nome da pessoa.
    Pessoa pessoa = new Pessoa("Um nome qualquer", new Date());
    assertEquals(0, pessoa.idade());
  }

  @Test public void idadeDeUmaPessoaQueNasceuHojeRetornaZero_versao3() {
    // É interessante deixar claro quando um objeto não deve
    // interferir no teste.
    // A biblioteca Mockito fornece alguns métodos com esse propósito,
    // tais como o anyString, anyObject, anyInt...
    Pessoa pessoa = new Pessoa(anyString(), new Date());
    assertEquals(0, pessoa.idade());
  }
}
```

Figura 6.15: Exemplo de Objeto Tolo.

6.4.4 Objeto Stub (*Test Stub*)

Tipo: Testabilidade

Quando utilizar: Quando os dados obtidos de objetos colaboradores influenciam e dificultam a criação de testes automatizados para uma funcionalidade.

Intenção: Substituir objetos colaboradores que são difíceis de serem manipulados por versões que possam ser controladas. Dessa maneira, o objeto pode ser configurado para construir diferentes cenários de teste.

Motivação: Muitos objetos colaboradores são difíceis de serem manipulados, consequentemente, os testes se tornam difíceis de serem realizados. Por exemplo, os que envolvem o relógio do computador, datas etc.

Solução: Criar objetos que são fáceis de serem manipulados para substituir o comportamento que prejudica a testabilidade.

Consequências: A testabilidade do sistema é melhorada, o que possibilita a simulação de diversos cenários de teste.

Implementação: Deve-se criar uma variação do objeto colaborador, seguindo a mesma interface, mas de modo que ele seja capaz de retornar dados controlados para a funcionalidade em teste. Então esse objeto deve ser injetado no objeto em teste.

Padrões Relacionados: O padrão Injeção de Dependência (*Dependency Injection*, Seção 6.4.1) é necessário para injetar o Objeto Stub no objeto em teste. Já o Objeto Emulado (*Mock Object*, Seção 6.4.6) também atua como esse padrão fornecendo dados estáticos para o objeto em teste, no entanto, o Objeto Emulado também possui funcionalidades que permitem verificar chamadas indiretas da funcionalidade em teste.

Usos Conhecidos: A solução proposta por esse padrão surgiu antes da solução proposta pelos Objetos Emulados [28, 93].

6.4.5 Objeto Falsificado (*Fake Object*)

Tipo: Testabilidade

Quando utilizar: O Objeto Falsificado é uma solução elegante para realização de testes difíceis de serem simulados. Por exemplo, quando queremos verificar o comportamento do sistema quando ocorre problemas de hardware, rede, sistemas de arquivos etc. Esse padrão também é útil para resolver problemas de testes de partes do sistema que dependem de módulos intrinsicamente lentos.

Entretanto, os Objetos Emulados (Seção 6.4.6) também servem para solucionar esses problemas, com a vantagem de que são mais fáceis de serem utilizados do que a solução proposta por este padrão. Sendo assim, esse padrão só deveria ser utilizado quando não existir uma biblioteca de Objetos Emulados apropriada para as tecnologias utilizadas pelo sistema em teste.

No entanto, os Objetos Falsificados são capazes de fornecer dados gerados dinamicamente, enquanto os Objetos Emulados não são apropriados para isso. Por isso, os Objetos Falsificados são mais interessantes para testes que precisam de uma grande quantidade de dados, tais como testes de sanidade.

Outra situação que requer geração de dados dinâmica ocorre quando o teste de correção é feito através da comparação dos resultados de algoritmos similares (Seção 6.4.9). Essa abordagem é especialmente útil quando está sendo feito uma otimização: os resultados gerados por um algoritmo que sabemos que está correto são utilizados como valores esperados do algoritmo que está sendo testado.

Intenção: Fornecer uma implementação simplificada e isolada de uma dependência da funcionalidade em teste (objeto colaborador) para que um cenário de teste se torne viável de ser realizado.

Motivação: Alguns cenários de testes são difíceis de serem criados ou executados, principalmente os que dependem de regras de negócio complexas, serviços externos ao sistema em teste etc.

Solução: A ideia é fornecer uma implementação auxiliar e exclusiva para os testes de uma funcionalidade do sistema, de modo que facilite a realização dos cenários de testes que utilizam indiretamente essa funcionalidade. A proposta é a mesma do Antipadrão Gancho para os Testes (*Test Hook*, Seção 6.5.1), mas a implementação deve ser feita de modo elegante, ou seja, sem poluir e aumentar a complexidade do sistema em teste. O código auxiliar deve ser visível apenas dentro do escopo dos testes automatizados.

Consequências: Os cenários de testes difíceis de serem realizados de forma isolada se tornam simples como quaisquer outros.

Implementação: Os Objetos Falsificados devem possuir a mesma interface do objeto colaborador a ser substituído, mas com uma implementação simplificada do comportamento esperado. A implementação pode ser desde uma versão limitada de um algoritmo até mesmo um conjunto de informações *hard-coded* que são simplesmente retornadas. Para a simulação de erros, a implementação pode simplesmente lançar a exceção adequada.

Exemplo - Python/UnitTest: A classe `Compra`, citada no exemplo do padrão Injeção de Dependência (*Dependency Injection*, Seção 6.4.1) possui um exemplo de Objeto Falsificado de uma implementação com dados estáticos (Figura 6.12), o que é bem simples de implementar.

Padrões Relacionados: Esse padrão é a solução elegante do Antipadrão Gancho para os Testes (*Test Hook*, Seção 6.5.1). Também pode-se utilizar Objetos Falsificados para implementar o padrão Teste por Comparação de Algoritmos (Seção 6.4.9). Já o padrão Objeto Emulado (*Mock Object*,

Seção 6.4.6) propõe um outra solução para resolver parte dos problemas que esse padrão também se propõe a resolver. Por fim, como todo Objeto Dublê, o padrão Injeção de Dependência (*Dependency Injection*, Seção 6.4.1) é fundamental.

6.4.6 Objeto Emulado (*Mock Object*)

Tipo: Testabilidade

Quando utilizar: Objetos Emulados também atuam como Objetos Stub, fornecendo dados para o objeto em teste através dos objetos colaboradores. Por isso, eles também podem ser utilizados rotineiramente durante a criação dos cenários de teste. Quando a criação e configuração dos objetos colaboradores é uma tarefa complexa, então deve-se utilizar preferencialmente esse padrão, que facilita essas tarefas. Esse padrão também possui similaridades com o Objeto Espião. Ambos armazenam informações do que foi executado, o que permite que sejam feitas verificações no comportamente interno da funcionalidade em teste.

Intenção: Possibilitar e facilitar a criação de testes para um objeto de forma isolada. Esse padrão também permite verificar as chamadas indiretas da funcionalidade em teste.

Motivação: Testar um código não trivial de maneira isolada é difícil. Além disso, criar e configurar objetos Objetos Stub e Espião pode ser uma tarefa complexa.

Solução: O Objeto Emulado cria uma implementação vazia do objeto colaborador e permite que o comportamento de cada método do objeto possa ser descrito de forma dinâmica.

Consequências: As funcionalidades são testadas isoladamente. Além disso, os Objetos Emulados são muito rápidos, o que melhora a performance dos testes.

Implementação: Criar um Objeto Emulado não é uma tarefa trivial, pois ele é feito de maneira dinâmica por meio de reflexão. Por isso, só é viável sua utilização se existir uma biblioteca de Objetos Emulados para a linguagem do sistema em teste.

Padrões Relacionados: Esse padrão atua como o Objeto Stub (*Test Stub*, Seção 6.4.4), fornecendo dados para o objeto em teste. Também permite verificar chamdas indiretas, como o Objeto Espião (*Test Spy*, Seção 6.4.7). Os Objetos Emulados também precisam ser injetados no objeto em teste, por isso o padrão Injeção de Dependência (*Dependency Injection*, Seção 6.4.1) também é importante.

Usos Conhecidos: Essa solução foi identificada no ano 2000 [93] e desde então tem sido muito estudada [138, 80].

6.4.7 Objeto Espião (*Test Spy*)

Tipo: Testabilidade

Quando utilizar: Quando o que se está querendo verificar é algum comportamento interno da funcionalidade em teste, que não se reflete diretamente nos resultados obtidos. Em outras palavras, o efeito colateral produzido pela funcionalidade em teste não pode ser verificado através de um valor de retorno ou de uma exceção lançada. Exemplos típicos são testes de classes abstratas (especialmente comum em APIs) e de sistemas de registros (*log*).

Intenção: Permitir que um teste consiga verificar se uma chamada indireta de uma funcionalidade está sendo executada corretamente.

Motivação: É importante verificar a correção de chamadas indiretas. Elas podem conter não apenas detalhes fundamentais para o funcionamento do sistema, como também podem interferir na correção do comportamento explícito de uma funcionalidade. Contudo, não é possível verificar esse tipo de funcionalidade do modo convencional, através dos efeitos colaterais diretos causados.

Solução: Criar um objeto que coleta informações das chamadas indiretas da funcionalidade em teste para que possam ser utilizadas posteriormente para verificação.

Consequências: Com objetos espiões se torna possível verificar a correção das saídas indiretas de dados de uma funcionalidade, incluindo chamadas de métodos abstratos.

Implementação: As chamadas indiretas podem ser do próprio objeto em teste ou de algum objeto colaborador. Para o primeiro caso, deve-se herdar a classe em teste acrescentando um sistema de registro dos métodos executados. Já para o segundo caso, deve-se fazer o mesmo mas com o objeto colaborador, com o trabalho adicional de que o objeto espião deverá ser injetado no objeto em teste. Contudo, existem bibliotecas que facilitam o trabalho de gerar objetos espiões, além de fornecerem funções padronizadas para verificação dos dados coletados.

Exemplo - Python/Unittest: A Figura 6.16 mostra uma classe abstrata com um algoritmo de sincronização que percorre duas listas simultaneamente enquanto vai comparando seus itens (`SincronizadorDeListas`). Os tipos que herdarem dessa classe devem definir qual o comportamento desejado quando os ítens forem comuns a ambas as listas ou exclusivo em alguma delas.

É importante notar que não é possível testar o comportamento dessa classe do modo convencional, comparando as saídas diretas de dados (valores retornados ou exceções lançadas) com valores esperados. Uma solução é criar uma classe espiã que adicionará um comportamento para capturar informações das chamadas indiretas de modo que essas informações possam ser verificadas posteriormente. A Figura 6.17 contém uma implementação de classe espiã e realiza os testes usando-a como base.

Exemplo - Python/Unittest/Python-Mockito: A principal vantagem de utilizar ferramentas de objetos espiões é que o trabalho de criar um novo objeto com um sistema de registro é dispensado. Outra vantagem é que os testes ficam padronizados e, consequentemente, mais fáceis de serem interpretados. A Figura 6.18 mostra um outro exemplo de teste para o código da Figura 6.16, mas dessa vez utilizando a ferramenta Python-Mockito para gerar os objetos espiões.

Padrões Relacionados: O padrão Injeção de Dependência (*Dependency Injection*, Seção 6.4.1) é fundamental para que o Objeto Espião seja inserido no objeto em teste. O Objeto Emulado (*Mock Object*, Seção 6.4.6) também armazena informações da sua execução, no entanto, ele não é capaz de fornecer dados dinâmicos para o sistema em teste.

```python
class SincronizadorDeListas(object):

    def __init__(self, lista1, lista2):
        self.lista1 = lista1
        self.lista2 = lista2
        self.lista1.sort()
        self.lista2.sort()

    def executa(self):
        self.__executa_recusivamente(0, 0)

    def __executa_recusivamente(self, indice1, indice2):
        sem_mais_elementos_na_lista1 = indice1 >= len(self.lista1)
        sem_mais_elementos_na_lista2 = indice2 >= len(self.lista2)

        if sem_mais_elementos_na_lista1 and sem_mais_elementos_na_lista2:
            return

        if sem_mais_elementos_na_lista2:
            elemento1 = self.lista1[indice1]
            self.processa_elemento_exclusivo_lista1(elemento1)
            return self.__executa_recusivamente(indice1 + 1, indice2)

        if sem_mais_elementos_na_lista1:
            elemento2 = self.lista2[indice2]
            self.processa_elemento_exclusivo_lista2(elemento2)
            return self.__executa_recusivamente(indice1, indice2 + 1)

        elemento1 = self.lista1[indice1]
        elemento2 = self.lista2[indice2]

        if elemento1 == elemento2:
            self.processa_elementos_iguais(elemento1, elemento2)
            return self.__executa_recusivamente(indice1 + 1, indice2 + 1)
        elif elemento1 > elemento2:
            self.processa_elemento_exclusivo_lista2(elemento2)
            return self.__executa_recusivamente(indice1, indice2 + 1)
        else: # elemento1 < elemento2
            self.processa_elemento_exclusivo_lista1(elemento1)
            return self.__executa_recusivamente(indice1 + 1, indice2)

    # métodos abstratos:
    def processa_elementos_iguais(self, elemento1, elemento2): pass
    def processa_elemento_exclusivo_lista1(self, elemento1): pass
    def processa_elemento_exclusivo_lista2(self, elemento2): pass
```

Figura 6.16: Uma classe python com métodos abstratos.

```python
import unittest

from sincronizador import SincronizadorDeListas

# Herda a classe em teste e adiciona comportamento para espioná-la.
class SincronizadorDeListasEspiao(SincronizadorDeListas):

    def __init__(self, lista1, lista2):
        super(SincronizadorDeListasEspiao, self).__init__(lista1, lista2)
        self.processa_elementos_iguais_contador = 0
        self.processa_elemento_exclusivo_lista1_contador = 0
        self.processa_elemento_exclusivo_lista2_contador = 0

    def processa_elementos_iguais(self, element1, element2):
        self.processa_elementos_iguais_contador += 1

    def processa_elemento_exclusivo_lista1(self, element1):
        self.processa_elemento_exclusivo_lista1_contador += 1

    def processa_elemento_exclusivo_lista2(self, element2):
        self.processa_elemento_exclusivo_lista2_contador += 1


class SincronizadorDeListasEspiaoTests(unittest.TestCase):

    def verificar_chamadas(self, espiao, a, b, c):
        self.assertEquals(a, espiao.processa_elementos_iguais_contador)
        self.assertEquals(b, espiao.processa_elemento_exclusivo_lista1_contador)
        self.assertEquals(c, espiao.processa_elemento_exclusivo_lista2_contador)

    def test_nao_deve_executar_nada_se_recebe_duas_listas_vazias(self):
        espiao = SincronizadorDeListasEspiao([], [])
        espiao.executa()
        self.verificar_chamadas(espiao, 0, 0, 0)

    def test_deve_processar_elemento_exclusivo_lista1(self):
        espiao = SincronizadorDeListasEspiao([1], [])
        espiao.executa()
        self.verificar_chamadas(espiao, 0, 1, 0)

    def test_deve_processar_elemento_exclusivo_lista2(self):
        espiao = SincronizadorDeListasEspiao([], [1])
        espiao.executa()
        self.verificar_chamadas(espiao, 0, 0, 1)

    def test_deve_processar_elemento_comum_em_ambas_as_listas(self):
        espiao = SincronizadorDeListasEspiao([1], [1])
        espiao.executa()
        self.verificar_chamadas(espiao, 1, 0, 0)

    def test_ambas_as_listas_com_elementos_distintos(self):
        espiao = SincronizadorDeListasEspiao([1], [2])
        espiao.executa()
        self.verificar_chamadas(espiao, 0, 1, 1)
```

Figura 6.17: Exemplo de teste com Objeto Espião.

```python
import unittest
from mockito import *

from sincronizador import SincronizadorDeListas

class SincronizadorDeListasTests(unittest.TestCase):

    def test_nao_deve_executar_nada_se_recebe_duas_listas_vazias(self):
        espiao = spy(SincronizadorDeListas([], []))
        espiao.executa()

        verify(espiao).executa()
        verify(espiao).__executa(0, 0)
        verifyNoMoreInteractions(espiao) # Não executou nenhum outro método

    def test_deve_processar_elemento_exclusivo_lista1(self):
        espiao = spy(SincronizadorDeListas([1], []))
        espiao.executa()

        # Não executou o método 'processa_elementos_iguais' com parâmetro '1'
        verify(espiao, times=0).processa_elementos_iguais(1)
        # Executou uma vez o método 'processa_elemento_exclusivo_lista1' com
            parâmetro '1'
        verify(espiao, times=1).processa_elemento_exclusivo_lista1(1)
        verify(espiao, times=0).processa_elemento_exclusivo_lista2(1)

    def test_deve_processar_elemento_exclusivo_lista2(self):
        espiao = spy(SincronizadorDeListas([], [1]))
        espiao.executa()

        verify(espiao, times=0).processa_elementos_iguais(1)
        verify(espiao, times=0).processa_elemento_exclusivo_lista1(1)
        verify(espiao, times=1).processa_elemento_exclusivo_lista2(1)

    def test_deve_processar_elemento_comum_em_ambas_as_listas(self):
        espiao = spy(SincronizadorDeListas([1], [1]))
        espiao.executa()

        verify(espiao, times=1).processa_elementos_iguais(1)
        verify(espiao, times=0).processa_elemento_exclusivo_lista1(1)
        verify(espiao, times=0).processa_elemento_exclusivo_lista2(1)

    def test_ambas_as_listas_com_elementos_distintos(self):
        espiao = spy(SincronizadorDeListas([1], [2]))
        espiao.executa()

        verify(espiao, times=0).processa_elementos_iguais(1)
        verify(espiao, times=0).processa_elementos_iguais(2)
        verify(espiao, times=1).processa_elemento_exclusivo_lista1(1)
        verify(espiao, times=0).processa_elemento_exclusivo_lista1(2)
        verify(espiao, times=0).processa_elemento_exclusivo_lista2(1)
        verify(espiao, times=1).processa_elemento_exclusivo_lista2(2)
```

Figura 6.18: Exemplo de teste de Objeto Espião com Python-Mockito.

Usos Conhecidos: As ferramenta Mockito para Java e Python-Mockito para Python são ferramentas que disponibilizam objetos espiões.

6.4.8 Objeto Protótipo

Tipo: Organizacional, Robustez e Testabilidade

Quando utilizar: Quando uma funcionalidade processa trechos arbitrários de código-fonte, como ocorre quando é utilizado Reflexão ou Programação Orientada a Aspectos.

Intenção: Agrupar em um ou poucos objetos de uso exclusivo dos testes, diferentes protótipos de implementação que sejam pertinentes para realização dos cenários de teste de uma funcionalidade que trabalha com informações do código-fonte.

Motivação: Pode ser necessário o uso de muitos objetos distintos e não relacionados para testar funcionalidades que processam código-fonte, principalmente quando elas são muito abrangentes. Por exemplo, um aspecto pode ser definido para adicionar código a todas as classes do sistema. Isso pode causar diferentes tipos de problema para os testes, como *set ups* complexos, testes com lógica condicional, entre outros problemas organizacionais.

Além disso, testes que envolvem muitos módulos de um sistema não são robustos. Alterações de código-fonte em diversos pontos do sistema podem quebrar os testes, mesmo que a funcionalidade que está sendo testada não tenha sido alterada.

Ainda há os casos em que os testes não podem ser realizados devido à indisponibilidade de objetos. Por exemplo, é comum que as APIs forneçam objetos abstratos e incompletos, contendo apenas um esqueleto de implementação (classes abstratas e *Template Methods*). Para esses casos, pode ser utilizado Protótipos de Objetos em Teste ou outros Objetos Dublês.

Solução: Os testes podem criar um ou mais objetos, visíveis apenas no escopo dos testes (Objetos Dublês), que contêm protótipos de diferentes tipos de implementação, de modo que diferentes cenários de testes possam ser realizados.

Diferentemente dos Objetos Falsificados (Seção 6.2), o Objeto Protótipo não precisa respeitar uma API definida rigidamente pelo sistema, a não ser que seja pertinente ao teste. De maneira geral, ele apenas deve fornecer recursos que imitem as características de código-fonte dos objetos do sistema. Ainda, o protótipo não fornece dados ao sistema, o próprio código-fonte compõe as informações necessárias para realização dos testes.

Outra grande diferença entre o Objeto Protótipo e outros Objetos Dublês é o processo de instalação do objeto no sistema em teste. No caso de orientação a aspectos, uma forma de instalação se dá através da compilação do código dos testes utilizando o compilador de aspectos. Isso adicionará o comportamento a ser testado no protótipo. Já para reflexão, a instalação ainda pode ser feita do modo convencional, através de injeção de dependência.

É importante ressaltar que o que deve ser testado são funcionalidades do sistema, nunca os Objetos Dublês. Sendo assim, não é a implementação do protótipo que deve ser utilizada para os testes, mas sim as informações do seu código-fonte ou trechos de código que foram embutidos.

Consequências: Todos os tipos de código-fonte necessários para os testes ficam encapsulados em um ou poucos objetos no escopo dos testes. Isso deixa os testes mais organizados, simples e robustos.

Implementação: No caso de orientação a aspectos, o protótipo deve ser um objeto simples, mas que seja identificado pelos Pontos de Atuação (*Pointcuts*, Seção 6.3.4) para que os testes consigam exercitar o código dos Adendos (*Advices*) que serão adicionados ao protótipo. Entretanto, também é interessante criar protótipos que não sejam identificados pelos Pontos de Atuação, justamente para testar que eles realmente não são encontrados pelos aspectos. Para reflexão, a implementação é similar, mas o objeto deve satisfazer alguma API para que as funcionalidades reflexivas reconheçam o objeto. Da mesma maneira, protótipos incompatíveis com as funcionalidades também

são recomendados para realização dos testes de casos de erros. Valor notar que, dependendo da funcionalidade, os nomes das classes e métodos podem descrever o que eles representam para os testes.

Exemplo - Java: A Figura 6.19 mostra um Objeto Protótipo que pode ser utilizado para testes de diferentes aspectos. Se um Ponto de Atuação representa apenas os métodos protegidos (*protected*), então o código do Adendo deve ser inserido apenas no método `umMetodoProtegidoParaTeste`. Se o comportamento desse método corresponder ao código do Adendo, então a expressão de mapeamento do Ponto de Atuação está encontrando o trecho de código desejado. No entanto, também é necessário verificar que o Adendo não é inserido nos outros métodos. Ainda, os testes do próprio Adendo pode ser verificado através do comportamento do método `umMetodoProtegidoParaTeste`.

```
1
2  public class PrototipoDeObjetoEmTeste {
3    public void umMetodoPublicoParaTeste() {}
4
5    protected void umMetodoProtegidoParaTeste() {}
6
7    private void umMetodoPrivadoParaTeste() {}
8
9    public void umMetodoQueLancaExcecao() throws Exception {
10     throw new Exception();
11   }
12   // ...
13 }
```

Figura 6.19: Objeto Protótipo.

Exemplo - Python/Unittest/Django-Dynamic-Fixture: A biblioteca Django-Dynamic-Fixture utiliza reflexão para identificar os tipos de dados esperados para cada variável de um objeto de dados do arcabouço Django. Depois que os tipos são identificados, a biblioteca preenche as variáveis com informações apropriadas e do tipo esperado. A Figura 6.20 mostra alguns testes dessa ferramenta utilizando esse padrão.

Padrões Relacionados: O padrão Objeto Humilde 6.4.2 deve ser sempre utilizado como pré-requisito, justamente para simplificar os casos de testes e, consequentemente, o Objeto Protótipo.

Ainda, como o objeto criado é um Objeto Dublê (Seção 6.2), existem similaridades entre ele e Objetos Stub, Mock, Falsificado e Tolo. Por exemplo, a implementação é visível apenas ao escopo dos testes, assim como a implementação deve ser simples, fácil e rápida.

Usos Conhecidos: A ferramenta Util4Selenium, que usa aspectos (AspectJ com Java) para gerar fotografias das interfaces Web, usa esse padrão para testar a funcionalidade. Já a biblioteca de testes Python-QAssertions e a *engine* de jogos de cartas Card Game Engine também utilizam essa solução para testar os trechos de código que envolvem reflexão.

```python
from django.test import TestCase
from django.db import models

# O método new preenche as variáveis do objeto com valores válidos
from django_dynamic_fixture import new

class MetodoNewPreencheInstanciaComDadosTest(TestCase):

    def test_preenche_integer_fields_com_inteiros(self):
        # Objeto Protótipo
        class UmModelo(models.Model):
            integer_field = models.IntegerField()
        instancia = new(UmModelo)
        # O método new identificou que a variável esperava receber um número
            inteiro
        self.assertTrue(isinstance(instancia.integer_field, int))

    def test_preenche_char_fields_com_strings(self):
        # Objeto Protótipo
        class UmModelo(models.Model):
            char_field = models.CharField(max_length=10)
        instancia = new(UmModelo)
        # O método new identificou que a variável esperava receber uma string
        self.assertTrue(isinstance(instancia.char_field, str))
```

Figura 6.20: Exemplo em Python de testes da biblioteca Django-Dynamic-Fixture utilizando o padrão Objeto Protótipo.

6.4.9 Teste por Comparação de Algoritmos

Tipo: Qualidade

Quando utilizar: Quando a funcionalidade em teste é um algoritmo de alta complexidade com muitas combinações de dados de entrada e saída, enquanto existem algoritmos mais simples que resolvem o mesmo problema, mas que são inviáveis de serem utilizados em ambiente de produção. Por exemplo, quando alguns algoritmos triviais para problemas complexos são extremamente lentos. Esse padrão é especialmente útil para verificar algoritmos de cálculo de propriedades matemáticas, análise combinatória, programação musical, computação gráfica etc.

Intenção: Verificar a correção de um algoritmo otimizado com base nos valores gerados por um outro algoritmo reconhecidamente correto.

Motivação: Muitos problemas computacionais são difíceis de serem resolvidos, ainda mais se o desempenho e a flexibilidade da solução for imprescindível. Além do mais, algoritmos complexos podem conter muitas partições de domínios, dificultando ou mesmo inviabilizando a criação de uma bateria de testes automatizados que traga segurança quanto à correção da implementação.

Solução: Testar o algoritmo com uma grande quantidade de dados de entrada com o intuito de encontrar algum cenário de teste que ainda não foi verificado. Para isso, deve ser implementado um algoritmo, que seja correto e fácil de implementar, para gerar os resultados esperados de um caso de teste que serão posteriormente utilizados para comparar com os resultados obtidos pelo algoritmo em teste.

Consequências: O algoritmo é testado com uma grande quantidade de dados, o que traz mais segurança quanto à sua correção, já que aumenta as chances de verificar alguma partição de domínio que o desenvolvedor pode ter esquecido de testar.

Implementação: O primeiro passo é implementar um algoritmo reconhecidamente correto dentro do escopo dos testes. A próxima etapa é programar para que tanto o algoritmo de controle, quanto o algoritmo em teste sejam executados diversas vezes com os mesmos dados de entrada, para que então seja comparado os resultados obtidos pelos dois.

Exemplo: Na Figura 6.21 temos um algoritmo eficiente para cálculo do Máximo Dividor Comum (M.D.C.) de dois números inteiros. Os primeiros cenários de testes a serem realizados devem ser os casos mais simples, tais como combinações de números pares, ímpares, números primos e números primos entre si. Caso haja insegurança quanto à correção do algoritmo, podemos implementar um teste de comparação de algoritmos para tentar encontrar um cenário de teste que ainda não foi pensado. Para esse caso, podemos utilizar um algoritmo lento, mas reconhecidamente correto, para servir de modelo para os testes (Figura 6.22).

```python
class MathHelper(object):
    # Algoritmo de Euclides: mdc(a, b) = mdc(b, r) onde r: a = q * b + r
    def mdc(self, a, b):
        valor = max([a, b])
        divisor = min([a, b])
        resto = valor % divisor
        while resto != 0:
            valor = divisor
            divisor = resto
            resto = valor % divisor
        return divisor
```

Figura 6.21: Algoritmo eficiente para cálculo do M.D.C. entre dois números inteiros.

```python
import random
import unittest

class MathHelperFake(object):
    # Algoritmo ingenuamente lento, mas simples de implementar.
    def mdc(self, a, b):
        maior_divisor_possivel = min((a, b))
        divisor_comum = 1
        for i in range(1, maior_divisor_possivel+1):
            if a % i == 0 and b % i == 0:
                divisor_comum = i
        return divisor_comum

class MathHelperTest(unittest.TestCase):
    def setUp(self):
        self.fake = MathHelperFake()
        self.math_helper = MathHelper()

    def test_aleatorio_por_comparacao_de_algoritmo_do_calculo_de_mdc(self):
        for i in range(1, 10):
            a = random.randint(1, 1000000)
            b = random.randint(1, 1000000)
            expected_value = self.fake.mdc(a, b)
            value = self.math_helper.mdc(a, b)
            self.assertEquals(expected_value, value)
```

Figura 6.22: Exemplo de Teste por Comparação de Algoritmos.

6.4.10 Teste por Probabilidade

Tipo: Robustez e Qualidade

Quando utilizar: Quando o que está sendo testado possui comportamento aleatório. Esta situação é típica em jogos e algoritmos de segurança, tais como geração de senhas aleatórias e algoritmos de criptografia que utilizam, em alguma etapa, números pseudo-aleatórios.

Intenção: Definir o resultado final de um teste baseado nos resultado de diversas execuções de um teste intermitente.

Motivação: Funcionalidades que produzem resultados aleatórios são difíceis de serem testadas, pois os resultados esperados não são previsíveis. Tentar prever os resultados leva ao indício de antipadrão Testes Intermitentes (Seção 5.2).

Solução: Executar um mesmo caso de teste diversas vezes e definir se o teste é aceito de acordo com a porcentagem de sucesso.

Consequências: É definido, de forma viável, o resultado de um caso de teste baseado na probabilidade de sucesso de uma verificação. Os testes continuarão sendo teoricamente intermitentes, mas, na prática, é robusto como outro qualquer.

Implementação: Antes de utilizar este padrão, é importante refatorar o sistema ou tentar utilizar o padrão Objeto Humilde (Seção 6.4.2) para isolar o comportamento aleatório de uma funcionalidade. Isso poderá facilitar a escrita dos Testes por Probabilidade ou até mesmo evitá-los.

Dado que é necessário utilizar esse padrão, então é necessário executar o teste uma quantidade determinada de vezes (um laço simples) e armazenar os resultados de todas as execuções (uma lista com os resultados), para no final, calcular a razão de sucesso sobre fracasso e comparar com a probabilidade desejada (uma conta e comparação simples). Idealmente, esse algoritmo deve estar integrado ao arcabouço de teste para seguir as convenções e enriquecer o relatório da bateria dos testes, embora seja possível implementá-lo de maneira independente, como é mostrado na Figura 6.23.

Exemplo - Java/JUnit/TestNG: A implementação não é complexa, mas pode tornar o código dos testes obscuro, outro indício de antipadrão. Por isso, é importante abstrair o conceito do Teste por Probabilidade de forma que fique transparente para os testes que o utilizam. Contudo, existem arcabouços que já fornecem essa implementação e facilitam a utilização deste padrão. A Figura 6.24 mostra um exemplo de teste de uma funcionalidade típica em jogos de cartas, embaralhar uma pilha de cartas. O exemplo é escrito em Java e utiliza o arcabouço TestNG, que permite criar facilmente testes por probabilidade através de metadados.

Padrões Relacionados: O padrão Objeto Humilde (Seção 6.4.2) é útil para separar o que for possível da lógica aleatória do resto da lógica de uma funcionalidade. Também, Testes por Probabilidade podem ser usados em conjunto com Testes de Sanidade (Seção 3.4.3) para possibilitar fazer verificações mais rígidas, assim, o padrão também pode ser utilizado com o intuito de garantir a qualidade do sistema.

Usos Conhecidos: O arcabouço de testes TestNG fornece uma maneira simples de configurar testes baseados na probabilidade de sucesso.

```java
public class TestePorProbabilidadeImplTests {

  public void testePorProbabilidade() {
    // Implementação do teste
  }

  // Teste será executado 20 vezes.
  // Se 8 ou mais vezes (>= 80%) passar, resultado é encarado como sucesso.
  // Caso contrário, é encarado como falha.
  @Test
  public void testePorProbabilidadeDecorator() {
    int QTDE_EXECUCOES = 20;
    int PORCENTAGEM_DE_SUCESSO_ESPERADA = 80;

    int quantidadeDeSucessos = 0;
    for(int i = 0; i < 20; i++) {
      try {
        testePorProbabilidade();
        quantidadeDeSucessos++;
      } catch(Exception e) {
      }
    }
    int porcentagemDeSucesso = 100 * quantidadeDeSucessos / QTDE_EXECUCOES;
    if(porcentagemDeSucesso < PORCENTAGEM_DE_SUCESSO_ESPERADA)
      throw new RuntimeException(
          "Falhou mais do que o esperado: " +
          porcentagemDeSucesso + "%");
  }
}
```

Figura 6.23: Exemplo de teste que verifica a correção de um teste pela probabilidade.

```java
//Referências do TestNG
import org.testng.annotations.Test;
//Referências do JUnit + Hamcrest
import static org.junit.Assert.*;
// Custom Matcher: isNotSorted
import static QAMatchers.*;

public class PilhaDeCartasTests {

  // Teste será executado 20 vezes.
  // Se 8 ou mais vezes (>= 80%) passar, resultado é encarado como sucesso.
  // Caso contrário, é encarado como falha.
  @Test(invocationCount = 20, successPercentage = 80)
  public void embaralharDeveMisturarAsCartasEmUmaOrdemAleatoria() {
    PilhaDeCartas pilhaDeCartas = new PilhaDeCartas();
    pilhaDeCartas.adiciona(new Carta(1, 1));
    pilhaDeCartas.adiciona(new Carta(2, 2));
    pilhaDeCartas.adiciona(new Carta(3, 3));
    pilhaDeCartas.embaralha();
    // Verifica que NÃO está ordenado.
    // Intermitente: A função embaralha pode deixar o baralho organizado
    assertThat(pilhaDeCartas.getCartas(), isNotSorted(Carta.comparadorPorNaipeValor
        ()));
  }
}
```

Figura 6.24: Exemplo de teste que verifica a correção de um teste pela probabilidade.

6.4.11 Verificar Inversibilidade

Tipo: Qualidade

Quando utilizar: Ao testar duas funções do sistema que precisam ser exatamente uma inversa da outra. Tipicamente em testes de funções matemáticas bijetoras, funcionalidades de importação e exportação de dados, voltar e refazer ações, além de efeitos de imagem e som que são inversíveis.

Intenção: Verificar a existência de erros de incompatibilidade entre uma função bijetora e sua inversa.

Motivação: Duas funcionalidades podem ser individualmente corretas, mas ao trabalharem como inversas são incompatíveis. Vários erros podem ser cometidos, especialmente em pequenos detalhes que passam despercebidos, tais como caracteres invisíveis em strings, diferenças de arredondamento em pontos flutuantes e até mesmo a ordem de elementos idênticos em listas ordenadas, que podem ser diferentes quando são utilizados algoritmos de ordenação instáveis.

Solução: Criar cenários de testes que comparem os resultados produzidos por duas funções que são inversas entre si, sendo que os dados de entrada de uma das funções são os dados de saída da outra.

Consequências: É verificado se as implementações de duas funções teoricamente inversas entre si podem ser utilizadas na prática, sem incompatibilidade.

Implementação: A ideia do teste é executar as duas funcionalidades supostamente inversas entre si fef^{-1}, sendo que uma processará um dado qualquer x e a outra receberá como dados de entrada os dados de saída produzida pela primeira, $f(x)$. Assim, temos: $y = f^{-1}(f(x))$. Se $x = y$, então o teste é tido como sucesso, caso contrário, uma falha.

Exemplo - Java/JUnit/Hamcrest: A Figura 6.25 mostra um algoritmo simples de criptografia e descriptografia, que são duas funcionalidades necessariamente inversas. Caso contrário, o usuário poderá perder dados criptografados importantes porque não conseguirá recuperá-los. Note que, propositalmente, um detalhe da implementação do método de descriptografia foi comentado para enfatizar o caso de teste de inversibilidade (Figura 6.26).

Exemplo - Python/UnitTest/QAssertions: A implementação do teste de inversibilidade é simples de ser feito e não suja o código-fonte. No entanto, é possível criar métodos de asserção que abstraem o objetivo do teste, o que é útil para enfatizar o que está sendo testado e também para lembrar e incentivar a realização deste tipo de teste. A Figura 6.27 mostra um exemplo com a ferramenta Python-QAssertions. O método de asserção recebe os dois métodos que teoricamente são funções inversas entre si e os argumentos que serão passados para as funções em si. A ferramenta executa os dois métodos apropriadamente e faz as comparações apropriadas.

Padrões Relacionados: Se a quantidade de argumentos que podem ser passados para as funções é infinita, então é impossível provar com testes automatizados, que duas funcionalidades são inversas entre si. Por isso, é interessante que sejam feitas verificações com diversos argumentos. Para isso, pode-se utilizar os padrões Teste de Sanidade (Seção 3.4.3) e Testes Aleatórios (Seção 3.4.1) para gerar argumentos para verificação de inversibilidade.

Usos Conhecidos: A ferramenta Python-QAssertions implementa um método de asserção (`assertFunctionsAreInversible`) que realiza este tipo de verificação.

```java
import java.util.Random;

public class Criptografador {
  private static final int _RANDOM_NUMBER = 10;

  public String criptografar(String senha, String texto) {
    Random random = new Random(senha.hashCode());
    byte[] bytes = texto.getBytes();
    for(int i = 0; i < bytes.length; i++) {
      bytes[i] = (byte) (bytes[i] + random.nextInt(_RANDOM_NUMBER));
    }
    return "!!" + new String(bytes) + "!!";
  }

  public String descriptografar(String senha, String texto) {
    // Precisa descomentar a linha abaixo para que
    // as funções criptografar e decriptografar sejam inversas entre si.
    // texto = texto.replaceFirst("^!!", "").replaceFirst("!!$", "");
    Random random = new Random(senha.hashCode());
    byte[] bytes = texto.getBytes();
    for(int i = 0; i < bytes.length; i++) {
      bytes[i] = (byte) (bytes[i] - random.nextInt(_RANDOM_NUMBER));
    }
    return new String(bytes);
  }
}
```

Figura 6.25: Algoritmo ingênuo de criptografar and descriptografar textos.

```java
//Referências do JUnit + Hamcrest
import org.junit.*;
import static org.junit.Assert.*;
import static org.hamcrest.Matchers.*;

public class CriptografiaSimplesTests {
  Criptografador c;

  @Before
  public void inicializaVariaveis() {
    c = new CriptografadorSimples();
  }

  @Test // Sucesso.
  public void testeCriptografar() {
    String texto = "abcdefghijk";
    String criptografado = c.criptografar("senha123", texto);
    assertThat(criptografado, equalTo("!!jddhfgkplmk!!"));
  }

  @Test // Sucesso.
  // Dependendo do requisito do cliente, pode ser um falso positivo.
  public void testeDescriptografar() {
    String criptografado = "jddhfgkplmk";
    String decriptografado = c.decriptografar("senha123", criptografado);
    assertThat("abcdefghijk", equalTo(decriptografado));
  }

  @Test // Falha.
  // As funções criptografar e descriptografar não são inversas entre si.
  public void criptografarInversaDeDescriptografar() {
    String texto = "abc";
    String criptografado = c.criptografar("senha123", texto);
    String descriptografado = c.descriptografar("senha123", criptografado);
    assertThat(texto, equalTo(descriptografado));
  }
}
```

Figura 6.26: Teste de inversibilidade dos algoritmos de criptografia e descriptografia.

```python
# Referência do UnitTest
import unittest
# Referência do Python-QAssertions
import qassertions as qa

class InversibilidadeTests(unittest.TestCase):

    def testFuncaoCriptInversaDeDecript(self):
        c = Criptador("senha123")
        # qa.assertFunctionsAreInversible(método1, método2, argumento)
        qa.assertFunctionsAreInversible(c.cript, c.decript, "texto")
```

Figura 6.27: Asserção de Inversibilidade da ferramenta Python-QAssertions.

6.4.12 Verificar Valores Limites

Tipo: Qualidade

Quando utilizar: Em testes de algoritmos que trabalham com intervalos de valores de dados de entrada, saída ou, até mesmo, de variáveis locais. O uso típico é em algoritmos com cálculos puramente matemáticos, laços complexos, vetores e matrizes, conjuntos de dados ordenaveis, pontos flutuantes etc.

Intenção: Verificar erros de programação causados pelos valores limites dos intervalos de dados que o algoritmo trabalha.

Motivação: Estudos mostram que erros em valores limites são frequentes [95]. Erros de validação de dados, falhas de segmentação, laços infinitos, divisão por zero etc. Além disso, erros de valores limites podem ser acarretados devido à interpretação incorreta de requisitos definidos de forma ambígua ou não clara pelos clientes.

Solução: Testar as funcionalidades de acordo com os dados extremos de entrada e saída.

Consequências: É feito a prevenção contra erros causados por valores extremos.

Implementação: A implementação dos testes de valores limites não necessitam de soluções especiais, apenas os valores de entrada e saída são escolhidos cuidadosamente para forçar cálculos e comparações específicos.

Exemplo - C/CUnit: A Figura 6.28 mostra uma funcionalidade escrita em C que faz a multiplicação de matrizes de inteiros. A implementação, apesar de curta, possui muitos detalhes sutis, o que aumenta as chances do programador cometer erros conceituais ou por distração, tais como trocas de variáveis e erros de precedência de operadores. Além do mais, essa é uma função que aceita uma combinação infinita de dados de entrada, o que torna impossível provar sua correção através de testes automatizados, embora é possível criar uma boa bateria de testes que dá segurança na funcionalidade.

```
1  /*
2     Matrizes de inteiros A, B e C
3     C(m x o) = A(m x n) * B(n x o)
4     Retorna 0 se sucesso, negativo caso contrário
5  */
6  int matrizXmatriz(int **A, int m, int n, int **B, int o, int **C){
7    int i, j, k;
8    if(m < 1 || n < 1 || o < 1) return -1;
9    for(i = 0; i < m; i++) {
10     for(j = 0; j < o; j++) {
11       C[i][j] = 0;
12       for(k = 0; k < n; k++)
13         C[i][j] = C[i][j] + (A[i][k] * B[k][j]);
14     }
15   }
16 }
```

Figura 6.28: Função escrita em C que calcula a multiplicação de matrizes.

Para essa funcionalidade, problemas de valores limites podem ocorrer tanto com os dados de entrada como os de saída. A Figura 6.29 mostra alguns dos testes que podem ser feitos para

verificar esse tipo de erro. O exemplo utiliza a ferramenta CUnit, mas mostra apenas os cenários de teste. No Apêndice B há mais informações sobre a ferramenta, em particular, como executar os testes com o arcabouço.

```c
/* Referências do CUnit, outras referências foram ocultas */
#include <CUnit/CUnit.h>

/* Funções auxiliares: Implementação oculta para simplificar o exemplo */
/* A = B = C|0 ... 0|
            |0 ... 0| ... */
void inicializaMatrizesZeradas(int** A, int m, int n, int** B, int o, int** C) {}

/* A|a b| B|e f| C|0 0|
    |c d|  |g h|  |0 0| */
void inicializaMatrizesDoisPorDoisComValores(int** A, int a, int b, int c, int d,
    int** B, int e, int f, int g, int h, int** C) {}

void verificaMatrizDoisPorDois(A, a, b, c, d) {
  CU_ASSERT_EQUAL(A[0][0], a);
  CU_ASSERT_EQUAL(A[0][1], b);
  CU_ASSERT_EQUAL(A[1][0], c);
  CU_ASSERT_EQUAL(A[1][1], d);
}

/* TESTES */

/* Função se comporta bem com zeros? */
void test_ValoresLimitesDosCalculos(void) {
  int **A, **B, **C;
  inicializaMatrizesDoisPorDois(A, 0, 1, 0, 1, B, 1, 0, 1, 0, C);
  int conseguiuCalcular = matrizXmatriz(A, 2, 2, B, 2, C);
  CU_ASSERT_TRUE(conseguiuCalcular == 0);
  verificaMatrizDoisPorDois(C, 1, 1, 1, 1);
}

/* E com números negativos? */
void test_ValoresLimitesDosCalculos(void) {
  int **A, **B, **C;
  inicializaMatrizesDoisPorDois(A, 1, -1, 1, -1, B, -1, 1, -1, 1, C);
  int conseguiuCalcular = matrizXmatriz(A, 2, 2, B, 2, C);
  CU_ASSERT_TRUE(conseguiuCalcular == 0);
  verificaMatrizDoisPorDois(C, 1, 1, 1, 1);
}

/* Resultados (dados de saída) devem pertencer ao intervalo dos inteiros */
void test_ValoresLimitesDosResultados(void) {
  int **A, **B, **C;
  inicializaMatrizesDoisPorDois(A, INT_MAX, INT_MAX, INT_MAX, INT_MAX, B, 2, 2, 2,
      2, C);
  int conseguiuCalcular = matrizXmatriz(A, 2, 2, B, 2, C);
  CU_ASSERT_TRUE(conseguiuCalcular < 0);
}
```

Figura 6.29: Teste da multiplicação de matrizes usando a biblioteca CUnit.

Exemplo - Scala/JUnit/TestNG: Contudo, esses tipos de erros podem acontecer até com algoritmos mais simples e que utilizam linguagens de programação de mais alto nível. A Figura 6.30 mostra

um exemplo de testes escritos em Scala com JUnit e TestNG para o jogo de carta Poker. As regras do jogo definem uma hierarquia simples de combinações de cartas, mas que programaticamente pode conter erros nos valores limites das regras.

```
// Referências do TestNG e JUnit
import org.testng.annotations._
import org.junit.Assert._
// Imports das classes do sistema foram ocultos

// Exemplos de testes de valores limites para o jogo de cartas Poker
// Obs: Nomes dos métodos de testes definem parte das regras do jogo
class ComparacaoEntreCombinacoesLimitesTests {

  @Test def menorParGanhaDeMaiorCartaMaisAlta() {
    assertTrue(menorPar() > maiorCartaMaisAlta())
    assertTrue(maiorCartaMaisAlta() < menorPar())
  }

  @Test def menorDoisParesGanhaDeMaiorPar() {
    assertTrue(menorDoisPares() > maiorPar())
    assertTrue(maiorPar() < menorDoisPares())
  }

  @Test def menorTrincaGanhaDeDeMaiorDoisPares() {
    assertTrue(menorTrinca() > maiorDoisPares())
    assertTrue(maiorDoisPares() < menorTrinca())
  }

  @Test def menorSequenciaGanhaDeDeMaiorTrinca() {
    assertTrue(menorSequencia() > maiorTrinca())
    assertTrue(maiorTrinca() < menorSequencia())
  }

  @Test def menorTodasMesmoNaipeGanhaDeDeMaiorSequencia() {
    assertTrue(menorTodasMesmoNaipe() > maiorSequencia())
    assertTrue(maiorSequencia() < menorTodasMesmoNaipe())
  }

  @Test def menorFullHouseGanhaDeDeMaiorTodasMesmoNaipe() {
    assertTrue(menorFullHouse() > maiorTodasMesmoNaipe())
    assertTrue(maiorTodasMesmoNaipe() < menorFullHouse())
  }

  @Test def menorQuadraGanhaDeDeMaiorFullHouse() {
    assertTrue(menorQuadra() > maiorFullHouse())
    assertTrue(maiorFullHouse() < menorQuadra())
  }

  @Test def menorSequenciaTodasMesmoNaipeGanhaDeDeMaiorQuadra() {
    assertTrue(menorSequenciaTodasMesmoNaipe() > maiorQuadra())
    assertTrue(maiorQuadra() < menorSequenciaTodasMesmoNaipe())
  }
}
```

Figura 6.30: Teste escrito em Scala dos valores limites das regras do Poker.

Exemplo - Python/UnitTest/QAssertions: Ainda, existem ferramentas que geram testes automatizados para verificar valores limites em regras de validação de dados de entrada. É o caso da

Python-QAssertions, que fornece o método de verificaçao `assertValidation`. Esse método recebe como argumentos o próprio método em teste e os parâmetros que serão utilizados para sua execução. Se os parâmetros forem objetos herdados da classe `ValidationTest` (`Min`, `Max`, `Positive`, `Negative`, `Range`, `InList`, `NotInList`, `Blank` e `NonBlank`), então o algoritmo identifica que é necessário verificar valores limites para determinado parâmetro, de acordo com o tipo de validação.

Por exemplo, se um parâmetro for `Range(56, 790)` (Figura 6.31), serão gerados os testes para valores os 56 e 790, onde é esperado sucesso, ou seja, nenhuma exceção é lançada. Também são feito testes para os valores 55 e 791, para os quais são esperadas exceções. Não obstante, ainda são feitos testes com outros valores menos significativos, tais como 423 (esperado sucesso), 46 e 800 (esperado falha).

```
1  # Referências do UnitTest
2  import unittest
3  # Referências do Python-QAssertions
4  import qassertions as qa
5  from qassertions import Range
6
7  class UmaClasseDoSistemaTests(unittest.TestCase):
8
9      def setUp(self):
10         self.sistema = UmaClasseDoSistema()
11
12     def testValorTemQueSerMaiorOuIgualQue56MenorOuIgualQue790(self):
13         qa.assertValidation(self.sistema.metodoEmTeste, Range(56, 790))
```

Figura 6.31: Exemplo de verificação de validação com casos limites com geração de casos de teste.

A Figura 6.32 mostra como poderiam ficar a implementação dos testes sem usar a geração de casos de teste de validação. Mesmo para os testes de validação de apenas um parâmetro a implementação fica mais extensa. Consequentemente, quanto mais parâmetros precisam ser validados, maior a vantagem do uso da ferramenta. A Figura 6.33 mostra um exemplo de um teste de validação de vários argumentos. É imporante notar que os valores limites são definidos subtraindo e somando 1 dos limites do intervalo, mas também pode-se alterar a precisão para outros valores, como mostra a validação `Max` no exemplo da figura.

Usos Conhecidos: A teoria de testes de software incentiva o uso desse padrão [95]. A ferramenta Python-QAssertions que fornece um método de asserção que gera testes de valores limites. API para criação de jogos de cartas Card Game Engine, que realiza esses testes para verificar erros de implementação.

```
# Referências do UnitTest
import unittest
# Referências do Python-QAssertions
import qassertions as qa

class UmaClasseDoSistemaTests(unittest.TestCase):

    def setUp(self):
        self.sistema = UmaClasseDoSistema()

    def testValorTemQueSerMaiorOuIgualQue56MenorOuIgualQue790(self):
        self.assertRaises(Exception, self.sistema.metodoEmTeste, 46)
        qa.assertDontRaiseAnException(self.sistema.metodoEmTeste, 56)
        self.assertRaises(Exception, self.sistema.metodoEmTeste, 55)
        qa.assertDontRaiseAnException(self.sistema.metodoEmTeste, 423)
        qa.assertDontRaiseAnException(self.sistema.metodoEmTeste, 790)
        self.assertRaises(Exception, self.sistema.metodoEmTeste, 791)
        self.assertRaises(Exception, self.sistema.metodoEmTeste, 800)
```

Figura 6.32: Exemplo de verificação de validação com casos limites sem geração dos casos de teste.

```
# Referências do UnitTest
import unittest
# Referências do Python-QAssertions
import qassertions as qa
# Tipos de validação:
# Números: Min, Max, Positive, Negative, Range
# Listas: InList, NotInList
# Strings: Blank, NonBlank
from qassertions import Min, Max, Range, InList, NotInList, Blank, NonBlank

class UmaClasseDoSistemaTests(unittest.TestCase):

    def setUp(self):
        self.sistema = UmaClasseDoSistema()

    def testValidacaoDeVariosDadosDeEntrada(self):
        qa.assertValidation(self.sistema.metodoEmTeste2,
               Min(5), Max(7, 0.1), 'valor sem regra de validação',
               Range(5, 10), InList([1, 5, 7]))
```

Figura 6.33: Exemplo de verificação de validação com casos limites para diversos parâmetros.

6.5 Antipadrões

A seguir serão descritos antipadrões de automação para testes de unidade, de acordo com o esqueleto definido na Seção 5.4. Os indícios de problemas citados nos padrões possuem mais informações na Seção 5.2.

6.5.1 Gancho para os Testes (*Test Hook*)

Tipo: Testabilidade

Contexto: Testar um sistema pode ser muito complexo, principalmente quando os módulos e objetos estão muito acoplados entre si, como é discutido na Seção 10.3. Quando não é possível substituir nem parte do comportamento do sistema através de Objetos Dublês, os testes podem se tornar inviáveis de serem realizados. Uma solução simples é modificar o próprio comportamento do sistema apenas para a execução dos testes, ou seja, é feito um gancho no sistema para torná-lo testável.

Apesar de simples, essa solução é ruim porque ela polui o código do sistema, tornando-o mais extenso e complexo. Isso aumenta as chances do sistema conter erros, que vai completamente contra o objetivo dos testes automatizados. Não obstante, essa solução ainda pode inibir refatorações para melhorar o *design* do sistema.

Exemplo: A implementação típica é a criação de uma *flag* para indicar que são os testes que estão executando o sistema. Se forem os testes, então é executado o trecho de código substituto e testável do sistema, como é exibido na Figura 6.34.

```
1  // Antipadrão: Gancho para os Testes (Test Hook)
2  public class UmaClasseDoSistema {
3    public boolean TESTANDO = false;
4    // ...
5    public void umaFuncionalidadeComBaixaTestabilidade() {
6      if(TESTANDO) {
7      // Simule algo de forma isolada e controlada para os testes ...
8      }
9      else {
10     // Faça o que deve ser feito ...
11     }
12   }
13   // ...
14 }
```

Figura 6.34: Antipadrão Gancho para os Testes.

Indícios de Problemas Relacionados: *Hard-to-Test Code*, *Test Logic in Production* e *Production Bugs* (vide Seção 5.2).

Referências: Esse antipadrão foi identificado por Meszaros como um padrão [99]. No entanto, ele explica que essa abordagem deve ser utilizada apenas em casos excepcionais, quando é inviável refatorar o sistema para aumentar a testabilidade.

6.5.2 Testes Encadeados (*Chained Tests*)

Tipo: Organizacional

Contexto: É normal que alguns casos de testes sejam parecidos e possuam trechos de código auxiliares em comum. Por isso, refatorações para diminuir a replicação de código-fonte deve ser uma tarefa rotineira durante a criação dos testes automatizados.

Uma situação comum, é que um caso de teste seja muito parecido com o método de *set up* de outro, isso porque os testes podem ser complementares. Uma possível solução para evitar replicação de código para este caso, é fazer com que ambos os testes compartilhem as mesmas variáveis e sejam executados em sequência, de forma encadeada.

Entretanto, essa solução incentiva o uso de informações compartilhadas entre os testes e, consequentemente, dependentes entre si, frágeis, difíceis de serem entendidos e mantidos. Além disso, essa abordagem pode até inviabilizar o uso de ferramentas que otimizam a bateria de testes como um todo, por meio de ferramentas que paralelizam a execução dos testes. Se uma funcionalidade é dependente de outra, pode-se utilizar Objetos Dublês para tornar os testes independentes.

Exemplo/Java/TestNG: A Figura 6.35 mostra um exemplo de teste com a ferramenta TestNG que possui funcionalidades para criar testes encadeados.

```
1  //Referências do TestNG
2  import org.testng.annotations.Test ;
3
4  public class UmaClasseDeTest {
5
6      @Test
7      public void teste1() {
8      }
9
10     // Antipadrão: Testes Encadeados (Chained Tests)
11     // Esse teste só será executado depois do que o teste1 tenha sido executado.
12     @Test(dependsOnMethods="teste1")
13     public void teste2() {
14     }
15 }
```

Figura 6.35: Um exemplo de esqueleto de código Java do antipadrão Testes Encadeados.

Indícios de Problemas Relacionados: *Obscure Test*, *Assertion Roulette*, *Erratic Test*, *Fragile Test*, *Frequent Debugging*, *Slow Tests*, *Buggy Tests*, *High Test Maintenance Cost* (vide Seção 5.2).

Referências: Esse antipadrão foi identificado por Meszaros como um padrão [99]. Em casos excepcionais, esse antipadrão pode ser útil em testes de integração, mas desde que ele seja utilizado com cautela.

Capítulo 7

Testes com Persistência de Dados

Controlar e garantir a qualidade da camada de persistência de dados é fundamental, não apenas devido ao valor da informação, mas também porque dados inconsistentes podem afetar a correção das outras camadas do sistema [64].

Falhas com os dados podem desencadear uma infinidade de situações indeterminadas de erros no sistema que gerencia a camada de persistência. As camadas superiores ao módulo de dados são implementadas para trabalhar com dados corretos, ou seja, não estão preparadas para lidar com situações onde os dados não seguem os formatos e as convenções definidas.

Quando esses erros são identificados antes da causa principal do problema, o processo de identificação e correção das falhas se torna um trabalho ainda mais complexo e demorado. O próprio sistema, ou até mesmo o usuário final, pode ser induzido a criar novos dados problemáticos, o que, talvez, resulte em um ciclo automático de adição de erros e na perda da credibilidade dos dados armazenados.

Por causa da gravidade que os problemas nos dados podem ter, muitas empresas possuem departamentos exclusivos de especialistas para implementar e gerenciar a camada de persistência de dados. Contudo, mesmo os especialistas na área e as ferramentas de persistência de dados não garantem que todas as situações propícias a erros sejam verificadas a cada alteração do sistema.

A camada de persistência pode possuir muitos pontos suscetíveis a falhas. Basicamente, a camada é composta por funcionalidades de escrita e leitura dos dados, que, geralmente, são implementadas com a ajuda de APIs e arcabouços, mas o uso destas ferramentas e a criação da lógica de leitura e escrita nem sempre são tarefas triviais.

Este capítulo irá descrever alguns padrões e técnicas de automação de testes para aumentar a produtividade e tornar os testes mais rápidos e robustos. Além disso, serão discutidos as situações típicas de cenários de testes tanto para sistema de arquivos quanto para bancos de dados.

7.1 Banco de Dados

Para verificar a correção da camada de persistência com banco de dados pode ser necessário tanto testes de unidade como de integração. Basicamente, a escolha do tipo de teste depende do que se está querendo verificar, mas a regra geral é utilizar os testes de unidade sempre que possível e fazer testes de integração para completar a bateria de verificações.

Entretanto, antes de definir os tipos de testes, é necessário entender o que é interessante de ser verificado. A camada de persistência de dados pode ter diversos trechos de código suscetíveis a erros, além de que os tipos de erros podem variar de acordo com os arcabouços utilizados. Por exemplo, sistemas que possuem *queries hardcoded* no código-fonte são muito suscetíveis a erros sintáticos. Já para os sistemas que utilizam arcabouços de ORM[1], são comuns problemas de desempenho, inconsistência de

[1] Arcabouços de ORM (*Object-Relational Mapping*) mapeiam objetos para estrutura de banco de dados relacionais.

dados causados por refatorações etc.

No caso de sistemas integrados com arcabouços ORM, algumas funcionalidades podem ser verificadas com testes de unidade. Tanto as verificações de sintaxe quanto as de lógica da aplicação podem ser isoladas. Além disso, alguns arcabouços disponibilizam bibliotecas específicas com Objetos Dublês e métodos de verificação para facilitar a criação de testes automatizados de unidade para testar o mapeamento dos objetos para a camada de persistência.

Já para verificar a semântica de *queries*, é imprescindível que os testes sejam integrados, pois elas precisam ser interpretadas pelos gerenciadores de banco de dados. Além disso, para testar uma *query* é preciso observar os efeitos colaterais causados na base de dados, ou seja, o teste verifica justamente a integração das funcionalidades à camada de persistência.

Contudo, é importante notar que existem diversos arcabouços que geram algumas *queries* automaticamente. Estas não precisam ser verificadas, pois é responsabilidade da equipe que desenvolveu o arcabouço avaliar se elas estão sendo geradas corretamente. Assim como para todos os tipos de testes de todas as camadas, deve-se testar apenas o sistema de interesse, dispensando as suas dependências. Uma situação excepcional é quando há interesse em verificar o comportamento do sistema sob a ação de erros conhecidos dos arcabouços.

Não obstante, a camada de persistência possui diversas particularidades que podem trazer dificuldades para automação de testes. Entre elas estão o compartilhamento de dados, sistema de conexões de usuários, níveis e permissões de acesso, replicação da bases de dados e auditoria. No decorrer desta seção serão discutidas algumas práticas e padrões para criar testes automatizados robustos para camadas de persistência com banco de dados.

7.1.1 Configuração do Ambiente de Teste

Os testes de unidade da camada de persistência são semelhantes aos de outras camadas. Sendo assim, as dependências devem ser substituídas sempre que possível por Objetos Dublês para que os testes sejam isolados. Já os testes de integração dependem da configuração e execução correta dos ambientes e dos gerenciadores de banco de dados necessários. Todavia, a forma de organização do ambiente reflete diretamente na eficácia e na produtividade dos testes automatizados, por isso é importante conhecer as abordagens de configuração do ambiente.

Bancos de Dados Compartilhados

É comum que empresas possuam um ou mais ambientes próprios para realização de testes (geralmente manuais), que são normalmente conhecidos como ambientes de desenvolvimento, de homologação ou de garantia de qualidade. Estes ambientes são idealmente semelhantes ao de produção e úteis para a realização de testes que integram todos os módulos do sistema e suas dependências. Como estes ambientes geralmente são de alto custo, vários departamentos e projetos acabam compartilhando os recursos e as bases de dados.

Uma das principais vantagens desse ambiente para realização de testes é que ele é próximo do real, o que dá maior credibilidade às verificações e possibilita a criação de cenários de testes fiéis aos comportamentos de usuários reais. Os testes realizados nesse ambiente também são úteis para evitar e identificar problemas de instalação, configuração e de portabilidade, que são aqueles que podem ser associados à frase: “Mas no meu ambiente funcionava!”.

Outra vantagem é que a base de dados é normalmente importada do ambiente real de produção, o que permite que os testes sejam feitos com dados reais de usuários. Além disso, o uso dos dados já existentes pode facilitar a criação dos testes, pois dispensa parte da tarefa de preparar os dados para execução dos testes. A criação do *set up* dos testes que envolvem banco de dados exigem, via de regra, um trabalho árduo de implementação, principalmente quando a modelagem do banco é muito complexa.

No entanto, é necessário um esforço para pesquisa e avaliação dos dados antes que eles sejam utilizados para os testes, o que pode ser mais custoso do que a criação de novos dados. Além disso, dificilmente os dados pesquisados poderão ser reutilizados no futuro, já que tanto os próprios testes quanto outras pessoas podem modificar os dados.

Não obstante, mesmo com as outras vantagens citadas, esse tipo de ambiente deve ser evitado para grandes baterias de testes, sejam eles automatizados ou manuais. Ambientes compartilhados trazem inúmeras dificuldades ao processo de criação, manutenção e execução dos testes, tornando a produtividade muito baixa.

Uma das dificuldades deve-se à preocupação aos dados já existentes e aos criados pelos casos de teste. É necessário um esforço adicional para organizar e controlar os ambientes e as equipes para que a base de dados não seja degradada com a utilização. Mesmo assim, é muito difícil manter o ambiente estável por muito tempo, pois muitos testes precisam criar cenários críticos. Por exemplo, podem exigir mudanças de configuração, geração de dados inválidos para testes, base de dados vazia, entre outras.

Outro problema difícil de resolver é a execução concorrente de testes em um ambiente compartilhado, sejam eles testes de um mesmo projeto ou de projetos diferentes. Os testes não possuem o conceito de transação, dessa forma, os comandos são executados no gerenciador de banco de dados de forma indeterminada. Isso resulta em testes intermitentes e erros difíceis de depurar. Para ilustrar, enquanto um teste pode estar inserindo dados para verificação, outro pode estar limpando a base para preparar o ambiente.

Ainda, a produtividade da verificação do sistema em teste também é prejudicada devido à redução do desempenho dos testes. A memória e o poder de processamento do ambiente são compartilhados entre diversos usuários que podem estar realizando operações demoradas e que exigem muitos recursos. Além disso, à medida que os testes vão deixando resíduos de dados na base, as *queries* de busca vão se tornando cada vez mais lentas.

Por estas características, a relação custo/benefício de se automatizar os testes nesse tipo de ambiente não é vantajoso. O código dos testes tende a ficar mais extenso e complexo para que eles fiquem mais robustos. Ainda assim, os testes são mais sucestíveis a serem frágeis, lentos e intermitentes do que se forem executados em outros tipos de ambientes, como veremos nas próximas subseções.

De qualquer maneira, esses ambientes são úteis para garantir a qualidade dos sistemas de software. Eles podem ser utilizados para buscar apenas erros mais específicos, a partir de testes de fumaça, de sanidade, de instalação e de configuração. Dessa forma, apenas uma pequena parcela de todas as verificações são feitas nesse ambiente, o que facilita a organização e a manutenção dos testes e do ambiente. Dependendo do tamanho da bateria dos testes e de outras características, pode até ser mais vantajoso não os automatizar.

Bancos de Dados Locais

Banco de Dados Local é a configuração em que os gerenciadores de banco de dados para testes são instalados nas máquinas de cada membro da equipe de desenvolvimento. Dessa forma, cada bateria de testes de cada membro da equipe pode ser executada paralelamente, sem grandes preocupações, pois os recursos do ambiente e os dados não são compartilhados. Vale notar que a base de dados ainda é compartilhada para mesma bateria de casos de testes, por isso os testes de uma mesma bateria não devem ser executados paralelamente, a não ser que sejam implementados de uma maneira completamente independente dos demais.

As vantagens e desvantagens desse tipo de configuração são praticamente opostas às de ambientes compartilhados, o que torna as duas opções complementares para verificação da qualidade dos sistemas de software. As configurações de hardware e software das máquinas de desenvolvimento podem apresentar grandes diferenças em relação ao ambiente de produção, no entanto, cada desenvolvedor tem total autonomia para criação de diferentes cenários de teste.

A preparação dos dados para os testes é feita em métodos de *set up* ou nos próprios métodos de teste, ou seja, há maior autonomia, controle e flexibilidade sobre os dados criados. Além disso, os registros nas tabelas da base de dados podem ser criados de forma abstrata, com a ajuda das próprias funcionalidades do sistema, o que facilita a criação dos cenários de teste.

O desempenho das baterias de testes também pode ser significativamente melhor com bancos de dados locais do que com ambientes compartilhados. Apesar de as máquinas de desenvolvimento não terem todo o potencial de processamento daquelas de servidores, elas são normalmente utilizadas por apenas um usuário. Outro aspecto importante do desempenho deve-se à menor quantidade de dados, já que apenas um usuário de um projeto insere dados na base. Por último, bancos de dados locais dispensam a comunicação entre máquinas, que pode até ser o principal gargalo de desempenho de muitos ambientes.

Contudo, o sucesso deste tipo de ambiente depende da configuração e organização dos repositórios de código-fonte. Como os bancos de dados são locais, cada desenvolvedor ou testador pode possuir configurações que são específicas de sua máquina, como o diretório de instalação do banco de dados. Para isso, existem duas soluções simples: ou selecionar alguns arquivos que serão ignorados durante a sincronização com o repositório, ou, então, utilizar endereços de arquivos relativos e outras configurações dinâmicas, ou seja, que não estejam *hard coded*.

Para finalizar, esse tipo de configuração de ambiente de teste também pode ser de alto custo se for necessário comprar licenças do gerenciador de banco de dados para cada máquina de desenvolvimento. Contudo, este problema pode ser facilmente resolvido se as máquinas de desenvolvimento utilizarem bancos de dados livres para realização dos testes. Entretanto, é fundamental que, em algum momento, as baterias de testes sejam executadas sobre o gerenciadores de banco de dados utilizados em ambiente de produção, o que pode ser feito idealmente no ambiente de integração contínua.

Bancos de Dados em Memória

Bancos de Dados em Memória utilizam, a priori, a memória principal do computador em vez dos discos de armazenamento. Por isso, este tipo de banco de dados é intrinsecamente mais rápido para pequenas quantidade de dados do que os tradicinais, que utilizam principalmente os discos de armazenamento.

Dessa forma, eles podem ser utilizados como banco de dados locais para o ambiente de testes. Além de todas as vantagens já descritas na subseção anterior, as baterias de testes podem tornar-se significativamente mais rápidas.

O principal obstáculo para utilizar este tipo de ambiente de testes são as grandes diferenças em relação aos bancos de dados de produção, que geralmente utilizam bancos de dados tradicionais. Por exemplo, nem todos os bancos de dados em memória oferecem *triggers* e *stored procedures*, o que pode impossibilitar os testes de vários módulos do sistema.

Mesmo assim, o recomendado é utilizar os bancos de dados em memória sempre que eles permitirem a realização de uma grande parcela dos cenários de testes necessários. No entanto, é fundamental que em algum momento seja executada a bateria de testes no ambiente real de produção para evitar erros de incompatibilidade. O mais indicado é configurar o ambiente de integração contínua [52] o mais próximo do real, para confirmar que os resultados dos testes realizados durante o desenvolvimento são confiáveis.

É importante observar que mesmo com banco de dados em memória ainda é necessário cuidados para executar os testes em paralelo. Se for necessário paralelizar os testes, uma possível solução é criar uma instância distinta de banco de dados para cada caso de teste (Padrão **Uma Instância de Banco de Dados por Linha de Execução**, Seção 7.2.1).

7.2 Padrões

Para que os testes com persistência de dados sejam de qualidade, é fundamental que a bateria como um todo esteja muito bem organizada. Para isso, pode-se utilizar uma combinação de diversos padrões organizacionais de testes de unidade, tais como *Testcase Superclass*, *Test Helper*, *Parametrized Test* e *Test Utility Method* [99]. Todos esses padrões são úteis para reutilização de código e dos dados.

No entanto, independente de como será a organização, é fundamental que o estado do banco de dados no início de cada teste seja sempre o mesmo: sem dados ou apenas com um conjunto de dados essenciais para o funcionamento do sistema. Para isso, pode-se utilizar qualquer padrão que seja responsável por limpar todos os dados gerados por um teste, por exemplo, o *Table Truncation Tear down* [99]. Apenas é importante notar que é inviável controlar quais foram os dados gerados por cada cenário de teste. Por isso, a solução mais simples e fácil é remover absolutamente todos os dados da base e recriar novamente os essenciais. Tendo esse padrão aplicado, novos padrões podem ser integrados à bateria de testes, como alguns que serão descritos a seguir.

7.2.1 Uma Instância de Banco de Dados por Linha de Execução

Tipo: Desempenho

Quando utilizar: Quando uma bateria de testes se torna lenta devido à grande quantidade de casos de testes. O que deve ser levado em conta é se o tempo de execução da bateria de testes está impactando na produtividade do desenvolvimento. O pré-requisito para que esse padrão seja utilizado é que os testes possam ser executados em paralelo, ou seja, eles não manipulem informações compartilhadas, tais como variáveis globais. Além disso, também deve ser avaliado se a infraestrutura é adequada para paralelizar os testes.

Intenção: Utilizar diferentes instâncias de bancos de dados para possiblitar a execução dos testes em paralelo, visando melhor desempenho.

Motivação: Os testes que envolvem bancos de dados tendem a possuir um desempenho inferior aos testes de unidade devido à preparação do comando a ser executado (*queries*) e à comunicação entre o aplicativo e o gerenciador de banco de dados. Em alguns casos, até é aceitável um teste demorar alguns segundos para ser realizado. Por isso, o desempenho de uma grande bateria desses testes integrados pode ser insatisfatória.

Uma boa solução de otimização é paralelizar a execução dos testes, no entanto, historicamente os arcabouços de testes foram implementados para executarem os testes sequencialmente, inclusive os que dependem de gerenciadores de banco de dados. Por isso, é preciso adaptá-los para que seja possível executar diversos testes em paralelo.

Executar testes integrados com bancos de dados paralelamente é difícil porque a própria base de dados se torna um recurso compartilhado entre os testes, o que torna os testes dependentes entre si. As alterações feitas por um cenário de teste no esquema (*schema*) do banco, ou mesmo nos próprios dados, podem quebrar os demais.

Solução: Executar a bateria de testes paralelamente em uma certa quantidade de linhas de execução, sendo que cada uma delas utilizará sua própria e independente instância do banco de dados.

Consequências: A bateria dos testes é otimizada de maneira global, o que pode dispensar micro-otimizações em cada cenário de teste.

Implementação: O aplicativo que executará os testes deve ter uma linha principal de execução que é responsável por criar e gerenciar uma fila de cenários de testes a serem executados. Além disso, é preciso que ela também inicialize e gerencie as demais linhas de execução que irão ser executadas em paralelo para rodar os testes automatizados. Opcionalmente, o comando para execução da bateria de testes pode ser parametrizado para receber a quantidade de linhas de execução que deverão ser criadas.

Cada linha de execução deve ser responsável por instanciar e configurar sua própria instância de banco de dados. Dado que o ambiente está configurado, a linha de execução desempilha um cenário de teste para ser executado sempre que ela estiver ociosa. Quando todas as linhas de execuções estiverem ociosas e a pilha estiver vazia, a bateria de testes terminou de ser executada.

7.2.2 Geração Dinâmica de Dados

Tipo: Robustez

Quando utilizar: É recomendado o uso em praticamente todos os testes que envolvem dados persistentes em gerenciadores de bancos de dados, especialmente aqueles que precisam de muitos dados pré-populados, tais como testes de buscas, de geração de relatórios ou de modelos que possuem um grande número de dependências.

Intenção: Gerar automaticamente objetos tolos para facilitar a criação dos testes, assim como torná-los mais robustos e legíveis.

Motivação: Os testes que envolvem a camada de persitência são os responsáveis por criar todos os dados necessários para sua execução. Esses dados podem ser classificados em duas categorias: os principais, que caracterizam o cenário de teste, e os tolos ((Seção 6.4.3)), que são criados apenas para evitar erros de compilação ou de execucão, causados comumente por validações realizadas pelo próprio gerenciador do banco de dados.

Um problema comum ocorre quando são necessários muitos dados tolos para realização de um cenário de teste. Além de ser uma tarefa repetitiva e tediosa, a inicialização dessas informações polui o código de teste. Ainda, esse problema pode ser multiplicado quando um cenário de teste precisar de dados já existentes na base.

Para amenizar o problema de legibilidade, os dados tolos podem ser definidos de forma estática, geralmente em arquivos externos que serão carregados durante a preparação dos testes (Figura 7.1). Entretanto, essa não é uma boa solução. Um dos principais problemas dessa abordagem é que a quantidade de dados para os testes cresce rapidamente à medida que novos casos de testes são adicionados, o que torna a manutenção dos testes muito complicada, pois qualquer mudança na modelagem pode implicar que todos os arquivos de testes precisem ser adaptados.

```
1  - model: Pessoa
2    id: 1
3    nome: Sharon Janny den Adel
4    sexo: F
5    data_nascimento: 1974/07/12
6    nacionalidade: Holandesa
7
8  - model: Pessoa
9    id: 2
10   nome: Mark Knopfler
11   sexo: M
12   data_nascimento: 1949/08/12
13   nacionalidade: Escocesa
```

Figura 7.1: Exemplo de dados estáticos em um arquivo no formato YAML.

Uma possível solução para esse problema é tentar reaproveitar os dados para diferentes cenários de teste, o que diminui a replicação de informações e facilita a manutenção em caso de mudanças na modelagem dos dados. Entretanto, essa abordagem possui problemas ainda maiores. Compartilhar informações entre testes os tornam frágeis, de modo que qualquer pequena alteração na bateria de testes ou no próprio sistema pode quebrar um grande conjunto de testes, o que dificulta a identificação do erro e, consequentemente, torna a manutenção muito complicada (Seção 7.1.1). Para dificultar ainda mais, essa abordagem prejudica a clareza dos testes, pois não fica explítico quais os dados que são realmente pertinentes para cada caso de teste.

Solução: Criar métodos que geram dinamicamente objetos de dados populados com dados tolos e válidos. Tendo esse objeto instanciado, basta atualizar as informações principais com valores pertinentes para simular o cenário de teste desejado.

Consequências: Primeiramente, não é mais necessário a criação e manutenção de arquivos estáticos de dados, muito menos o gerenciamento de dados compartilhados. Além disso, os testes são mais facilmente adaptados às mudanças da modelagem dos dados. Por fim, a clareza e a legibilidade dos testes são melhoradas, não só porque os testes tendem a ficar mais enxutos e coesos, como também porque os dados tolos de um objeto podem ser escondidos.

Implementação: No escopo dos testes, deve-se criar uma biblioteca de funcionalidades que são responsáveis por instanciar e persistir objetos de dados populados com objetos tolos. Contudo, os objetos tolos precisam ser únicos, para evitar problemas de integridade de dados, e não aleatórios, para facilitar a depuração de testes incorretos e para não tornar os testes intermitentes. Também é importante notar que, como um objeto de dado pode depender de outros, essa biblioteca pode fazer chamadas internas para reaproveitamento de código.

Já para tornar essa biblioteca mais flexível, pode-se permitir que um cenário de teste consiga personalizar as informações que forem pertinentes para seu contexto, ou seja, os métodos de geração de objetos devem ser parametrizáveis. Entretanto, o trabalho de implementação dessa biblioteca de objetos de dados é semelhante para todos os objetos de dados, logo, é altamente recomendável que seja utilizado reflexão para criação de bibliotecas genéricas, ou seja, que consigam criar dinamicamente qualquer tipo de objeto de dados.

Exemplo Python/Unittest: Para testar uma funcionalidade de busca de pessoas, muitos objetos de dados `Pessoa` precisarão ser criados para poder simular todos os cenários desejados. Por isso, antes mesmo da criação dos testes, é importante criar classes para gerar dinamicamente os objetos de dados desejados. Esse objeto terá a responsabilidade de criar um objeto válido, com pelo menos os atributos obrigatórios do objeto preenchidos, mas de modo que possa ser customizado de acordo com a necessidade dos testes. A Figura 7.2 mostra um exemplo de implementação desse objeto (`GeradorDePessoas`) e de um teste que o utiliza.

Exemplo Python/Unittest/Django Dynamic Fixture: Como foi citado nos tópicos anteriores, a implementação dessa classe de geração de dados é um trabalho árduo e repetitivo . Por isso, preferencialmente, é recomendado utilizar bibliotecas genéricas de geração dinâmica de dados, como é o caso da Django Dynamic Fixture, para sistemas Web que utilizam o arcabouço Django para Python. A Figura 7.3 mostra alguns exemplos de teste para a mesma funcionalidade de busca de pessoas.

Usos Conhecidos: A ferramenta Django Dynamic Fixture implementa esse padrão para aplicações que utilizam o arcabouço Django para Python.

```python
from cadastro.models import Pessoa

import unittest

class GeradorDePessoas(object):
    def cria(nome='X', sexo='F', nascimento='2000/01/01'):
        p = Pessoa(nome=nome, sexo=sexo, nascimento=nascimento)
        p.save()
        return p

gerador = GeradorDePessoas()

class BuscaDePessoasPeloNomeTest(unittest.TestCase):
    def test_deve_encontrar_pessoas_que_possuem_parte_do_nome_buscado(self):
        pessoa_que_deve_ser_encontrada = gerador.cria(nome='Sharon Janny den Adel')
        pessoa_que_nao_deve_ser_encontrada = gerador.cria(nome='Mark Knopfler')

        pessoas_encontradas = Pessoa.objects.com_parte_do_nome('Jan')

        self.assertTrue(pessoa_que_deve_ser_encontrada in pessoas_encontradas)
        self.assertTrue(pessoa_que_nao_deve_ser_encontrada not in
            pessoas_encontradas)
```

Figura 7.2: Exemplo em Python de classe de geração dinâmica de um objeto de dados específico.

```python
from django_dynamic_fixture import get
from cadastro.models import Pessoa

import unittest

class BuscaDePessoasPeloNomeTest(unittest.TestCase):
    def test_deve_encontrar_pessoas_que_possuem_parte_do_nome_buscado(self):
        # Apenas o atributo nome é pertinente para esse caso de teste.
        # Portanto, os outros atributos do objeto Pessoa devem ficar ocultos.

        # cria e salva na base de dados uma pessoa com um nome específico.
        pessoa_que_deve_ser_encontrada = get(Pessoa, nome='Sharon Janny den Adel')
        pessoa_que_nao_deve_ser_encontrada = get(Pessoa, nome='Mark Knopfler')

        pessoas_encontradas = Pessoa.objects.com_parte_do_nome('Jan')

        self.assertTrue(pessoa_que_deve_ser_encontrada in pessoas_encontradas)
        self.assertTrue(pessoa_que_nao_deve_ser_encontrada not in
            pessoas_encontradas)

    def test_deve_ignorar_se_texto_buscado_esta_em_minuscula_ou_maiuscula(self):
        pessoa_que_deve_ser_encontrada = get(Pessoa, nome='Kyra Gracie Guimarães')
        pessoa_que_nao_deve_ser_encontrada = get(Pessoa, nome='Roger Gracie Gomes')

        pessoas_encontradas = Pessoa.objects.com_parte_do_nome('guiMARães')

        self.assertTrue(pessoa_que_deve_ser_encontrada in pessoas_encontradas)
        self.assertTrue(pessoa_que_nao_deve_ser_encontrada not in
            pessoas_encontradas)

class BuscaDePessoasPorSexoTest(unittest.TestCase):
    def test_deve_encontrar_apenas_pessoas_do_sexo_buscado(self):
        # Dessa vez, o nome não é importante, pode-se utilizar um nome tolo
            qualquer,
        # que é criado pela biblioteca automaticamente.
        pessoa_que_deve_ser_encontrada = get(Pessoa, sexo='F')
        pessoa_que_nao_deve_ser_encontrada = get(Pessoa, sexo='M')

        pessoas_encontradas = Pessoa.objects.de_sexo('F')

        self.assertTrue(pessoa_que_deve_ser_encontrada in pessoas_encontradas)
        self.assertTrue(pessoa_que_nao_deve_ser_encontrada not in
            pessoas_encontradas)
```

Figura 7.3: Exemplo do padrão de Geração Dinâmica de Dados com a biblioteca genérica de objetos de dados Django Dynamic Fixture.

Capítulo 8

Testes de Interface de Usuário

Interface de usuário (IU) é a camada que cria a ligação entre o usuário e as camadas internas do sistema. Os tipos mais comuns de IUs são as interfaces gráficas (GUI, de *Graphic User Interface*) e as interfaces Web (WUI, de *Web User Interface*). Os componentes das IUs observam eventos originários de dispositivos de hardware, identificam os eventos e fazem chamadas a outros módulos do sistema por meio de troca de mensagens ou de chamada de métodos.

As IUs são fundamentais para muitos projetos e podem ser a camada determinante para o sucesso de um sistema. Elas podem tornar o uso do sistema intuitivo, o que possibilita que mesmo pessoas inexperientes consigam aprender a utilizá-lo sem a necessidade de um estudo árduo de documentos. Atrativos que tornem o uso da interface mais fácil e interessante são diferenciais que valorizam muito o sistema como um todo, vide as interfaces *multi touch* que têm atraído a atenção de milhões de usuários em todo o mundo [103].

Devido à importância das IUs, é interessante que seja dedicada uma parcela considerável de esforço para que ela seja bem implementada. A qualidade das IUs pode ser interpretada pelas características de correção, usabilidade, acessibilidade, portabilidade e beleza. Além disso, estes fatores são, muitas vezes, determinantes para a aceitação do sistema por parte do cliente [139].

No entanto, controlar e garantir a qualidade dessa camada do sistema é uma tarefa delicada. A interface de usuário fica exposta ao manuseio dos seres humanos, que possuem comportamentos indeterminados. Portanto, esta camada está sujeita a lidar com uma infinidade de combinações inusitadas de eventos. Além disso, muitos sistemas estão sujeitos a alterações frequentes da interface, principalmente os sistemas Web que evoluem constantemente seu leiaute [73].

Essas particularidades das IUs requerem que o controle de qualidade seja ágil (de fácil adaptação), eficaz e seguro contra erros de regressão. Métodos formais são geralmente inviáveis devido às mudanças constantes que a interface está sujeita e também por causa do pouco embasamento matemático do código-fonte, com exceção de componentes *multi touch* que precisam interpretar traços e figuras geométricas. Já os testes manuais são eficazes para certos aspectos de qualidade, como beleza e usabilidade, mas não são seguros contra erros de regressão, principalmente por causa da infinidade de maneiras que uma interface pode ser manipulada.

A automação de testes é uma alternativa satisfatória para o controle de qualidade das IUs, por todos os motivos que já foram discutidos na Parte I. Contudo, para que os testes automatizados para esta camada sejam de qualidade, é necessário o uso de ferramentas especializadas, que permitam a escrita de testes legíveis com alta produtividade.

No decorrer deste capítulo serão discutidos alguns aspectos de ferramentas que são úteis para criação de testes automatizados de interface com qualidade. Também serão apresentadas abordagens próprias para criação de testes de interface, assim como padrões e antipadrões que são específicos para este tipo de teste.

8.1 Princípios Básicos

Como muitas das responsabilidades de um sistema são associadas à interface de usuário, é mais difícil testá-la do que módulos isolados que são responsáveis por apenas uma tarefa. Por mais que o sistema e a interface sejam bem modularizados, os problemas de outras partes do sistema podem acabar interferindo na interface gráfica.

Por isso, fazer testes automatizados de interface de qualidade requer não só conhecimento de padrões e antipadrões de testes, como também muita disciplina. Para organizar bem os testes é necessário conhecer primeiramente as responsabilidades da interface de usuário, assim como modularizar o sistema corretamente para facilitar a escrita dos mesmos.

Um projeto mal arquitetado não separa as regras de negócio e de dados do código da interface de usuário. Por exemplo, comandos SQL integrados com código-fonte responsável por desenhar a interface. Este problema de arquitetura é comum, por isso, hoje em dia, muitos arcabouços possuem uma estrutura definida para evitar que isso aconteça, principalmente os arcabouços para desenvolvimento de sistemas Web.

Os arcabouços para criação de interfaces gráficas seguem, em geral, a arquitetura baseada no padrão MVC (*Model-View-Controller*) [32]. Alguns seguem variações deste padrão, mas sempre incentivando a separação do sistema em módulos bem definidos, o que torna o software mais fácil de implementar e testar. O padrão MVC sugere uma arquitetura na qual o sistema é dividido em três partes: (1) **Modelo** responsável por encapsular as regras e os dados do sistema; (2) **Visão**, que exibe os dados e (3) **Controlador**, que recebe informações do usuário e as envia para camada de Modelo. O Controlador recebe da Visão as ações do usuário e as mapeia para as funcionalidades pertinentes ao Modelo. A partir dos resultados, o Controlador manuseia a Visão apropriadamente. A Figura 8.1 possui um diagrama simplificado mostrando as associações entre as camadas.

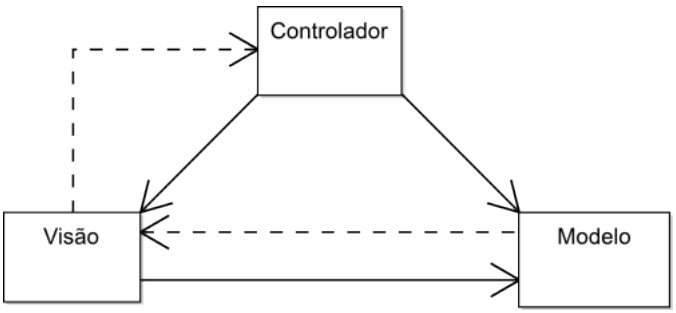


Figura 8.1: Diagrama simplificado do padrão MVC. As linhas sólidas indicam associações diretas enquanto as tracejadas representam associações indiretas.

Dado que a aplicação segue a arquitetura MVC, pode-se dizer que a interface gráfica está associada à Visão, apesar de nem toda Visão ser uma interface gráfica. O termo Visão é mais amplo, pois abrange tudo que tenha o propósito de exibir os dados. Por exemplo, Visão pode ser imagens, gráficos, tabelas, sons etc. Entretanto, os testes de interface também envolvem, comumente, a camada Controlador, pois ela interfere diretamente nas informações que serão exibidas, além de que o código-fonte do controlador pode estar entrelaçado com o código dos arcabouços que facilitam a criação da Visão.

Controlador

A responsabilidade do Controlador é basicamente conectar duas outras partes do sistema, pois ela recebe os dados do usuário a partir da Visão e as envia apropriadamente para camada Modelo. Por isso a sua lógica não deve ter grande complexidade. Entretanto, erros do controlador podem ser tão desastrosos

quanto erros em módulos mais complexos. Portanto, automatizar os testes dos controladores também é importante.

Os controladores podem ser avaliados por meio de efeitos colaterais causados pelo manuseio da interface de usuário. Como sua execução possivelmente altera o estado da interface, é possível seguir essa estratégia. Essa forma de testar é especialmente útil nas situações em que o código do controlador é difícil de ser isolado e testado. Sistemas legados mal implementados ou os que usam arcabouços complexos tornam inviável a criação de testes de unidade de qualidade. Estes testes integrados com a camada de interface do usuário se tornam uma alternativa produtiva e segura, até que os controladores sejam refatorados.

Contudo, sempre que possível e viável, é preferível testar isoladamente os controladores com testes de unidade, assim como qualquer outro componente do sistema. Testes de unidade são mais fáceis e rápidos de serem escritos e mantidos do que testes que integram diversos módulos do sistema.

Os controladores da interface não possuem tantas particularidades quanto à Visão, já que ela não é acessível ao usuário final. Entretanto, ela possui duas características expressivas que podem afetar sua testabilidade: (1) é comum que a lógica da camada fique entrelaçada com trechos de código para satisfazer as necessidades de arcabouços e (2) ela comumente possui referências para diversos outros módulos do sistema.

No caso de aplicações Web, o controlador está sujeito a lidar com detalhes do protocolo HTTP. Ao ser chamado, ele pode ter de processar as informações que vieram por meio de requisições GET ou POST. Para o retorno, ele pode lançar os erros 403 (funciolidade proibida) e 404 (página não encontrada) ou, até mesmo, trechos de código HTML, JavaScript, JSON e XML para serem exibidos apropriadamente na Visão.

O mesmo ocorre para o caso dos controladores de interfaces gráficas, os quais são mais fáceis de ser implementados com a ajuda de arcabouços. O controlador obtém as informações vindas do usuário através de objetos ou estruturas de dados próprias. Sua execução também pode ter de seguir a forma que os arcabouços são organizados para que a interface seja implementada corretamente. Por exemplo, programas escritos com o arcabouço Swing para linguagem Java precisam se preocupar com o modo que irão fazer as chamadas para as camadas base do sistema, pois a execução lenta das tarefas pode impedir que o processo de renderização da interface seja executado, bloqueando a interface.

A primeira recomendação para realização de bons testes de unidade dos controladores é utilizar o padrão *Humble Object* (Seção 6.4) sempre que for pertinente. Dessa forma, o teste fica direcionado totalmente à lógica do sistema enquanto a lógica do arcabouço de apoio é ignorada. Erros dos arcabouços podem afetar o sistema, mas sempre supomos que ele funciona como o esperado. Se o arcabouço possuir um erro já conhecido, então pode ser implementado um teste para verificar se o sistema consegue administrá-lo apropriadamente.

Atualmente, diversos arcabouços fornecem não apenas seus arquivos compilados como, também, bibliotecas para facilitar a escrita de testes automatizados dos módulos que utilizem suas funcionalidades. Isso facilita e aumenta a produtividade da criação de testes. É o caso dos arcabouços Web *Django* e *Ruby on Rails* que fornecem bibliotecas que permitem simular requisições de usuários e que possuem métodos de verificações que tornam os testes mais legíveis e fáceis de serem escritos.

Outra recomendação é implementar os controladores de um modo que seja possível injetar as dependências [113], isto é, que o teste consiga substituir os objetos da camada de negócios por Objetos Dublês. Dessa forma, a persistência de dados e a comunicação entre servidores podem ser facilmente isoladas.

Visão

Teste de interface verifica a correção da interface do usuário. Outros tipos de testes também avaliam a interface, mas utilizando outros critérios, como, por exemplo, os testes de usabilidade, acessibilidade,

leiaute e aceitação. Portanto, as ferramentas de testes de interface podem ser utilizadas para a realização desses outros tipos de testes. No entanto, como foi discutido no Capítulo 4, é importante focar no que se está querendo verificar, assim como é imprescindível não misturar baterias de testes com objetivos distintos. É possível separar as baterias de testes que envolvem a interface sem a necessidade de replicação de código-fonte, como será discutido na seção de padrões deste capítulo.

Existem duas abordagens para realização dos testes que envolvem a interface de usuário: (1) testar a implementação da interface, verificando o uso das bibliotecas e arcabouços de MVC e de renderização; ou (2) fazer os testes diretamente na interface a partir da simulação de um usuário final. Essas duas abordagens podem ser simplificadamente ditas como testes de caixa-branca e caixa-preta, respectivamente. Na prática, as duas abordagens podem ser implementadas com um certo conhecimento do código-fonte, ou seja, o termo mais apropriado seria caixa-cinza em vez de caixa-preta.

A abordagem de caixa-branca nada mais é do que fazer testes de unidade para a Visão. Para isso, é necessário a criação dos eventos que representam as ações dos usuários e é importante isolar a interface dos controladores. A interface também pode ter métodos auxiliares que encapsulam parte de sua lógica. Esses métodos também devem ser testados, como qualquer outra regra do sistema. Para esse tipo de teste, podem ser utilizadas as ferramentas normais de testes de unidade, como as da família xUnit, assim como as ferramentas de criação de Objetos Dublês (vide Seção 6.2).

No caso das interfaces Web, é mais complexo fazer os testes de unidade. A interface é implementada usando linguagem de marcação (HTML) e outras que são interpretadas pelo navegador Web, como JavaScript. O arcabouço JsUnit, que é a versão JavaScript da família xUnit, utiliza o próprio navegador para executar os testes e fornece um servidor Web leve para tornar possível a execução automática, por exemplo, em servidores de integração contínua, como descrito na Seção 4.5.

Além disso, as aplicações Web são distribuídas: a interface do usuário fica localizada no lado do cliente, enquanto os controladores e o modelo ficam no lado do servidor. Isso impossibilita que os controladores ou outros módulos do sistema sejam facilmente substituídos por Objetos Dublês. Essa dificuldade é um tópico interessante para pesquisas futuras. Outra dificuldade ainda não solucionada é medir a cobertura dos testes em relação ao código (vide Capítulo 10) de interfaces Web.

Existem ferramentas que fazem chamadas ao sistema pela interface simulando um usuário, mas o teste passa a ser integrado, pois envolve também o lado do servidor. É o caso das ferramentas HtmlUnit[1], JWebUnit[2] e Selenium[3]. Elas simulam os usuários finais e coletam informações da interface para verificação. Alguns detalhes da implementação são utilizados, tais como propriedades de componentes, como, por exemplo, nomes dos componentes, identificadores etc. No entanto, grande parte dos detalhes de implementação da interface e do controlador são abstraídos pelas ferramentas.

Também existem ferramentas análogas para fazer testes em interfaces *Desktop*. Assim como as ferramentas para testes Web podem depender do navegador, as de interfaces *Desktop* podem depender do gerenciador de janelas do sistema operacional, da linguagem de programação e das bibliotecas utilizadas para implementar a interface do sistema. Por exemplo, as ferramentas Fest e Jemmy são específicas para testes de interfaces gráficas implementadas em Java Swing.

Essas ferramentas são as mesmas utilizadas em um teste caixa-preta. Entretanto, mais importante do que o conhecimento dos testes em relação ao código-fonte, é se os testes irão substituir módulos do sistemas por Objetos Dublês para tornar o teste menos acoplado. Essa decisão dependerá dos tipos de erros que serão verificados, assim como da facilidade de implementação dos testes e do uso das ferramentas.

[1] Apesar de o nome sugerir que a ferramenta é da família xUnit, HtmlUnit é um navegador leve escrito em Java que não renderiza as páginas, mas consegue processar o código HTML e interpretar o código JavaScript. Ele é utilizado para testes de integração e serve como base para muitas ferramentas, tanto para testes em Java quanto para outras linguagens, como Python e Ruby.

[2] JWebUnit é um *Wrapper* do HtmlUnit para facilitar a escrita dos testes e torná-los mais claros.

[3] Selenium é uma ferramenta escrita em JavaScript que é executada internamente em navegadores reais. Atualmente, está sendo unificada com a ferramenta WebDriver, que possui o mesmo propósito da JWebUnit.

8.2 Testes que Simulam Usuários

Os testes que simulam ações dos usuários possuem peculiaridades importantes. O modo pelo qual o sistema é exercitado não é por intermédio de chamadas diretas a objetos e módulos do sistema, mas, sim, por arcabouços que manipulam a própria interface de forma transparente, sem a necessidade de conhecer a fundo o código-fonte. Já a forma de verificação pode variar dependendo do que se está querendo verificar. Podem ser feitas verificações diretamente no estado da interface, ou, ainda, em outras camadas do sistema, como na camada de persistência de dados.

A lógica dos testes de interface segue sempre uma estrutura definida, que é ilustrada pelo exemplo de uso da ferramenta HtmlUnit na Figura 8.2. Primeiro é preciso localizar um componente da página (linhas 17, 19 e 23) para, em seguida, simular algum evento do usuário final (linhas 21 e 26) ou capturar propriedades para fazer as verificações (linha 28). Os eventos mais comuns são cliques do *mouse* e teclas digitadas. Entretanto, existem ferramentas que também disponibilizam métodos para simular eventos de arrastar e soltar (*drag and drop*), apertar e segurar teclas etc.

```
1  // referências do JUnit
2  import org.junit.Test;
3  import static org.junit.Assert.assertEquals;
4  // referências do HTMLUnit
5  import com.gargoylesoftware.htmlunit.WebClient;
6  import com.gargoylesoftware.htmlunit.html.*;
7
8  public class BuscaTests {
9
10   @Test
11   public void buscaComSucessoDeveRedirecionarParaPaginaDeResultados() {
12     // Cria um cliente (navegador)
13     WebClient cliente = new WebClient();
14     // Abre uma página armazenada em um servidor local
15     HtmlPage pagina = cliente.getPage("http://localhost:8000");
16     // Localiza o form de busca
17     HtmlForm formulario = pagina.getFormByName("busca_form");
18     // Localiza o campo de texto para busca do formulário
19     HtmlTextInput texto_busca = formulario.getInputByName("texto_busca");
20     // Digita o texto para busca
21     texto_busca.setValueAttribute("tdd");
22     // Localiza o componente para submeter o form
23     HtmlSubmitInput botao_busca = formulario.getInputByName("busca");
24     // Simula um clique no botão de envio de dados para o servidor
25     // Abre a página de resultados
26     HtmlPage pagina_resultados = botao_busca.click();
27     // Verifica se a ação foi executada corretamente
28     assertEquals("Resultados da Busca - tdd", page.getTitleText());
29   }
30 }
```

Figura 8.2: Exemplo de teste de interface Web com HtmlUnit.

As verificações deste exemplo são feitas com auxílio da ferramenta JUnit. Como a HtmlUnit apenas cria um navegador para execução dos testes, é interessante a utilização de outras ferramentas que facilitem as verificações. Outras ferramentas também fornecem métodos de verificações que são mais pertinentes para testes de interface do que as da família xUnit, como é o caso da JWebUnit e da Selenium. Esses métodos podem ser desde açucares sintáticos que melhoram a legibilidade dos testes até métodos fundamentais, por aumentar consideravelmente a produtividade da automação. Além disso,

algumas ferramentas possuem gravadores de interação, discutidos a seguir.

8.2.1 Gravadores de Interação

Gravadores de Interação são ferramentas que detectam ações executadas por usuários reais sobre um programa e geram código-fonte ou metadados que podem ser interpretados e reproduzidos para simular um usuário. Além disso, elas permitem adicionar pontos de verificação manualmente durante ou depois da gravação por meio da edição do código-fonte gerado. Exemplos de ferramentas são Marathon[4], para testes de GUIs geradas com Java/Swing, e Selenium-IDE[5], para WUIs.

A principal vantagem da gravação de testes é a facilidade de sua escrita, já que não é necessário conhecimento de programação. Assim, qualquer pessoa pode criar casos de testes automatizados, mesmo clientes e usuários finais. A produtividade também pode ser alta, já que o trabalho de localização dos componentes da tela é transparente para o usuário.

Contudo, o código gerado pode não ser muito legível e modularizado, o que resulta na replicação de código e difícil manutenção. Qualquer alteração na interface de usuário pode afetar diversos casos de testes. Se muitos testes precisarem ser arrumados ou até mesmo refeitos ou regravados do zero, a produtividade dessa abordagem pode diminuir.

A qualidade dos testes automatizados de interface depende das ferramentas utilizadas, por isso também é interessante conhecer as limitações das ferramentas. Existem casos onde nem todos os eventos de usuários podem ser gravados, sendo necessária a edição do código-fonte para complementar o teste.

Por essas complicações, a abordagem recomendada para evitar esses problemas é refatorar o código-fonte gerado após a gravação da simulação de forma a organizar o código-fonte gerado. Utilizar a gravação pode aumentar a produtividade durante a criação dos testes, assim como é mais produtivo dar manutenção em código-fonte organizado e modularizado.

No entanto, a gravação da simulação só faz sentido após a implementação da interface, portanto esta abordagem é conflitante com as técnicas de escrever os testes antes mesmo da implementação (vide Capítulo 9). Entretanto, nada impede que as duas abordagens sejam mescladas, dependendo das situações.

Já para os sistemas legados, em que o código-fonte está muito embaralhado ou incompreensível, a gravação de testes é uma técnica rápida e interessante para trazer segurança nas futuras manutenções do sistema[6]. Além disso, ela pode ser útil para criação de outros tipos de testes, como os de usabilidade, acessibilidade e aceitação.

Outro uso interessante dos gravadores é a criação de testes de fumaça bem simples e sem a necessidade de refatoração do código-fonte gerado. Como a criação dos testes é rápida e os cenários são simples, o custo de refazê-los do princípio tende a ser menor do que o custo de refatorar o código gerado. Testes de fumaça são, muitas vezes, utilizados em ambientes de produção, desde que não sejam realizadas ações críticas que possam comprometer o sistema e os dados em caso de erro.

8.3 Desempenho dos Testes

Os testes de interface são intrinsecamente mais lentos do que aqueles das outras camadas do sistema. Por mais que eles sejam isolados, podem envolver processos de renderização e de manuseio de dispositivos de hardware para criação dos eventos de usuário. Para os sistemas Web, a situação é ainda mais delicada,

[4]Marathon é uma ferramenta implementada em Python que permite gravar testes para aplicações escritas em Java/Swing.

[5]Selenium-IDE é uma extensão para o navegador Firefox que permite gravar, editar e executar testes escritos com a ferramenta Selenium

[6]O autor deste trabalho aplicou esta recomendação em um sistema Web durante seu estágio na Assembleia Legislativa do Estado de São Paulo.

pois pode ser necessário o apoio de navegadores Web e de servidores para integrar o lado do cliente e o do servidor.

No caso dos testes de integração, o desempenho pode baixar a níveis críticos. A persistência de dados no sistema de arquivos, ou em banco de dados, também é uma operação lenta, podendo até ser um dos gargalos de desempenho, assim como a comunicação lenta entre redes e servidores[7].

O ideal da automação de testes é que todos os cenários possam ser executados a todo momento e praticamente instantaneamente. No caso dos testes de interface, pode haver uma certa tolerância quanto ao desempenho, desde que eles não sejam tão lentos a ponto de atrasar o desenvolvimento do projeto. O mesmo ocorre com outros tipos de testes intrinsecamente demorados, tais como os de aceitação. No caso de grandes baterias de testes de WUI, elas podem demorar até horas para serem executadas.

Existem algumas ferramentas que podem acelerar a execução dos testes ou recomendações para tornar o uso da automação mais prática. A ferramenta Selenium Grid, por exemplo, cria uma grade computacional para possibilitar que os testes escritos com a ferramenta Selenium sejam executados paralelamente em diversos ambientes. Essa ferramenta é especialmente útil quando uma mesma bateria de testes é executada em vários navegadores Web para testar a portabilidade da aplicação. Contudo, para que ela seja utilizada com êxito, é fundamental que os casos de testes sejam completamente independentes uns dos outros e da ordem em que são executados. Como mais de um caso de teste pode ser executado ao mesmo tempo, um poderá influenciar o sucesso de outro, caso haja algum recurso compartilhado.

Os erros causados pela dependência dos testes inviabilizam a execução paralela da bateria de testes. A quantidade de erros que podem surgir é indeterminada além de que os erros possivelmente serão intermitentes. Mesmo os relatórios dos resultados dos testes não serão mais úteis porque a bateria de testes perde a sua propriedade de reprodutibilidade.

Outra recomendação é utilizar as ferramentas com melhor desempenho durante o período de desenvolvimento e deixar as mais lentas para serem executadas em ambientes de integração contínua, pelo menos uma vez ao dia. Para interfaces Web, existem ferramentas que permitem executar a mesma bateria de testes em diferentes navegadores. Dessa forma, durante o dia a dia de desenvolvimento, os testes podem ser executados no navegador do HtmlUnit, que é leve e tem melhor desempenho porque não renderiza a interface. Já no ambiente de integração contínua, os testes podem ser executados de forma assíncrona nos navegadores reais, como Firefox e Google Chrome.

[7] O autor teve essa experiência enquanto trabalhava com o sistema Janus, da Pró-Reitoria de Pós-Graduação da USP. O sistema possuia uma bateria de testes de integração e uma de testes de fumaça para a sua interface Web. Os testes de integração eram executados em um ambiente de integração contínua, que era um servidor Linux com uma sessão gráfica virtual

8.4 Padrões

A seguir serão descritos padrões de automação para testes que envolvem a interface de usuário. Estes padrões foram identificados pelo autor em projetos próprios e com o estudo de certas ferramentas de testes que já implementam algumas das soluções propostas. Os padrões serão descritos segundo o esqueleto exibido na Seção 5.3. Quando houver diferença significativa na implementação do padrão para interfaces GUI e WUI, haverá subtópicos próprios para descrever cada caso.

8.4.1 Tela como Unidade

Tipo: Organizacional

Quando utilizar: Idealmente, em todos os testes de interface de usuário.

Intenção: Facilitar a implementação dos métodos de *set up*, tornar os testes menos sensíveis a mudanças do sistema e melhorar o desempenho deles.

Motivação: Para o teste chegar até a tela a ser testada pode ser necessária a navegação por meio de diversas outras telas do sistema, o que acaba adicionando responsabilidades adicionais ao teste desnecessariamente. Esta navegação normalmente deve ficar armazenada no método de *set up*, pois é uma preparação do ambiente para realização dos testes.

A navegação torna a criação do teste mais complexa e gera uma dependência de outras telas do sistema. Além disso, o carregamento de outras telas pode armazenar na memória informações que não estão explícitas na descrição do teste, o que, se ocorrer, dificulta o entendimento dos casos de teste. Por último, este processo exige que outras telas que não são pertinentes para os testes sejam carregadas e renderizadas, o que demanda tempo.

Solução: Abrir a tela diretamente, dispensando a necessidade de navegar por outras telas do sistema.

Consequências: A navegação entre telas é evitada, o que torna o caso de teste mais rápido e independente de outras partes do sistema, assim como é esperado com os testes de unidade.

Implementação - GUI: Deve-se instanciar os objetos que representam a tela diretamente. Se não for possível abrir a tela dessa maneira, pode-se adaptar o código do sistema, ou, então, o objeto pode ser encapsulado no método de *set up* em um outro objeto que permita a abertura da tela. Algumas ferramentas de testes fornecem suporte para isso.

Implementação - WUI: A página deve ser aberta diretamente através da URL, por mais que o endereço seja longo. Só é necessário cautela para que o domínio da aplicação esteja definido em apenas um local da aplicação, o que evita a replicação de código e facilita a manutenção dos casos de teste.

Exemplo - GUI - Java/Swing/JUnit/Fest: Temos uma tela de configurações em teste, exibida na Figura 8.3 (fotografia de uma tela do sistema GinLab). O primeiro passo é abrir a tela para que ela seja manipulada pelos testes. Essa tela é aberta da mesma forma que um usuário do sistema, utilizando *links* ou atalhos de teclado presentes em outra tela do sistema, como é mostrado na Figura 8.4.

Neste caso, o *set up* dos testes precisa descrever todos os passos que o usuário faria, como é mostrado na Figura 8.5. Note que é necessário conhecer alguns detalhes de implementação e do funcionamento da tela principal (representada pelo objeto `TelaPrincipalFrame`) do sistema para poder fazer os testes da tela de configuração. Isso torna o teste menos robusto, mais complexo, lento e ilegível.

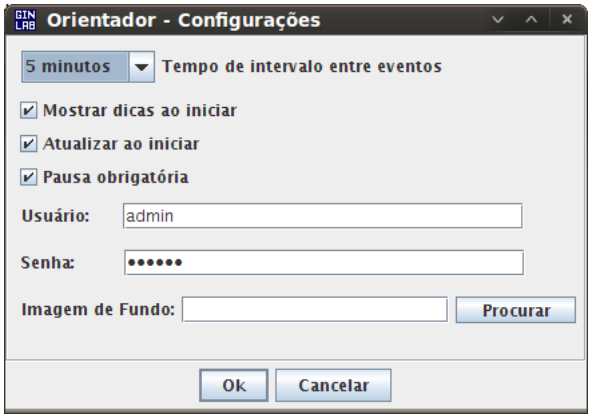


Figura 8.3: Tela de configurações a ser testada.

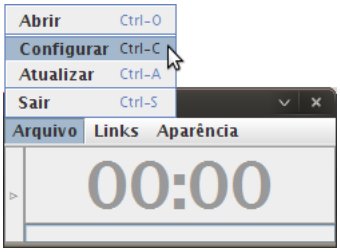


Figura 8.4: Tela principal do sistema que contém *links* e atalhos de teclado para abrir a tela de configurações a ser testada.

Contudo, é possível quebrar esta dependência se conseguirmos instanciar isoladamente a janela que será testada (representada pelo objeto `TelaConfiguracoesFrame`), como é apresentado na Figura 8.6. É pertinente notar que os testes do menu e dos atalhos são importantes de serem realizados, mas eles devem estar agrupados com os testes da janela que os contém, ou seja, com os testes da tela principal.

Exemplo em Java - WUI/WebDriver: Para testar sistemas Web, podemos acessar sua página inicial e então navegar até o módulo desejado. A Figura 8.7 segue esta estratégia para acessar uma página de autenticação da aplicação. Note que este é um exemplo simples, onde é necessário passar por apenas uma página para encontrar um *link* para o local que queremos testar. Em outras situações, pode ser necessário navegar por diversas outras páginas, deixando o código dos testes ainda mais complexo e lento.

Para sistemas Web, é mais simples acessar as páginas diretamente, já que, normalmente, o endereço completo da página desejada pode ser aberto diretamente, como é mostrado na Figura 8.8. Quando não for possível acessar o módulo para teste diretamente, deve ser utilizado o endereço da página mais próxima para que se possa navegar até o local desejado. Isso, geralmente, é necessário quando for preciso carregar algumas informações por intermédio da submissão de formulários Web antes de acessar a página que será testada.

Padrões Relacionados: O padrão Estado Inicial da Tela (Seção 8.4.2) facilita a implementação desse padrão.

Usos Conhecidos: Os testes dos sistemas Janus e GinLab seguem essa organização.

```java
// junit
import org.junit.Before;
// FEST
import org.fest.swing.fixture.FrameFixture;
// Código do pacote AWT utilizado pelo FEST
import java.awt.event.KeyEvent;

public class UmTest {
  private FrameFixture window;

  @Before
  public void abreTelaEmTeste() {
    window = new FrameFixture(new TelaPrincipalFrame());
    window.show();
    window.focus();
    window.menuItem("arquivo").click();
    window.menuItem("configurar").click();

    // ou poderia abrir a janela por atalhos de teclado:
    // window.pressKey(KeyEvent.VK_CONTROL);
    // window.pressKey(KeyEvent.VK_C);
    // window.releaseKey(KeyEvent.VK_C);
    // window.releaseKey(KeyEvent.VK_CONTROL);

    // tela de configurações aberta
  }
  // casos de testes aqui...
}
```

Figura 8.5: Exemplo de teste com a ferramenta Fest de uma tela sem o padrão Tela como Unidade.

```java
// junit
import org.junit.Before;
// FEST
import org.fest.swing.fixture.FrameFixture;

public class UmTest {
  private FrameFixture window;

  @Before
  public void abreTelaEmTeste() {
    window = new FrameFixture(new TelaConfiguracoesFrame());
    window.show();
    window.focus();
    // tela de configurações aberta
  }
  // casos de testes aqui...
}
```

Figura 8.6: Refatoração do exemplo da Figura 8.5 para utilizar o padrão Tela como Unidade.

```
// referências do JUnit
import org.junit.Before;
// referências do WebDriver
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;

public class LoginTests {
  public static final HOST = "http://localhost:8000";

  @Before
  public void abreTelaDeLogin() {
    // Cria um cliente (navegador)
    WebDriver driver = new HtmlUnitDriver();
    // Abre uma página armazenada em um servidor local
    driver.get(HOST);
    // Procura o link para página de autenticação
    WebElement login_link = driver.findElement(By.id("login"));
    login_link.click();
    // tela de login aberta
  }

  // casos de testes aqui
}
```

Figura 8.7: Exemplo de teste de uma página Web de autenticação sem o padrão Tela como Unidade.

```
// referências do JUnit
import org.junit.Before;
// referências do WebDriver
import org.openqa.selenium.WebDriver;

public class LoginTests {
  public static final HOST = "http://localhost:8000";

  @Before
  public void abreTelaDeLogin() {
    // Cria um cliente (navegador)
    WebDriver driver = new HtmlUnitDriver();
    // Abre uma página de login diretamente
    driver.get(HOST + "/login");
    // tela de login aberta
  }

  // casos de testes aqui
}
```

Figura 8.8: Refatoração do exemplo da Figura 8.7 para utilizar o padrão Tela como Unidade.

8.4.2 Estado Inicial da Tela

Tipo: Organizacional

Quando utilizar: Idealmente, em todos os testes de interface de usuário. Se muitos casos de testes necessitarem de um mesmo estado diferente do inicial, este padrão ainda deverá ser usado, mas o *set up* nessa situação necessita ter um código adicional que alterará o estado da interface para o desejado.

Intenção: Facilitar o entendimento do código-fonte e a escrita de testes independentes.

Motivação: A tela a ser testada geralmente é aberta no método de *set up*, no entanto, uma mesma tela da interface de usuário pode ter vários estados; por exemplo, com dados carregados ou com configurações específicas. Se a cada vez que o *set up* é executado, a interface é iniciada em um estado diferente devido a informações previamente carregadas, então não fica claro para os testes qual o estado da interface que está sendo testada. Isso ajuda a tornar os testes intermitentes e rebuscados.

Solução: Padronizar que o método de *set up* sempre irá carregar o estado inicial da tela em teste, ou seja, todas as informações que podem influenciar o estado da tela devem ser apagados. Se for necessário fazer alguma modificação no estado inicial para realização de algum cenário, essa modificação deverá ser feita nos respectivos métodos de teste. Se muitos casos de testes precisarem de um mesmo estado diferente do inicial, então os testes podem ser agrupados em outra classe que definirá o estado desejado como estado inicial da tela.

Consequências: O método de *set up* se torna mais consistente, já que a tela sempre será a mesma e com o mesmo estado após a sua execução. Isso torna os testes mais fáceis de serem entendidos, pois é evitada a adição de estruturas de controle de fluxo para lidar com cada estado da tela. Ainda, quando o estado precisar ser modificado para realização de um teste, o trecho de código fica explícito no método de teste, o que facilita o entendimento do caso de teste específico e não prejudica a legibilidade dos outros casos.

Implementação: O padrão **Tela como Unidade** pode ser o suficiente para carregar uma tela no seu estado inicial, pois evita que dados sejam lidos e processados durante a navegação por outras telas do sistema em teste. Se não for suficiente, devem ser desmantelados os dados já carregados no método de *set up* antes mesmo de exibir a tela.

Quando muitos testes possuírem comportamento inicial repetido, devem ser feitas duas refatorações: (1) a configuração do estado pode ser movida para um método separado que será chamado por cada caso de teste; ou (2) pode ser criada uma nova classe específica para estes casos de testes que podem compartilhar um método de *set up* idêntico.

Implementação - WUI: No caso de sistemas Web é necessário atenção com os dados carregados no escopo da aplicação e na sessão de usuário, pois podem influenciar o estado das telas. Ainda, se for necessário navegar entre algumas páginas, evite submeter formulários desnecessários para evitar o carregamento de novas informações.

O carregamento implícito de informações não pertinentes aos casos de testes pode produzir falsos resultados positivos ou negativos, principalmente quando as verificações são pouco precisas; por exemplo, quando é verificado se uma determinada palavra aparece em qualquer local da tela.

Exemplo: A Figura 8.9 mostra uma organização comum de testes de interface, mas que não se preocupa com o estado da tela no momento do teste. Já a Figura 8.10 apresenta soluções simples que garantem que no início de cada teste o estado da tela deverá ser sempre o mesmo.

```java
// referências do JUnit
import org.junit.BeforeClass;
// referências do WebDriver
import org.openqa.selenium.WebDriver;

public class MapaDoSiteTests {
  public static final HOST = "http://localhost:8000";

  // A página em teste é acessada apenas uma vez.
  // Todos os testes irão manipular a interface, então não é claro
  // qual o estado inicial da interface no início de cada teste.
  @BeforeClass
  public void acessaPaginaDoMapaDoSite() {
    WebDriver driver = new HtmlUnitDriver();
    // Acessou uma página diferente da que está sendo testada.
    // Informações adicionais e irrelevantes para os testes
    // podem ter sido carregadas desnecessariamente.
    driver.get(HOST);
    // Acessa a página que será testada.
    WebElement login_link = driver.findElement(By.id("login"));
    login_link.click();
  }

  // Testes...
}
```

Figura 8.9: Exemplo de organização sem o padrão Estado Inicial da Tela.

```java
// referências do JUnit
import org.junit.BeforeClass;
// referências do WebDriver
import org.openqa.selenium.WebDriver;

public class MapaDoSiteTests {
  public static final HOST = "http://localhost:8000";

  // Página é acessa antes de cada teste, garantindo que todas as
  // manipulações (sem estado) feitas por testes anteriores
  // serão descartadas.
  @Before
  public void acessaPaginaDoMapaDoSite() {
    WebDriver driver = new HtmlUnitDriver();
    // Página é acessada diretamente.
    driver.get(HOST + "/mapadosite");

    // Uma maneira radical de forçar que uma nova sessão seja utilizada
    // para cada um dos testes.
    driver.manage().deleteAllCookies();
  }

  // Testes ...
}
```

Figura 8.10: Exemplo de organização com o padrão Estado Inicial da Tela.

Padrões Relacionados: O padrão **Tela como Unidade** (Seção 8.4.1) ajuda a garantir que o mínimo de informações necessárias serão carregadas para o teste, facilitando a implementação desse padrão.

Usos Conhecidos: Os testes dos sistemas Janus e GinLab seguem esta organização.

8.4.3 Camada de Abstração de Funcionalidades

Tipo: Organizacional

Quando utilizar: Idealmente, para todos os testes de interface de usuário. A camada de abstração de funcionalidades, descrita no padrão, pode ser compartilhada pelos testes de correção da interface, leiaute, usabilidade e acessibilidade.

Intenção: Separar o código da manipulação da interface daquele que descreve os cenários de testes para facilitar a escrita e o entendimento dos testes.

Motivação: Muitos casos de testes da interface são descritos por meio de verificações realizadas após a execução de uma ou mais funcionalidades do sistema. No entanto, a chamada de uma funcionalidade do sistema via interface gráfica é, geralmente, representada por diversos comandos de um usuário. Por isso, o código-fonte que contém os cenários de testes de interface tende a ficar extenso e pouco legível.

Além disso, muitos cenários de testes utilizam o mesmo conjunto de ações que manipulam a interface, mas com pequenas diferenças. Portanto, é natural a separação destas ações para reutilização de código-fonte.

Solução: Separar os casos de testes em duas camadas: uma de abstração de funcionalidades, que irá manipular a interface como um usuário, e outra, que irá fazer as chamadas das funcionalidades e as verificações necessárias na interface.

Consequências: A abstração das funcionalidades diminui a extensão do código dos testes e melhora a legibilidade. Esta modularização também permite a melhor reutilização do código de acesso à interface, podendo até ser utilizada por baterias de tipos de testes diferentes.

Implementação: O teste deve conter apenas chamadas às funcionalidades da interface e verificações. Todo conjunto de ações de usuário que compõe uma funcionalidade deve ser encapsulado em um método separado de uma classe independente dos casos de teste, ou seja, uma classe que pertence à camada de abstração de funcionalidades. A Figura 8.11 mostra como fica a organização do código-fonte dos testes utilizando esse padrão.

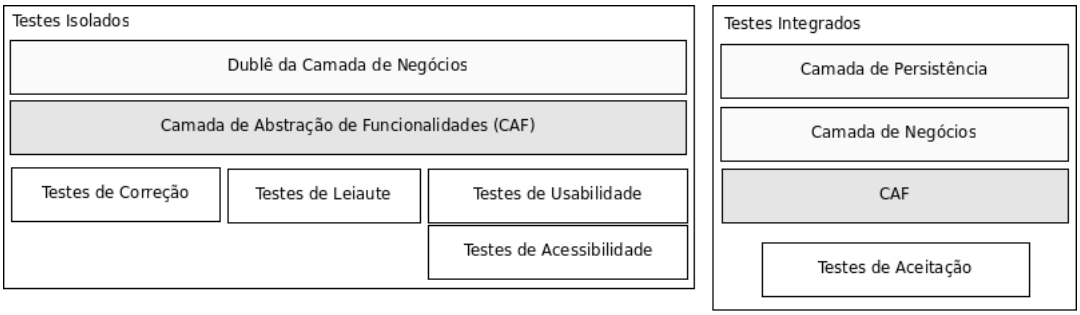


Figura 8.11: Organização recomendada de testes de interface de usuário.

Exemplo - WUI - Java/JUnit/WebDriver: A Figura 8.12 possui um exemplo de teste de uma tela de autenticação de um sistema Web sem utilizar o padrão **Camada de Abstração de Funcionalidades**. Note que os testes são relativamente extensos para simplicidade do que se pretende verificar. Isso se deve às diversas ações necessárias para se autenticar no sistema.

Esses testes podem ser separados em duas camadas: a **Camada de Abstração de Funcionalidades** como é mostrado na Figura 8.13 e a de verificações, que contém a descrição legível dos

```java
// referências do JUnit
import org.junit.Test;
import static org.junit.Assert.*;
// referências do WebDriver
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;

public class LoginTests {
  public static final HOST = "http://localhost:8000";
  // Cria uma instância do navegador
  WebDriver driver = new HtmlUnitDriver();

  @Test
  public void testaLoginValidoRedirecionaParaPaginaInicial() {
    // Abre página de login
    driver.get(HOST + "/login");
    // Encontra objetos na tela
    WebElement usuario = driver.findElement(By.id("usuario"));
    WebElement senha = driver.findElement(By.id("senha"));
    WebElement login = driver.findElement(By.id("login"));
    // Simula ações do usuário
    usuario.sendKeys("admin");
    senha.sendKeys("1234");
    login.click();
    // Faz verificações
    assertTrue(driver.getCurrentUrl().endsWith("/home"));
    assertTrue(driver.getPageSource().contains("Olá admin!"));
  }

  @Test
  public void testaLoginInvalidoMostraMensagemDeErroNaMesmaTela() {
    // Abre página de login
    driver.get(HOST + "/login");
    // Encontra objetos na tela
    WebElement usuario = driver.findElement(By.id("usuario"));
    WebElement senha = driver.findElement(By.id("senha"));
    WebElement login = driver.findElement(By.id("login"));
    // Simula ações do usuário
    usuario.sendKeys("admin");
    senha.sendKeys("senha_errada");
    login.click();
    // Faz verificações
    assertTrue(driver.getCurrentUrl().endsWith("/login"));
    assertTrue(driver.getPageSource().contains("Login inválido."));
  }

}
```

Figura 8.12: Exemplo de testes de uma página Web de autenticação sem utilizar o padrão **Camada de Abstração de Funcionalidades**.

testes, como é mostrado na Figura 8.14. Note que a camada de verificações possui a mesma estrutura do exemplo original e apenas substitui os comandos de usuário por chamadas à camada de abstração das funcionalidades.

Já a camada de abstração de funcionalidades deve apenas possuir métodos ou funções que encapsulem um conjunto de ações que simulem a chamada de uma funcionalidade por um usuário do sistema. Dessa forma, essa camada deve ser independente dos casos de teste; ou seja, os testes devem ter conhecimento da camada de abstração, mas não o contrário. Além disso, a camada não deve possuir verificações, pois não faz parte de seu objetivo testar o sistema.

```
1  // É interessante separar a camada de abstração das funcionalidades em um local
       diferente
2  package funcionalidadesUI;
3
4  // referências do WebDriver
5  import org.openqa.selenium.By;
6  import org.openqa.selenium.WebDriver;
7  import org.openqa.selenium.WebElement;
8  // Note que não há dependências do arcabouço de teste porque não são feitas
       verificações nesta camada
9
10 public class Sistema {
11   WebDriver driver;
12   String host;
13
14   public Sistema(WebDriver driver, String host) {
15     this.driver = driver;
16     this.host = host;
17   }
18
19   public void login(String usuario, String senha) {
20     // Abre página de login
21     driver.get(host + "/login");
22     // Encontra objetos na tela
23     WebElement username = driver.findElement(By.id("username"));
24     WebElement password = driver.findElement(By.id("password"));
25     WebElement login = driver.findElement(By.id("login"));
26     // Simula ações do usuário
27     username.sendKeys(usuario);
28     password.sendKeys(senha);
29     login.click();
30   }
31 }
```

Figura 8.13: Refatoração do exemplo da Figura 8.12. Essa classe faz parte da **Camada de Abstração de Funcionalidades**.

Ainda, a camada de abstração pode ser parametrizável para que ela seja adaptável a vários casos de teste sem replicação de código. Também podem ser criados métodos como *açúcar sintático* para as combinações de parâmetros mais comuns, o que facilita a escrita dos testes, assim como melhora a sua legibilidade. A Figura 8.15 possui açúcares sintáticos comuns que podem ser usados em diversos casos de testes e em diferentes baterias de testes.

Padrões Relacionados: A inicialização das interfaces pode ser incluída na camada de abstração de funcionalidades, dessa forma, essa organização pode ser integrada com as recomendações do padrão **Tela como Unidade** (Seção 8.4.1).

```java
// referências do JUnit
import org.junit.Test;
import static org.junit.Assert.*;
// referências do WebDriver
import org.openqa.selenium.WebDriver;

public class LoginTests {
  public static final HOST = "http://localhost:8000";
  WebDriver driver = new HtmlUnitDriver();
  Sistema sistema = new Sistema(driver, HOST);

  @Test
  public void testaLoginValidoRedirecionaParaPaginaInicial() {
    sistema.login("admin", "123456");
    // Faz verificações
    assertTrue(driver.getCurrentUrl().endsWith("/home"));
    assertTrue(driver.getPageSource().contains("Olá admin!"));
  }

  @Test
  public void testaLoginInvalidoMostraMensagemDeErroNaMesmaTela() {
    sistema.login("admin", "senha_errada");
    // Faz verificações
    assertTrue(driver.getCurrentUrl().endsWith("/login"));
    assertTrue(driver.getPageSource().contains("Login inválido."));
  }

}
```

Figura 8.14: Continuação da refatoração do exemplo da Figura 8.12. Camada que contém as verificações utilizando a **Camada de Abstração de Funcionalidades**.

```java
// ...
public class Sistema {

  public void login(String usuario, String senha) {
    // ...
  }

  public void logaComoAdministrador() {
    login("admin", "123456");
  }

  public void logaComoUsuarioComum() {
    login("user", "1234");
  }
}
```

Figura 8.15: Açúcares sintáticos para melhorar ainda mais a legibilidade dos testes.

Usos Conhecidos: O sistema Janus organiza seus testes de interface utilizando esse padrão. A documentação da ferramenta WebDriver descreve o padrão Objeto Página (*Page Object*)[8], que descreve como devem ser implementados os objetos da Camada de Abstração de Funcionalidades.

[8] `http://code.google.com/p/selenium/wiki/PageObjects`.

8.4.4 Fotografia da Interface

Tipo: Organizacional

Quando utilizar: Idealmente, em todos os casos de teste de interface gráfica de usuário.

Intenção: Facilitar a análise dos testes que falham por meio de uma fotografia da interface que é capturada no momento em que o erro ocorre.

Motivação: Testes de interface podem ter resultados falso-negativos por muitas razões, como devido a alterações do leiaute, erros de implementação do teste ou, até mesmo, por problemas de infraestrutura, para o caso de testes integrados. Por causa disso, pode ser demorado identificar o motivo das falhas do teste, principalmente quando é necessário depurar o sistema. A fotografia serve de complemento às outras informações armazenadas pelo arcabouço de teste, como os valores de variáveis.

Solução: Capturar uma fotografia da interface toda vez que um teste falhar, a qual deverá ser o ponto de partida para análise do erro.

Consequências: A fotografia pode facilitar o entendimento do que aconteceu de errado ou até mesmo elucidar imediatamente a causa do erro.

Implementação: A implementação desse padrão é complexa, pois pode depender de detalhes do sistema operacional, das bibliotecas de interface ou dos navegadores. Por isso, só é recomendável utilizar esse padrão com o auxílio de alguma ferramenta que disponibilize essa funcionalidade. Contudo, algumas ferramentas de teste de interface já fornecem em sua biblioteca padrão de funcionalidades esse tipo de comando, como é o caso da ferramenta Selenium.

Exemplo - WUI Java/Selenium/Util4Selenium: A ferramenta Util4Selenium disponibiliza um aspecto que facilita a implementação desse padrão de modo que é reduzido drasticamente a replicação de código-fonte. A Figura 8.16 mostra um trecho da implementação desse aspecto. Todos os métodos que forem anotados com o metadado `@Screenshot` receberão um código que automaticamente capturará uma fotografia da interface Web caso uma exceção seja lançada. Quando uma classe possuir essa informação, então o mesmo código é adicionado a todos os seus métodos.

Um bom modo de utilizar essa ferramenta é com a criação de uma classe base (Figura 8.17) que deverá ser herdada por todas as classes de testes de interface do sistema (Figura 8.18). Como ela possui a anotação `@Screenshot`, então o padrão Fotografia da Interface será automaticamente propagado para todos os casos de teste do sistema.

Padrões Relacionados: Esse padrão também pode ser utilizado na **Camada de Abstração de Funcionalidades** (Seção 8.4.3).

Usos Conhecidos: A ferramenta Selenium disponibiliza uma funcionalidade que bate uma fotografia da página renderizada no navegador Web e a Util4Selenium utiliza essa funcionalidade de forma automática, por exemplo, ela pode ser configurada para bater fotos toda vez que um caso de teste falhar, como é mostrado no exemplo anterior.

```
// Referências Java e AspectJ
import org.aspectj.lang.reflect.MethodSignature;
import java.lang.annotation.Annotation;
import org.aspectj.lang.Signature;
// Outras referências foram ocultas

public aspect AspectScreenshotByAnnotation {
  // Qualquer método ou classe anotado com @Screenshot
  pointcut annotationHandler():
    if(System.getProperty("selenium.screenshot") != null &&
      System.getProperty("selenium.screenshot").equals("true")) &&

    // Método
    // @Screenshot (void|objeto) umMetodo(argumentos): public,private..
    (execution(@utilities.util4selenium.annotations.Screenshot * *(..)) ||

    // Classe
    // (public/private..) (@Screenshot UmaClasse).umMetodo(argumentos)
    (execution(* (@utilities.util4selenium.annotations.Screenshot *).*(..)) &&
    !execution(public * selenium())));

  after() throwing(): annotationHandler() {
    Signature signature = thisJoinPoint.getSignature();
    MethodSignature methodSignature = (MethodSignature) signature;
    Annotation annotation = methodSignature.getMethod().getAnnotation(Screenshot.
        class);
    if(annotation == null) {
      annotation = signature.getDeclaringType().getAnnotation(Screenshot.class);
    }
    if(annotation != null && thisJoinPoint.getThis() instanceof SeleniumClass) {
      SeleniumClass obj = (SeleniumClass) thisJoinPoint.getThis();
      ScreenshotHelper helper = new ScreenshotHelper(obj.selenium());
      String clazz = signature.getDeclaringType().getSimpleName();
      String method = signature.getName();
      helper.screenshot(clazz + "-" + method);
    }
  }
}
```

Figura 8.16: Aspecto fornecido pela biblioteca Util4Selenium para bater fotografias da interface.

```java
// Classe base pra todas clases de teste da interface Web

// Referências do TestNG
import org.testng.annotations.AfterSuite;
import org.testng.annotations.BeforeSuite;
// Referências do Util4Selenium
import testutilities.util4selenium.annotations.Screenshot;
import testutilities.util4selenium.annotations.Screenshot.ScreenshotPolicy;
// Referências do Selenium-RC/Java
import com.thoughtworks.selenium.DefaultSelenium;
import com.thoughtworks.selenium.Selenium;

// Padrão Fotografia da Interface
@Screenshot(policy = ScreenshotPolicy.ON_ERROR)
public class SeleniumTestCase {
  public Selenium navegador;

  @BeforeSuite public void abreNavegador() {
    navegador = new DefaultSelenium(
        "localhost", 4444,
        "*chrome", "http://localhost:8000");
    navegador.start();
  }

  @AfterSuite public void fechaNavegador() {
    navegador.stop();
  }
}
```

Figura 8.17: Classe base que ativa o padrão Fotografia da Interface.

```java
// Referências do JUnit + Hamcrest
import org.junit.*;
import static org.junit.Assert.*;
import static org.hamcrest.Matchers.*;
// Referências do Selenium-RC/Java
import org.testng.annotations.Test;

public class MapaDoSiteTests extends SeleniumTestCase {

  @Before public void acessaPaginaEmTeste() {
    navegador.open("/mapadosite");
  }

  @Test public void verificaLinksImportantes() {
    // Se falhar, será capturada uma fotografia da interface e
    // salva no arquivo MapaDoSiteTests-verificaLinksImportantes.png
  }
}
```

Figura 8.18: Exemplo de classe de teste que utiliza a classe base `SeleniumTestCase`.

8.4.5 Localizar Elemento por ID

Tipo: Robustez

Quando utilizar: Em todos os testes, principalmente para localização dos principais componentes da interface.

Intenção: Tornar a localização de componentes na interface independente do leiaute, do estado dos componentes e da internacionalização (i18n) do sistema.

Motivação: A localização dos componentes da interface é uma tarefa delicada. Os testes não devem ser frágeis a tal ponto de quebrarem por qualquer alteração do leiaute.

Solução: Atribuir IDs aos principais componentes da interface de usuário para facilitar a localização dos mesmos. Apenas é necessário cuidado para que mais de um componente não possua ID idêntico, já que uma mesma tela pode conter diversos painéis que possuem campos semelhantes. Uma forma de organização que evita que isso aconteça é concatenar os nomes dos painéis ou das telas como prefixo do ID do componente.

Consequências: O código de localização dos elementos pode ficar mais claro, pois não é necessário o uso de expressões de busca como DOM e XPath. Além disso, o identificador tem como premissa ser uma propriedade única e exclusiva de um elemento, portanto facilita a identificação dos componentes.

Implementação: Os componentes que serão manipulados precisam ter IDs definidos pelos programadores. Apesar disso, algumas ferramentas de testes permitem que se defina os IDs em tempo de execução do teste. Neste caso, é recomendado separar a definição dos IDs em um local isolado para não prejudicar a legibilidade dos testes. Um bom local para definição dos IDs é na **Camada de Abstração de Funcionalidades**.

É importante notar que existem arcabouços de interface de usuário que geram IDs dinamicamente para os componentes, contudo esses IDs podem prejudicar a legibilidade dos testes. Para que não sejam gerados identificadores repetidos, os respectivos algoritmos concatenam caracteres adicionais e não intuitivos que podem criar ambiguidade no entendimento dos cenários de teste.

Exemplo: A Figura 8.19 mostra um exemplo do padrão com a ferramenta WebDriver e HTMLUnit.

```
1 // Cria um cliente (navegador)
2 WebDriver driver = new HtmlUnitDriver();
3 // Procura um link pelo ID
4 WebElement link = driver.findElement(By.id("mapadosite"));
```

Figura 8.19: Exemplo de localização de um elemento por ID com WebDriver e HTMLUnit.

Padrões Relacionados: O padrão **Localizar Elemento por Tipo do Componente** (Seção 8.4.6) serve de alternativa para quando esse padrão não puder ser utilizado.

Usos Conhecidos: É um padrão das ferramentas de teste de interface fornecer mecanismos para localizar elementos por um identificador.

8.4.6 Localizar Elemento por Tipo do Componente

Tipo: Robustez

Quando utilizar: Quando o componente a ser localizado não possui um ID, ou, então, quando o tipo do componente utilizado é importante para o entendimento do teste.

Intenção: Tornar explícito o tipo de componente no código do teste para facilitar a compreensão.

Motivação: Em algumas situações, é mais fácil de entender um caso de teste quando sabemos quais os tipos dos componentes que estão sendo manipulados. Quando utilizamos apenas o ID, o tipo do componente é abstraído.

Solução: Localizar um componente pelo seu tipo. No entanto, quando a tela possuir mais de um componente do mesmo tipo, outras propriedades devem ser usadas para filtrar apenas os elementos desejados. Nesses casos, dê preferência para propriedades que não estejam relacionadas com o leiaute da tela.

Consequências: O código de localização dos elementos tende a ficar mais extenso e mais preso aos tipos dos componentes utilizados. Contudo, o teste pode ficar mais fácil de ser entendido.

Implementação: Algumas ferramentas disponibilizam métodos genéricos, que buscam qualquer tipo de componente. Já outras possuem métodos que buscam um tipo de componente específico. Para esse último, caso pode-se utilizar apenas o identificador do componente.

Implementação - WUI: Geralmente, as ferramentas permitem localizar elementos por expressões XPath. Para explicitar que o componente é uma caixa de texto, poderia-se utilizar a seguinte expressão: `input[type="text", id="id_do_componente"]`. Este tipo de expressão pode encontrar mais de um elemento, por isso a necessidade do ID. Ainda há a possibilidade de utilizar outras propriedades, mas devem ser evitadas, pois isso poderia tornar o teste frágil em relação a alterações do leiaute.

Exemplo: A Figura 8.20 mostra um exemplo com a ferramenta WebDriver e HTMLUnit para buscar um elemento na árvore HTML. A busca é feita por meio de uma expressão XPath, que é muito flexível e aceita tanto como parâmetros de leiaute como de propriedades do componente. Contudo, a expressão contém apenas informações referentes ao tipo de componente e de propriedades que são únicas do elemento.

```
1 // Cria um cliente (navegador)
2 WebDriver driver = new HtmlUnitDriver();
3 // Procura um link pelo seu tipo e por suas propriedades, com XPath.
4 WebElement link = driver.findElement(By.xpath("//a[@href='/mapadosite']"));
```

Figura 8.20: Exemplo de localização de um elemento pelo tipo com WebDriver e HTMLUnit.

Padrões Relacionados: Estratégia alternativa para o padrão **Localizar Elemento por ID** (Seção 8.4.5).

Usos Conhecidos: É um padrão das ferramentas de testes de interface Web, tais como Selenium e HTMLUnit, fornecer mecanismos para localizar os elementos através de XPath.

8.4.7 Localizar Célula de Tabela pelo Cabeçalho e Conteúdo

Tipo: Robustez

Quando utilizar: Em testes que fazem verificações em células específicas de tabelas.

Intenção: Tornar a localização de células de tabelas menos frágil em relação a alterações de leiaute ou a mudança dos dados.

Motivação: Muitas vezes é necessário verificar o conteúdo ou obter componentes de células específicas de tabelas. Um modo de acessar células específicas é a partir dos índices da linha e coluna (Figura 8.21), mas essa abordagem tornam os testes muito frágeis. Qualquer alteração do leiaute ou novo registro carregado pode quebrar os testes.

```
1  // Referências do HTMLUnit
2  import com.gargoylesoftware.htmlunit.html.HtmlPage;
3  import com.gargoylesoftware.htmlunit.html.HtmlTable;
4  import com.gargoylesoftware.htmlunit.html.HtmlTableCell;
5
6  public class HTMLTableHelper {
7    public HtmlTableCell buscaCelulaPeloLayout(HtmlPage pagina, String idTabela, int
         linha, int coluna) {
8      HtmlTable tabela = pagina.getHtmlElementById(idTabela);
9      return tabela.getCellAt(linha, coluna);
10   }
11 }
```

Figura 8.21: Busca da célula de uma tabela pelo leiaute.

Solução: Identificar o índice da célula por meio de um algoritmo que percorra todas as linhas de uma coluna com determinado cabeçalho até que o conteúdo da célula corrente seja o desejado. Quando os dados da tabela estão organizados por colunas em vez de linhas, o algoritmo deve seguir a mesma ideia, mas percorrendo todas as colunas de uma linha.

Consequências: Apesar de os testes perderem desempenho, eles ficam mais resistentes a alterações do leiaute. A legibilidade do teste também melhora, pois os números dos índices são substituídos por *strings* que são mais intuitivas.

Implementação: Algumas ferramentas já disponibilizam métodos prontos para facilitar a busca de células. Quando é utilizada uma que não possui esta facilidade, é importante criar um método independente do caso de teste para realização dessa tarefa. Dessa forma, o método pode ser re-utilizado para outros casos de teste e até mesmo ser enviado como sugestão para a equipe da ferramenta.

Este método deve receber o identificador da tabela e uma informação da linha e uma da coluna. Supondo que cada linha da tabela representa um registro, a informação da coluna pode ser o ID da célula de cabeçalho, que, geralmente, é a primeira da tabela. Para identificar a linha, pode se passar parte do conteúdo esperado, como textos, componentes ou expressões regulares. Outra abordagem mais orientada a objetos seria criar as estruturas Tabela, Linha, Coluna e Célula. Com isso, o algoritmo pode ficar mais modularizado e flexível.

Algumas tabelas são mais complexas, podendo conter outras tabelas internas ou outras formas de divisão. Se não for possível seguir este padrão completamente, pode ser interessante utilizá-lo em conjunto com índices. Dessa forma, é reduzido o uso de índices das células, tornando o teste menos frágil.

Exemplo: A ferramenta HTMLUnit já fornece diversos métodos para leitura de tabelas que facilitam bastante a criação dos testes, contudo, ainda é possível criar métodos auxiliares para forçar o uso desse padrão. A Figura 8.22 mostra duas funcionalidades que percorrem as células da tabela em busca da célula desejada para manipulação e verificação. Apenas é importante notar que essas funcionalidades nem sempre podem ser utilizadas, pois elas não consideram a complexidade das tabelas e também supõe que as células possuem conteúdo único.

```
1  // Referências do HTMLUnit
2  import com.gargoylesoftware.htmlunit.html.HtmlPage;
3  import com.gargoylesoftware.htmlunit.html.HtmlTable;
4  import com.gargoylesoftware.htmlunit.html.HtmlTableCell;
5  import com.gargoylesoftware.htmlunit.html.HtmlTableRow;
6
7  public class HTMLTableHelper {
8    public HtmlTableCell buscaCelulaPorConteudo(HtmlPage pagina, String idTabela,
         String conteudoLinha) {
9      HtmlTable tabela = pagina.getHtmlElementById(idTabela);
10     for(HtmlTableRow linha: tabela.getRows()) {
11       for(HtmlTableCell celula: linha.getCells()) {
12         if(celula.asText().contains(conteudoLinha))
13           return celula;
14       }
15     }
16     throw new RuntimeException("Célula com conteúdo " + conteudoLinha + " não
           encontrada.");
17   }
18
19   public HtmlTableCell buscaCelulaPorCabecalhoEConteudo(HtmlPage pagina, String
         idTabela, String idCabecalho, String conteudoLinha) {
20     HtmlTable tabela = pagina.getHtmlElementById(idTabela);
21
22     HtmlTableRow linhaCabecalho = tabela.getRow(0);
23     List<HtmlTableCell> celulasCabecalho = linhaCabecalho.getCells();
24     int indiceColunaReferencia = -1;
25     for(HtmlTableCell celula: celulasCabecalho) {
26       indiceColunaReferencia += 1;
27       if(celula.getId().equals(idCabecalho))
28         break;
29     }
30     if(indiceColunaReferencia == -1 || indiceColunaReferencia == celulasCabecalho.
           size())
31       throw new RuntimeException("Cabeçalho de ID " + idCabecalho + " não
             encontrado.");
32
33     for(HtmlTableRow linha: tabela.getRows()) {
34       HtmlTableCell celula = linha.getCell(indiceColunaReferencia);
35       if(celula.asText().contains(conteudoLinha))
36         return celula;
37     }
38     throw new RuntimeException("Célula com conteúdo " + conteudoLinha + " não
           encontrada.");
39   }
40 }
```

Figura 8.22: Exemplo de Localizar Célula pelo Cabeçalho e Conteúdo com o HTMLUnit.

Padrões Relacionados: Os padrões **Localizar Elemento por ID** (Seção 8.4.5) e **Localizar Elemento**

por Tipo do Componente (Seção 8.4.6) devem ser utilizados para implementação desse padrão.

Usos Conhecidos: Sistema GinLab e Janus implementam e utilizam esse padrão. As ferramentas HTMLUnit e Util4Selenium também fornecem essas funcionalidades.

8.5 Antipadrões

Nas subseções seguintes serão descritos antipadrões de automação para testes que envolvem a interface de usuário, seguindo o esqueleto apresentado na Seção 5.4. Os indícios de problemas citados nos padrões são descritos na Seção 5.2.

8.5.1 Navegação Encadeada

Tipo: Organizacional

Contexto: Para testar uma tela pode ser necessário a navegação entre diversas telas do sistema. Pensando em desempenho e praticidade para criação dos testes, pode-se criar um grande teste que vai verificando as telas à medida que elas vão sendo carregadas. Essa prática tende a criar testes muito extensos, complexos e ilegíveis. Isso os torna difíceis de manter e entender, além de que alterações em uma das telas pode atrapalhar os testes de outras.

Indícios de Problemas Relacionados: *Obscure Test*, *Test Code Duplication*, *Erratic Test*, *Fragile Test*, *Frequent Debugging*, *Slow Tests*, *Buggy Tests*, *Developers Not Writing Tests* e *High Test Maintenance Cost* (vide Seção 5.2).

8.5.2 Localizar Componente pelo Leiaute

Tipo: Robustez

Contexto: Dependendo das ferramentas de teste é possível localizar um componente de muitas maneiras. Uma das alternativas é utilizar propriedades como a posição na tela ou, ainda, a ordem de exibição. No entanto, qualquer alteração no leiaute pode quebrar o teste, mesmo que o sistema esteja correto. Como todo teste que falha requer uma avaliação, os testes quebrados, que são falso-negativos, desperdiçam tempo de desenvolvimento com a depuração dos testes e do sistema.

Indícios de Problemas Relacionados: *Obscure Test*, *Fragile Test*, *Frequent Debugging* e *High Test Maintenance Cost* (vide Seção 5.2).

8.5.3 Verificações Rígidas

Tipo: Robustez

Contexto: Existem ferramentas ou ténicas que fazem verificações muito rígidas na interface de usuário. Por exemplo, há ferramentas que fazem análises pixel a pixel para verificar se houveram alterações. Outra abordagem transforma o código da interface em um *hash*, que pode ser comparado com o mesmo propósito. Estas abordagens são muito inflexíveis, pois qualquer refatoração e até identação do código-fonte pode quebrar os casos de teste.

Indícios de Problemas Relacionados: *Fragile Test*, *Developers Not Writing Tests* e *High Test Maintenance Cost* (vide Seção 5.2).

8.6 Conclusões

Para fazer testes de interface é fundamental a utilização de arcabouços de testes especializados. Não obstante, a qualidade dos testes automatizados que simulam usuários dependem da boa abstração e das facilidades que os arcabouços de testes de interface disponibilizam. Também é importante notar que essas ferramentas são geralmente complexas de implementar, por isso muitas ainda possuem limitações.

As boas ferramentas de gravação de interação dos usuários são muito úteis para criação de vários tipos de testes que envolvem a interface de usuário. No entanto, é necessário atenção com o código-fonte por elas gerado para que os testes automatizados não sejam de baixa qualidade, prejudicando e atrasando o desenvolvimento do sistema.

O desempenho dos testes de interface também é importante. É recomendável utilizar ferramentas leves e criar testes completamente independentes para que seja possível executá-los em paralelo. Baterias de testes independentes podem ser executadas em diversos ambientes dinstintos, por exemplo, para buscar erros de portabilidade.

Além disso, há diversos padrões e antipadrões que podem influenciar significativamente na organização e na robustez dos testes automatizados, o que é fundamental para que a automação dos testes seja bem feita e ajude no dia a dia do desenvolvimento de sistemas.

Capítulo 9

Técnicas de Desenvolvimento de Software com Testes Automatizados

Como foi discutido nos capítulos anteriores, as baterias de testes automatizados precisam estar bem implementadas para que o custo-benefício dessa prática seja baixo. Assim como o código-fonte do sistema, o dos testes automatizados também está sujeito a imperfeições que podem trazer malefícios graves para um projeto. Por isso, é importante o conhecimento de boas práticas e padrões que auxiliem na criação de testes automatizados de qualidade.

Existem diversas técnicas de desenvolvimento de software com testes automatizados que influenciam diretamente na qualidade dos testes e do sistema. Essas técnicas são descritas por processos simples e sistemáticos. Basicamente, elas definem a relação dos testes automatizados com o processo de desenvolvimento e propõem um roteiro de quando implementar os casos de teste de correção.

Dentre as técnicas que serão descritas estão TAD, TFD, TDD e BDD, que já foram citadas no Capítulo 3. Entretanto, serão apresentadas as vantagens e as desvantagens de cada técnica, assim como algumas comparações entre elas.

Apesar de as técnicas citadas neste capítulo serem generalizadas para qualquer tipo de teste (vide a NASA que usava o ciclo de TDD para cartões perfurados [84]), será evidenciado apenas os testes automatizados de unidade, com exceção do caso de BDD que incluirá testes de aceitação.

Os testes de correção de interface de usuário também podem ser escritos com as técnicas citadas, mas algumas delas são incompatíveis com certas ferramentas de testes de interface. Por exemplo, TDD e TFD requerem que os testes sejam feitos antes da implementação, o que inviabiliza o uso de gravadores de interação.

9.1 Testes Após a Implementação (TAD)

Testar após a implementação (TAD de *Test After Development*) é a técnica de implementar e executar os testes depois que um ou todos os módulos de um sistema estão finalizados (Figura 9.1). Esse é o modo convencional e natural da abordagem dos testes dinâmicos (vide Seção 3.1), já que para executar testes em tempo de execução é necessário que o sistema ou parte dele esteja implementado.

Quando os testes são implementados após classes ou métodos serem finalizados, TAD pode influenciar significativamente o código-fonte e a arquitetura do sistema devido ao rápido *feedback* dos testes. Já quando os testes são realizados apenas ao término do desenvolvimento, TAD tende a se tornar uma prática de controle e garantia de qualidade para as unidades do sistema.

Todavia, a proposta principal de TAD é fazer verificações no sistema. TAD não só surgiu em conjunto com testes automatizados e arcabouços de teste, como também herdou a característica da abordagem tradicional de testes manuais de software de realizar os testes após a implementação. Não faz

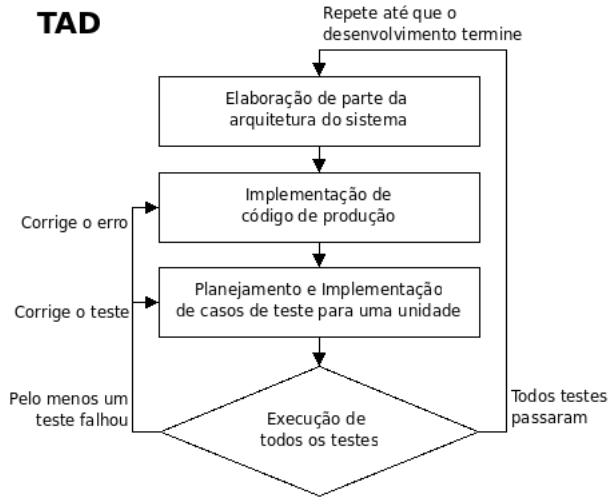


Figura 9.1: Fluxo do TAD.

parte da proposta principal de TAD influenciar na criação de código-fonte do sistema.

Fazer os testes após a implementação é coerente, já que sistemas sem testes podem funcionar corretamente, mas testes sem sistema não fazem sentido; portanto, o ideal é sempre priorizar a implementação do sistema (mesmo nos dias de hoje). Apesar disso, essa abordagem, assim como as outras, não é a mais indicada para todas as situações e projetos. A seguir serão apresentadas as principais características de TAD e quando recomendamos sua utilização.

Características

Quando os testes são implementados após um trecho do sistema ser finalizado, eles precisam se adaptar à estrutura de código já existente. Se a testabilidade do código é alta, então não há grandes problemas durante a automação. Os testes não influenciam no *design* do sistema, mas se tornam úteis para verificação. Entretanto, se a testabilidade do código é baixa, então a automação de testes pode seguir por vários caminhos.

A primeira opção é parar a automação de teste até que o sistema seja refatorado para melhorar a estrutura e aumentar a testabilidade. Nesse caso, a automação não é efetivamente concretizada, mas ela serviu para fornecer *feedback* sobre a modelagem do sistema.

No entanto, se os testes automatizados forem realmente implementados, então pode ser necessário o uso do antipadrão Ganho para os Testes (Seção 6.5.1), que suja o código do sistema, para contornar as dificuldades causadas pelo *design* acoplado. Caso esse antipadrão não seja aplicado, então o código dos testes que se tornarão rebuscados. Como os testes precisam contornar as falhas de testabilidade para conseguir simular os cenários desejados, seu código-fonte provavelmente ficará mais extenso e, consequentemente, com pior legibilidade e clareza.

Por causa disso, a implementação e a manutenção dos testes ficam mais complexas e, portanto, mais cara. Esta queda do custo-benefício da automação dos testes pode resultar na diminuição do escopo das verificações, ou seja, cenários importantes podem deixar de ser realizados e a cobertura dos testes tende a diminuir.

A falsa impressão de o sistema estar finalizado e correto, em conjunto com a dificuldade de criar os casos de teste, pode levar a equipe de desenvolvimento ou gerentes a crer que a automação dos testes é desnecessária. Por isso, é importante lembrar que o alto custo de manutenção dos sistemas sem testes automatizados não se deve apenas à fase de desenvolvimento, mas, principalmente, a médio e longo prazo.

De qualquer maneira, em casos extremos de dificuldade, a automação de testes pode ser realmente desnecessária. Em situações em que a testabilidade é muito baixa, é mais vantajoso fazer testes manuais e integrados que consigam simular os principais casos de testes. Apenas é importante notar que qualquer cenário que diminua as tarefas de verificação do sistema torna propício a identificação de erros em ambientes de produção.

Quando Utilizar

O fato de TAD possuir fases distintas para implementar e testar é uma vantagem para situações específicas, como para a manutenção de sistemas legados. Como a automação de testes ainda é uma prática recente, muitos sistemas legados que precisam de manutenção não possuem baterias de testes automatizados.

Já que a manutenção desses sistemas pode ser uma tarefa crítica, é recomendável que sejam feitos pelo menos algumas baterias de testes automatizados antes de alterar o código-fonte para certificar que erros de regressão não serão adicionados [55]. Assim, TAD é a abordagem mais recomendada para estes cenários. Contudo, após a criação de um conjunto pertinente de casos de testes, podem ser utilizadas outras técnicas para implementar novos testes e alterações do sistema.

Outra situação típica em que TAD é recomendada é quando um sistema possui uma falha já identificada. Antes corrigi-la, deve-se criar um caso de teste que reproduz o cenário defeituoso. Este caso de teste ajuda a identificar quando a tarefa de correção foi finalizada, e serve de precaução para que o erro não ocorra novamente.

Testar depois da implementação também é útil para as equipes que estão começando a aprender testes automatizados e as ferramentas de teste. Como TAD possui como proposta principal apenas testar o sistema, o estudo fica voltado para a automação de testes. Dessa forma, o aprendizado não é desvirtuado para a solução de problemas, elaboração de arquitetura ou detalhes de implementação.

9.2 Testes a Priori (TFD)

Desenvolvimento com testes a priori (TFD de *Test-First Development*) é a técnica que propõe implementar todos ou uma boa quantidade de casos de teste antes de desenvolver o código-fonte de uma unidade do sistema.

Para que esses testes sejam implementados, é necessário o conhecimento prévio do que será a arquitetura do sistema e a assinatura das classes em teste. Ainda, é preciso o planejamento prévio dos casos de testes.

Depois da implementação dos testes, é inviável a execução dos mesmos porque todos devem falhar, já que nada foi implementado ainda. Além disso, se a linguagem de programação utilizada tiver tipagem estática, o código dos testes pode até mesmo não compilar. Assim, os testes só poderão ser executados após a implementação dos trechos pertinentes do sistema, que devem seguir o *design* definido. A implementação do sistema deve ser feita até que todos os testes possam ser executados com sucesso. Depois disso, as etapas são repetidas até que a fase de desenvolvimento termine. Esse fluxo pode ser visualizado na Figura 9.2.

A prática de elaborar casos de testes independentemente da implementação do sistema não é nova. Muitas equipes de analistas de qualidades trabalham dessa forma, só que executam os testes após o sistema estar finalizado. Isso é possível porque, para elaborar os casos de testes, é necessário apenas o conhecimento detalhado dos requisitos.

Contudo, a ideia de implementar os testes antes mesmo do sistema é uma abordagem completamente diferente do modelo tradicional de testes de software. Ela segue fortemente a proposta da prevenção de erros, já que, provavelmente, casos importantes de serem verificados não serão deixados de lado por problemas com prazo ou por irresponsabilidade, pois o desenvolvimento se dá até que todos os testes

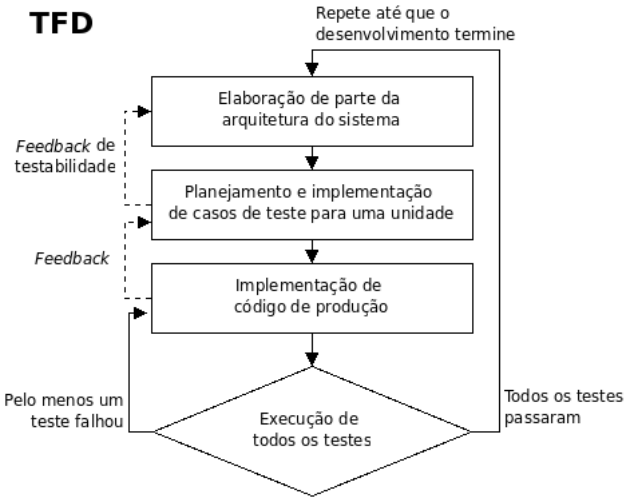


Figura 9.2: Fluxo do TFD.

obtenham sucesso. Esse aumento da prioridade da verificação do sistema tende a melhorar a sua qualidade, assim como minimizar a quantidade de erros encontrados em fases posteriores do desenvolvimento.

Características

Note que TFD, assim como TAD, sugere que a implementação dos testes e a do sistema sejam tarefas que se complementem. Contudo, a ordem em que os testes e o sistema são implementados é a oposta. Essa diferença resulta em características bem peculiares de TFD em relação a TAD.

Da mesma forma em que com TAD os testes precisam se adaptar ao código do sistema, com TFD o código do sistema é induzido a se adaptar ao dos testes, o que resulta na criação de sistemas com alta testabilidade. Apesar de os testes serem implementados com base em uma arquitetura previamente definida, as estruturas de dados devem ser modificadas de acordo com o *feedback* de testabilidade dos testes. Essas mudanças na arquitetura devem ser de fácil realização porque o sistema ainda não foi implementado.

Sendo assim, a automação de testes com TFD pode afetar significativamente o desenvolvimento dos sistemas de software. Contudo, é imprescindível o conhecimento de padrões e boas práticas de automação de testes para que o *feedback* gerado influencie positivamente no código do sistema. Se os testes forem mal implementados, provavelmente as falhas de *design* não serão identificadas.

Além disso, implementar os testes antes ou depois do sistema não impede que antipadrões sejam utilizados. A principal vantagem de implementar os testes antes do código do sistema deve-se à liberdade para criação de testes implementados de qualidade, ou seja, os antipadrões são mais facilmente evitados, pois não é necessário adaptar o código dos testes a uma arquitetura indesejada do sistema.

Uma boa maneira para evitar antipadrões é manter o código dos testes o mais simples possível. Se os casos de testes estiverem complicados de se elaborar ou de se implementar, é um indício de que algo do *design* do sistema ou da unidade precisa ser melhorado. Por exemplo, pode ser um sinal de que a unidade possui mais de uma responsabilidade, ou de que existe uma intimidade inapropriada entre elas. Já quando a unidade em teste está difícil de ser isolada, é um indício de que não é possível injetar as dependências, ou seja, que a construção do objeto está mal implementada.

Contudo, um dos maiores benefícios de se realizar os testes antes da implementação é o fato de que o desenvolvedor é forçado a refletir sobre o comportamento esperado do sistema e o que pode dar errado antes mesmo de implementá-lo.

9.3 Desenvolvimento Dirigido por Testes (TDD)

Desenvolvimento Dirigido por Testes (TDD de *Test-Driven Development*) é uma técnica de desenvolvimento de software que se dá pela repetição disciplinada de um ciclo curto de passos de implementação de testes e do sistema (Figura 9.3) [82]. Esta técnica foi descrita em 2002 por Kent Beck no livro *Test-Driven Development: By Example* [16].

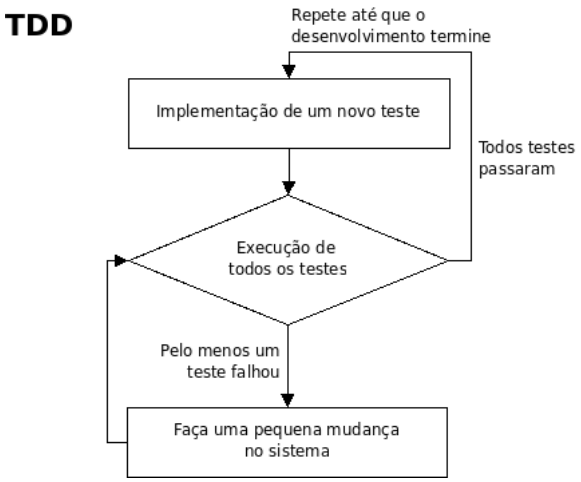


Figura 9.3: Ciclo de TDD.

O ciclo de TDD é definido por três passos:

1. Implementar apenas um caso de teste;
2. Implementar apenas um trecho de código de produção que seja suficiente para o novo caso de teste ter sucesso de tal modo que não quebre os testes previamente escritos;
3. Se necessário, refatorar o código produzido para que ele fique mais organizado. Vale notar que refatoração não altera a funcionalidade do sistema e, portanto, não deve quebrar os testes já implementados [59, 14, 122, 79]. Ainda, a refatoração pode ser feita no próprio código dos testes.

TDD é comumente confundido com TFD, pois ambas as técnicas sugerem criar os testes antes da implementação do sistema, no entanto, as diferenças são significativas. Ao contrário de TFD, TDD sugere a criação dos testes e do sistema como apenas uma tarefa, ou, por outro lado, como duas tarefas distintas que são realizadas paralelamente e sincronizadamente.

Além disso, o ciclo de TDD é muito menor do que o de TFD, pois escrever apenas um teste e um pequeno trecho de código por vez é muito mais rápido do que pensar em várias situações de teste e do sistema de uma vez. Exatamente por isso, TDD também **não** pode ser descrito como "TFD com Refatoração."

Ainda, TDD não é apenas uma técnica voltada para criação de sistemas com testes automatizados, ela também é utilizada para auxiliar a criação do *design* do sistema, por isso, certos autores preferem a denominação "*Test-Driven Design*". Isso pode ser resumido como um dos objetivos de TDD, que é escrever um código limpo que funcione.

Características

Enquanto TAD e TFD apenas adicionam a criação de testes automatizados ao processo do desenvolvimento dos sistemas de software, TDD muda completamente a forma tradicional de implementação de sistemas. Por isso, existem diversos estudos e pesquisas que tentam provar empiricamente a eficácia

dessa técnica [92]. Dentre eles, existem os que obtêm conclusões neutras, contra e a favor de TDD. Por causa das inúmeras pesquisas realizadas, um estudo foi feito para tentar unir os resultados das pesquisas para chegar a uma conclusão única, mas mesmo este estudo não obteve conclusões significativas [33].

Os aspectos mais questionáveis são: se TDD aumenta a qualidade dos sistemas [147, 25, 94]; se desenvolver com TDD é produtivo [116]; e se TDD influencia positivamente o *design* do sistema [75, 74].

TDD é uma prática que ajuda a controlar e garantir a qualidade do sistema. Ela sugere uma forma de desenvolvimento simples e disciplinada, com forte ênfase na prevenção de erros e na criação de sistemas bem modularizados. Dessa forma, se ela for aplicada corretamente, dificilmente não melhorará a qualidade do produto final. Não obstante, o próprio ciclo de TDD propicia que os testes cubram uma grande parcela do código-fonte, já que só deve ser implementado o suficiente para que os testes sejam executados com sucesso. A alta cobertura dos testes não é uma garantia de qualidade, mas é um bom indício.

Em relação à produtividade, é difícil chegar a uma conclusão, pois depende muito de cada pessoa. O tipo e o nível de experiência de um programador em relação a TDD e o desenvolvimento de software influencia na produtividade. Além disso, existem questões ainda mais subjetivas, como as relacionadas com os gostos e costumes de cada um. Pode ser difícil e até prejudicial mudar o comportamento de um desenvolvedor que trabalha há décadas de uma mesma forma.

Quanto ao *design*, o ciclo curto de passos definido por TDD cria uma dependência forte entre o código do sistema e os testes, o que favorece e facilita a criação de sistemas com alta testabilidade. É um mito dizer que com TDD todo o *design* de um sistema emerge dos testes, mas eles ajudam significativamente a criação de parte dele. O mesmo ocorre com TFD, mas com TDD o ritmo de alteração do *design* é mais dinâmico, já que a cada teste e refatoração podem surgir novas ideias.

Contudo, a elaboração prévia de uma arquitetura do sistema deve ser pensada e inicialmente seguida, mas, a partir daí, os casos de teste e as refatorações guiam a criação e as alterações no *design* do sistema. O *design* inicial do sistema pode ser elaborado com a ajuda de DDD (*Domain-Driven Design*) [8]. Outra maneira de pensar a respeito do *design* inicial das unidades é com o auxílio dos testes de aceitação. Quando estas estratégias são utilizadas, é recomendado o uso de ferramentas apropriadas, como as que seguem a linguagem de BDD, que será descrito a seguir.

9.4 Desenvolvimento Dirigido por Comportamento (BDD)

Como foi descrito na seção anterior, TDD não é apenas uma prática de verificação do código-fonte. Ela é uma prática de desenvolvimento de software que ajuda a criação de um código limpo que funcione e que influencie na elaboração do *design*. Apesar disso, TDD não deve ser utilizado como solução única para criação de sistemas bem desenhados. É imprescindível um vasto conhecimento de programação, tais como os principais conceitos de modularização e de orientação a objetos.

Além disso, um bom *design* de sistema é aquele que representa adequadamente o contexto da aplicação e as necessidades do cliente. Para isso, é fundamental o entendimento dos requisitos. Métodos ágeis recomendam que haja colaboração com o cliente, principalmente por meio da comunicação frequente e efetiva (preferencialmente face a face).

Contudo, há uma grande ponte entre o entendimento dos requisitos e a implementação pertinente do código-fonte do sistema e dos testes. Este distanciamento pode ser reduzido com a ajuda dos testes de aceitação e com a utilização de uma linguagem única e fluente a todos os membros da equipe, incluindo os clientes [104]. Uma das tendências das ferramentas de testes automatizados é tornar os testes cada vez mais próximos de uma documentação do sistema com linguagem natural. Algumas ferramentas criam DSLs (*Domain Specific Languages*) próprias para isso, como a Hamcrest; já outras geram documentos legíveis a partir do código dos testes, como a TestDox.

Entretanto, mesmo com a ajuda destas ferramentas, não é trivial criar uma linguagem ubíqua entre cliente e time de desenvolvimento. Também não é certo que as histórias serão bem definidas e os testes de aceitação serão bem implementados. A comunicação e a colaboração entre os envolvidos no desenvolvimento de um sistema de software é algo tão subjetivo que pode ser impossível determinar todos os possíveis problemas que podem ocorrer. Pensar através do comportamento de um sistema pode ajudar a amenizar essas dificuldades.

Desenvolvimento Dirigido por Comportamento (BDD de *Behavior-Driven Development*) é uma prática de desenvolvimento identificada por Dan North em 2003 [34]. Ela recomenda o mesmo ciclo de desenvolvimento de TDD, contudo, induzindo os participantes a utilizar uma linguagem diferente. Ao invés de usar os termos típicos de testes e verificações como *test suite*, *test case* e *assert* das ferramentas xUnit, as ferramentas de BDD induzem o uso de uma linguagem única (ubíqua) entre cliente e equipe de desenvolvimento. Os termos utilizados por elas são comuns em descrições de requisitos, tais como *specification*, *behavior*, *context* e *should*.

Não obstante, BDD integra explicitamente alguns princípios de DDD, de testes de aceitação e das áreas de qualidade de software para simplificar e sistematizar a definição das funcionalidades e dos cenários de teste. A definição das funcionalidades deve complementar o esqueleto definido na Figura 9.4. Este formato simples torna a descrição das funcionalidades específica, mensurável, viável, relevante e estimável. Um exemplo de história pode ser visto na Figura 9.5.

```
1 Funcionalidade: ...
2 Como um(a) ...
3 Quero ...
4 Com o objetivo de ...
```

Figura 9.4: Esqueleto de história sugerido por BDD.

```
1 Funcionalidade: Cálculo total de imposto da empresa
2 Como um(a) contador(a)
3 Quero somar todos impostos da empresa dentro de um período
4 Com o objetivo de exibir os dados na internet para protestar contra o governo
```

Figura 9.5: Exemplo de história no formato sugerido por BDD.

Os cenários de teste também possuem um esqueleto predefinido de passos, que é o padrão das descrições de teste de analistas de qualidade (Figura 9.6). Os passos `Dado` são semelhantes aos métodos de *set up* das ferramentas xUnit. Os passos `Quando` correspondem à chamada da funcionalidade em teste. Por último, a chamada `Então` é análoga às verificações. A Figura 9.7 possui um exemplo de cenário de teste para a história da Figura 9.5.

A ferramenta JBehave (primeira ferramenta de BDD), que foi criada por Dan North, baseia-se na leitura de arquivos de texto com histórias descritas no formato de passos, com uma linguagem próxima a de pessoas que não possuem perfil técnico de computação. O código dos testes (comportamentos) carregam estes arquivos e os traduzem para fazer chamadas às funcionalidades em teste. Vale ressaltar que os passos podem ser parametrizáveis, o que facilita a reutilização de código-fonte dos testes.

A ideia do JBehave é semelhante à da ferramenta Fit[1], criada por Ward Cunningham por volta de 2002. A principal diferença é que, enquanto Fit trabalha com arquivos contendo diversos tipos de tabelas, JBehave trabalha com arquivos unicamente neste formato. Essas ferramentas podem ser tanto utilizadas

[1] *Framework for Integrated Testing*.

```
Cenário: ...
Dado ...
Quando ...
Então...

# Cenários mais complexos podem possuir passos extras concatenados com "E":
Cenário: ...
Dado ...
E ...
Quando ...
Então ...
E ...
```

Figura 9.6: Esqueleto de história sugerido por BDD.

```
Cenário: Calcular total de impostos sobre o faturamento anual
Dado que em 2009 a empresa faturou R$ 200.000,00 bruto
E o total de impostos chega a 40% do total bruto
Quando calculo o total de impostos da empresa em 2009
Então a empresa gastou R$ 80.000,00 em impostos
E obteve rendimento líquido de apenas R$ 120.000,00
```

Figura 9.7: Exemplo de história no formato sugerido por BDD.

para testes de unidade quanto para testes integrados (como os de aceitação). Como mostra a Figura 9.8, pode-se escrever os testes com BDD seguindo o ciclo de ATDD, descrito na Seção 3.2.

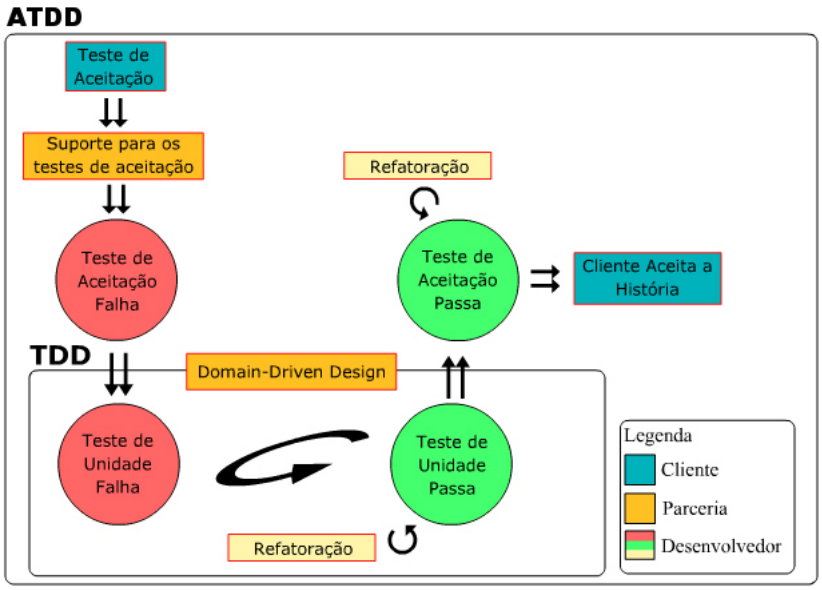


Figura 9.8: Ciclo de ATDD.

Hoje, existe uma grande gama de ferramentas de BDD para diversas linguagens de programação, todas seguindo a mesma abordagem. Dentre elas estão RSpec, Cucumber, JDave e BDoc.

9.5 Conclusões

O nível de testabilidade do sistema implica diretamente na qualidade dos testes automatizados. Mesmo que o desenvolvedor conheça padrões e boas práticas de automação, nem sempre ele conseguirá colocar em prática seu conhecimento durante a implementação dos testes porque eles precisarão necessariamente contornar as dificuldades causadas pelo alto acoplamento dos módulos e objetos da aplicação.

Quando os sistemas são implementados sem se preocupar com os possíveis cenários de testes, os sistemas tendem a possuir uma baixa testabilidade, mesmo que eles tenham arquiteturas elegantes e estejam bem implementados. Apesar de os sistemas com código-fonte de qualidade serem mais fáceis de se testar, os testes automatizados precisam ter total controle do sistema em teste para que os casos de testes sejam implementados com facilidade.

Por isso, é altamente recomendável utilizar as abordagens em que os testes são implementados antes da implementação (TFD, TDD e BDD), pois elas forçam os desenvolvedores a criarem código mais coeso, pouco acoplado e com alta testabilidade.

Entretanto, programas que possuem uma boa modelagem orientados a objetos tende a ter alta testabilidade. Se cada classe tiver apenas uma responsabilidade e for possível injetar suas dependências, então a preparação dos testes tornam-se mais simples com o auxílio de Objetos Dublês (vide Seção 6.2) e as verificações necessárias ficam fáceis de serem elaboradas.

Para esses casos, o uso de TAD também se torna uma alternativa promissora. TAD só deve ser evitada quando o custo da criação e da manutenção dos testes se torna mais alto do que a execução monótona e repetida dos testes manuais.

Parte III

Gerenciamento de Testes Automatizados

Capítulo 10

Métricas

Métrica é uma relação de uma ou mais medidas para definir se um sistema, componente ou processo possui um certo atributo. Uma medida é uma avaliação em relação a um padrão definido. Por exemplo, para saber se um sistema é grande (atributo) podemos utilizar a medida de linhas de código, que também podemos chamar de métrica de linhas de código [128, 127], já que toda medida representa um atributo, e uma métrica pode ser composta de uma única medida.

Métricas são fundamentais para revisão, planejamento e gerenciamento de projetos. Este capítulo discute seus benefícios e apresenta algumas métricas que são pertinentes para acompanhar os testes automatizados e a qualidade do sistema. Dentre elas estão Cobertura e Testabilidade, que podem ser úteis para muitos projetos, e outras que são mais benéficas para projetos com certas características, tais como projetos legados ou os que possuem equipes inexperientes em testes.

10.1 Métricas para Testes Automatizados

Toda metodologia de desenvolvimento de software busca respeitar acordos e contratos definidos com o cliente, sejam acordos de prazos, qualidade, escopo ou custo. O sucesso de um projeto depende de organização, planejamento, gerenciamento e aprendizado.

Organização é o princípio básico para que todas as informações estejam claras para facilitar o entendimento do contexto e da situação corrente do projeto. Áreas de trabalho informativas [130] contendo poucas informações, mas que são muito relevantes, são mais valiosas do que documentos completos e detalhados que acabam sendo deixados em segundo plano devido ao grande tempo que é necessário para o estudo.

Planejamento é uma proposta de organização de tarefas para serem executadas [41]. A proposta é feita contendo previsões do futuro, criadas, geralmente, com base na experiência obtida de trabalhos anteriores. Como nenhum planejamento e ninguém consegue prever o futuro com exatidão, é indiscutível que todos eles estão sujeitos a enganos. O que pode ser feito é tentar minimizar a quantidade de enganos. Para isso, os métodos ágeis recomendam trabalhar em iterações curtas de desenvolvimento, pois é mais fácil prever o futuro próximo (uma ou duas semanas) do que prever o que irá acontecer a longo prazo (meses ou anos).

Já o gerenciamento é feito pela observação do andamento do projeto e pela coordenação da equipe para concentrar o trabalho nas tarefas mais prioritárias no momento da iteração. Para observar, com êxito, o andamento dos projetos, são necessárias informações rápidas, claras, atualizadas e pertinentes [41].

O aprendizado é fundamental para o sucesso de trabalhos futuros devido à experiência adquirida que ajuda a evitar que erros sejam repetidos. Métodos ágeis incentivam a criação de reuniões de retrospectivas depois do término de uma iteração de desenvolvimento para revisar e relembrar o andamento do projeto, principalmente referente às dificuldades encontradas [49]. Nessa reunião, devem ser reforçados

os pontos que foram positivos e que precisam ser valorizados nas iterações seguintes, assim como é previsto que sejam encontradas soluções para aspectos que não foram satisfatórios e que precisam ser melhorados [130].

As métricas são fundamentais para qualquer metodologia alcançar o sucesso em um projeto, pois elas são um artifício básico para revisão, planejamento e gerenciamento. Como diz Morris A. Cohen: "Não podemos gerenciar o que não podemos medir" [47]. Elas exercem um papel fundamental no gerenciamento de tarefas e projetos, mas precisam ser bem organizadas e utilizadas nos momentos corretos. Existem estudos e abordagens sistematizadas para coleta de métricas e avaliação da qualidade de sistemas [48].

As métricas podem ser utilizadas apropriadamente quando houver necessidade de conhecer certos aspectos de um trabalho para estabelecer objetivos, como sugere o processo PDCA[1] [134]. Em alguns momentos, os problemas não estão claros ou visíveis para a equipe, por isso a coleta de métricas ajuda a identificar pontos que devem ser melhorados. As curtas iterações das metodologias ágeis seguem esta estrutura.

Quando os objetivos já estão definidos, pode-se utilizar a abordagem GQM[2] [12], que sugere a coleta unicamente das métricas pertinentes para alcançar o objetivo definido. Isso evita um esforço desnecessário de coletar e interpretar outras métricas menos importantes para o contexto. Portanto, GQM é útil para ajudar a acompanhar as soluções propostas dos problemas citados nas retrospectivas.

Uma das tarefas mais importantes do processo de desenvolvimento é gerenciar a qualidade do código-fonte e do produto produzido, que é uma tarefa complexa devido ao caráter altamente subjetivo e pessoal da característica. Para acompanhar a evolução da qualidade, é fundamental o emprego de uma ou mais métricas que consigam representar a qualidade para o contexto do projeto.

No caso das metodologias que integram testes automatizados como controle de qualidade, em especial a Programação eXtrema que recomenda TDD como uma de suas práticas primárias, é comum a coleta de métricas de testes automatizados para o acompanhamento dos testes, da qualidade do código-fonte e do produto final [11, 101]. Como os testes influenciam diretamente a qualidade e o progresso de um projeto, um bom conjunto de métricas dos testes pode elucidar o estado e ajudar a corrigir o andamento do projeto, estabelecer prioridades e estipular novas metas.

As seções seguintes apresentam algumas métricas de testes automatizados e de qualidade que são úteis tanto para equipes que estão começando a aprender e aplicar testes automatizados como para aquelas já experientes que possuem grandes baterias de testes automatizados. Também são valiosas tanto para projetos legados quanto para os recém-criados.

10.2 Cobertura

A métrica de cobertura de código indica quais pontos do sistema foram exercitados (executados ou cobertos) pelos casos de teste [144]. Os pontos do sistema podem ser classes, métodos, blocos e linhas, sendo que a granularidade mais fina de cobertura que geralmente as ferramentas obtêm são as linhas de código executadas. A Figura 10.3 apresenta o relatório de cobertura dos testes gerados com a ferramenta Eclemma para Java após a execução dos testes da Figura 10.2 sob o código da Figura 10.1 que possui simplesmente um método que calcula o máximo divisor comum (m.d.c.) de dois números naturais.

As linhas em tons mais claros (verde: 7, 8, 10, 11, 12 e 17) foram executadas pelos testes, ao contrário das linhas com tom mais escuro (vermelho: 13, 14 e 15). As linhas que não foram executadas indicam que faltam testes para o cenário onde o resto da divisão entre dividendo e divisor é diferente de zero. O tom intermediário (amarelo: 9) significa que a linha foi parcialmente executada, isto é, algumas das operações foram realizadas e outras não. Neste caso, a cobertura aponta que faltam cenários de teste

[1] *Plan* -> *Do* -> *Check* -> *Act* ou Planeje -> Faça -> Estude -> Aja.
[2] GQM: *Goal* -> *Question* -> *Metrics* ou Objetivo -> Questão -> Métricas.

```java
public class MathHelper {

  // Algoritmo de Euclides: mdc(a, b) = mdc(b, resto(a, b)) => (a = q * b + r)
  public static long mdc(long a, long b) {
    long dividendo = Math.max(a, b);
    long divisor = Math.min(a, b);
    if(dividendo < 0 || divisor < 0) throw new IllegalArgumentException("ops");
    if(dividendo == 0  || divisor == 0) return 0;
    long resto = dividendo % divisor;
    while(resto != 0) {
      dividendo = divisor;
      divisor = resto;
      resto = dividendo % divisor;
    }
    return divisor;
  }

}
```

Figura 10.1: Exemplo de código para verificação da cobertura.

```java
// referências do JUnit
import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class MathHelperTests {

  @Test public void mdcComZeroEhZero() {
    assertEquals(0, MathHelper.mdc(0, 1));
    assertEquals(0, MathHelper.mdc(1, 0));
    assertEquals(0, MathHelper.mdc(0, 0));
  }

  @Test public void mdcComUmEhUm() {
    assertEquals(1, MathHelper.mdc(1, 1));
    assertEquals(1, MathHelper.mdc(1, 7));
    assertEquals(1, MathHelper.mdc(1, 20));
    assertEquals(1, MathHelper.mdc(1, 25));
    assertEquals(1, MathHelper.mdc(1, 100));
  }

}
```

Figura 10.2: Exemplo de testes para verificação da cobertura.

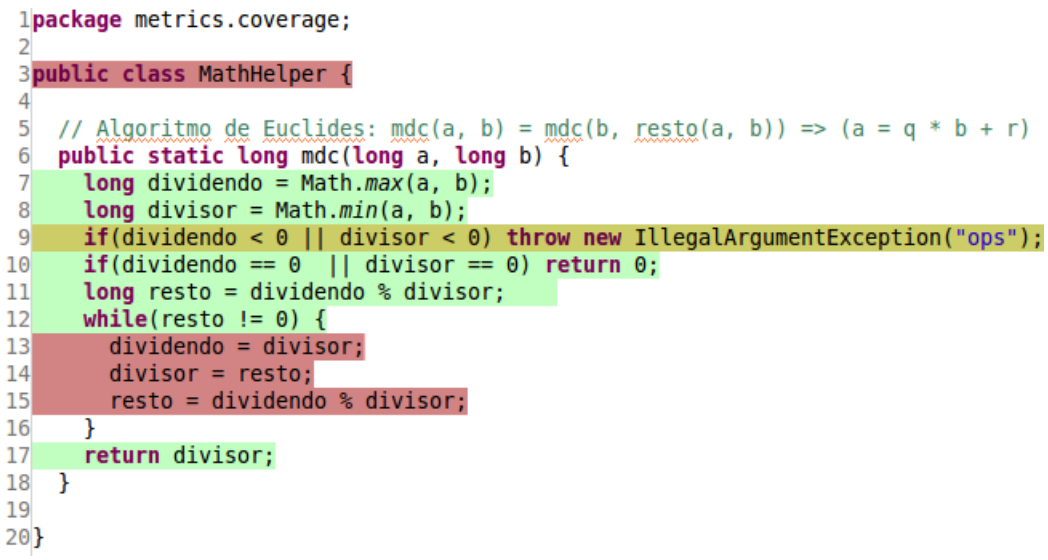


Figura 10.3: Visualização da cobertura do código-fonte com a ferramenta Eclemma.

com dados de entrada inválidos, em que pelo menos um dos dados de entrada é um número negativo. Sendo assim, a condição `if` foi executada, enquanto o lançamento da exceção de erro não foi processada.

Note que um trecho exercitado não significa que ele está livre de defeitos ou que foi testado completamente [108]. Por exemplo, se a linha 11 possuísse o código incorreto `long resto = 1`, todos os testes da Figura 10.2 continuariam sendo executados sem falhas. A cobertura nem sequer mostra se um trecho está realmente sendo testado. Por exemplo, se o código dos testes fizesse apenas as chamadas do método `mdc` sem fazer as verificações, isto é, `MathHelper.mdc(a, b)`, em vez de `assertEquals(x, MathHelper.mdc(a, b))`, o resultado da cobertura continuaria sendo o mesmo.

A única certeza que a métrica de cobertura fornece é que os trechos não exercitados não foram testados. A falta de testes pode indicar trechos do sistema que são desnecessários ou, ainda, pontos específicos que podem conter falhas e que precisam ser verificados. Para exemplificar, qualquer erro que não seja de compilação na linha 15 passa despercebido pela fraca bateria de testes apresentada. Poderia ter desde erros de distração, como, por exemplo, `resto = dividendo / divisor`, até um erro grosseiro, como `resto = 0`.

Portanto, esta métrica precisa ser interpretada com atenção e não deve ser utilizada como único indicador de qualidade do sistema, é necessário outras métricas que complementam o conhecimento adquirido da cobertura de testes.

Esta métrica é muito útil para diversos contextos. Quando o sistema já possui uma bateria de testes consideravelmente grande, ela é fundamental para indicar novos pontos que precisam ser verificados. Quando a bateria de testes é pequena, não é útil para dar visão do sistema como um todo, mas pode ser utilizada para encontrar novos pontos de verificação dentro de um pequeno módulo do sistema.

Para os sistemas que não possuem testes automatizados, esta métrica é desnecessária porque o resultado sempre será 100% de trechos não cobertos. É até possível utilizar as ferramentas de coleta de cobertura durante a realização de testes manuais, mas os resultados obtidos só são úteis para o instante da execução, isto é, eles não são úteis para serem acompanhados com o tempo porque as execuções não são fielmente idênticas, o que torna inviável a interpretação coerente dos resultados.

Quando o sistema é implementado com TDD ou TFD, a cobertura de código tende a ser alta, já que cada trecho de código só deve ser adicionado após um teste que o cubra [153]. Então, ela pode ser utilizada para auxiliar o desenvolvimento com testes a priori para indicar falhas no processo e para verificar se um sistema possui indícios de ter sido escrito com TDD ou TFD.

10.3 Testabilidade

Testabilidade de software mede a facilidade da escrita de testes de qualidade dentro de um contexto [142]. Testar um sistema ou uma funcionalidade nem sempre é trivial, seja por meio de testes manuais ou automatizados. Primeiramente, tem de ser possível controlar o software, isto é, executá-lo de acordo com as necessidades do teste. Posteriormente, é necessário observar os efeitos colaterais causados por uma determinada execução que serão comparados com valores esperados.

Para que seja possível controlar e observar um sistema apropriadamente é necessário que ele esteja bem modularizado e isolado, pois, caso contrário, os testes podem ser difíceis ou até impossíveis de serem realizados. Testes complicados de serem criados ou mantidos tendem a ser pouco legíveis e muito suscetíveis a erros. Tudo isso aumenta o custo-benefício da automação de testes.

Testabilidade não é uma métrica intrínseca ao sistema, como total de linhas de código, número de classes etc. É necessário medir diversos fatores para então calcular o grau de testabilidade de acordo com alguma fórmula matemática baseada em determinações subjetivas. Por isso, é importante ter cuidado ao interpretar esta métrica, já que ela pode ser mais ou menos apropriada para um contexto específico. Também é preciso cautela ao comparar o grau de testabilidade entre sistemas que possuem contextos diferentes.

A Figura 10.4 mostra um exemplo do grau de testabilidade de um módulo do software Eclipse medido com a ferramenta *Testability-Explorer* [69]. A ferramenta analisa o código-fonte de cada classe Java em busca de variáveis de classe mutáveis, incoerências segundo a Lei de Demeter[3] [88] e classes que não permitem injetar dependências. As informações coletadas são convertidas em um grau de testabilidade que representa custo para se testar uma classe do sistema, sendo assim, quanto maior o custo, pior.

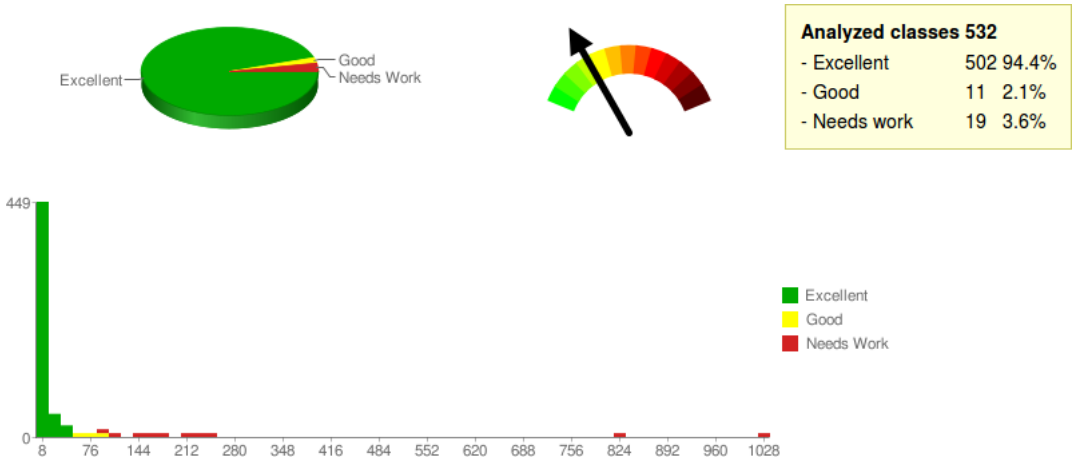


Figura 10.4: Grau de testabilidade do módulo *Workbench* do software Eclipse, medido com a ferramenta Testability-Explorer.

As classes que possuem boa ou excelente testabilidade são mais fáceis de testar automaticamente, enquanto as que possuem baixa testabilidade (*Needs Work*) requerem mais atenção. Pode ser bem complexo criar testes para as classes com baixa testabilidade, pois, geralmente, é necessário uma grande quantidade de código para preparar o ambiente de testes apropriadamente. Em algumas situações, pode até ser fácil de implementar os testes, mas é provável que eles contenham antipadrões, tais como testes lentos, intermitentes e frágeis.

[3] A Lei de Demeter propõe um estilo de *design* em que cada unidade deve ter conhecimentos limitados sobre outras unidades. Uma unidade só deve trocar informações com suas dependências imediatas. Este princípio é particularmente útil para criar sistemas orientados a objetos com baixo acoplamento.

10.3.1 Padrões e Antipadrões Influenciam a Testabilidade

A testabilidade de um sistema está diretamente relacionada com o bom *design* [117, 97, 27, 78]. O grau de testabilidade pode ser avaliado a partir de boas práticas de modularização e de programação orientada a objetos, que ajudam a tornar os sistemas mais simples e coesos, e, consequentemente, os testes ficam mais claros e fáceis de implementar.

Um dos fatores mais comuns que afetam a testabilidade do sistema está relacionado com a implementação do construtor de objetos. Objetos que são difíceis de criar e de configurar dificultam a criação de testes isolados. Existem inúmeros antipadrões não nomeados que prejudicam a testabilidade de um sistema. A Figura 10.5 apresenta exemplo de construtores escritos em Java com alguns destes antipadrões. Para facilitar, os comentários de cada exemplo estão nas próprias figuras.

Já a Figura 10.6 mostra objetos que são fáceis de serem criados e isolados por meio do padrão Injeção de Dependência [113]. Quando a criação dos objetos é mais complexa, é recomendado que o processo de criação seja isolado em objetos próprios, através do uso de padrões de projeto de criação, tais como *Builder*, *Factory* e *Prototype* [61].

A definição das responsabilidades dos objetos e da forma como eles irão interagir determinam o *design* de um sistema orientado a objetos. Isso é uma das tarefas mais difíceis e delicadas da orientação a objetos, pois qualquer falha pode comprometer a manutenibilidade e a testabilidade. A Figura 10.7 apresenta um método pouco coeso que viola a Lei de Demeter [88], portanto torna os testes difíceis de serem isolados. A Figura 10.8 apresenta como seria o método refatorado para aumentar a coesão e testabilidade do sistema.

Uma das boas práticas de orientação a objetos é definir uma responsabilidade por classe. Esse padrão torna as classes fáceis de serem implementadas, entendidas e testadas. Quando uma classe possui muitas responsabilidades, o conjunto de casos de testes para testá-la tende a aumentar consideravelmente, já que as combinações de dados de entrada podem aumentar fatorialmente. Além disso, os testes tendem a ter a legibilidade prejudicada. Quanto mais complexa for uma classe, maior serão os métodos de *set up* e mais verificações são necessárias por casos de teste.

Existem vários indícios que ajudam a identificar se uma classe está realizando mais tarefas do que deveria. O primeiro deles se dá pela facilidade de entendimento da classe. Outros indícios são os atributos da classe raramente usados, que podem indicar que ela deve ser repartida entre outras menores. Mais um indício comum é o uso de nomes de classes e variáveis muito genéricas, tais como *manager*, *context*, *environment*, *principal*, *container*, *runner* etc. Devido ao caráter extremamente abstrato dessas classes, é coerente que muitas responsabilidades sejam associadas a elas.

Ainda, a modelagem incorreta de heranças entre classes, por exemplo, que ferem o Princípio de Substituição de Liskov [89], podem tornar os testes difíceis de serem modularizados e, consequentemente, propiciar a replicação de código-fonte (vide Seção 6.3.3).

Outro aspecto que prejudica a automação de testes são os estados globais mutáveis, tais como variáveis globais e classes com o padrão *Singleton* [61]. Estados globais são acessíveis a vários casos de testes, portanto, os testes não ficam completamente isolados. Isso significa que se um caso de teste alterar um valor global, outros testes poderão falhar.

Uma solução que torna os testes mais complexos e lentos, mas que pode resolver o problema de maneira padronizada, é utilizar os métodos de *set up* para definir o estado inicial das variáveis globais antes da execução dos testes. Contudo, esta solução requer que os testes sejam executados individualmente e sequencialmente, o que inviabiliza o uso de ferramentas de otimização de testes que executam paralelamente os casos de testes.

Esses padrões e antipadrões citados compõem apenas uma pequena parcela das inúmeras formas de implementação que influenciam a testabilidade do sistema. Ainda, existem diversos fatores próprios de cada linguagem de programação que também podem facilitar ou dificultar os testes de software. O que é válido para todas elas é que definir e escrever os casos de testes antes da própria implementação propicia a criação de código altamente testável, já que o código do sistema se adapta aos testes, e não o contrário.

```java
public class ObjetoComContrutorDeBaixaTestabilidade1 {
  // Não da para isolar (classe não tem método setter)
  private Dependencia dependencia = new Dependencia();
  public ObjetoComContrutorDeBaixaTestabilidade1() {
  }
}

public class ObjetoComContrutorDeBaixaTestabilidade2 {
  private Dependencia dependencia;
  public ObjetoComContrutorDeBaixaTestabilidade2() {
    // Não da para isolar (classe não tem método setter)
    dependencia = new Dependencia();
  }
}

public class ObjetoComContrutorDeBaixaTestabilidade3 {
  private Dependencia dependencia;
  public ObjetoComContrutorDeBaixaTestabilidade3() {
    // Arcabouços que usam reflexão para criação de objetos precisam do construtor
        padrão.
    // Mas é necessário cuidados porque o objeto não está inicializado
    // apropriadamente enquanto não forem injetadas as dependências.
  }
  public void setDependencia(Dependencia dependencia) {
    this.dependencia = dependencia;
  }
}

public class ObjetoComContrutorDeBaixaTestabilidade4 {
  public ObjetoComContrutorDeBaixaTestabilidade4() {
    // Muita lógica no contrutor pode prejudicar a testabilidade.
    // É necessário um trabalho adicional para criar o objeto apropriadamente.
    // Use algum padrão de projeto de criação de objetos.
    if (x == 3) { ... }
    for(int i = 0; i < n; i++) { ... }
    while(true) { ... }
  }
}

public class ObjetoComContrutorDeBaixaTestabilidade5 {
  public ObjetoComContrutorDeBaixaTestabilidade5(Dependencia dependencia) {
    // Testes não ficam isolados da classe DependenciaGlobal
    DependenciaGlobal.metodoGlobal(dependencia);
  }
}
```

Figura 10.5: Exemplo de implementação de construtores que tornam os objetos difíceis de serem testados.

```java
public class ObjetoComContrutorDeAltaTestabilidade1 {
  private List list;
  public ObjetoComContrutorDeAltaTestabilidade1() {
    // Detalhes internos do objeto podem ser instanciados no contrutor
    // lista geralmente é uma exceção
    list = new ArrayList();
  }
}

public class ObjetoComContrutorDeAltaTestabilidade2 {
  private Dependencia dependencia;
  public ObjetoComContrutorDeAltaTestabilidade2(Dependencia dependencia) {
    this.dependencia = dependencia; // Possível injetar dependência
  }
}
```

Figura 10.6: Exemplo de implementação de construtores que tornam os objetos fáceis de serem testados.

```java
public class A {

  public metodo(B b) {
    // Violação da Lei de Demeter:
    // Objeto A conhece toda hierarquia de classes do objeto B
    // Difícil de isolar: A pergunta para B por informações
    var estado = b.getObjetoC().getObjetoD().getObjetoE().getEstado();
    // ...
  }
}
```

Figura 10.7: Exemplo de implementação de métodos que são difíceis de serem testados.

```java
public class A {

  // Não pergunte, diga!
  public metodo(E e) { // Possível isolar dependências
    var estado = e.getEstado(); // A só conhece E
    // Objetos B, C e D são dispensáveis
    // ...
  }
}
```

Figura 10.8: Exemplo de implementação de métodos que são fáceis de serem testados.

Por isso é recomendado o uso de TFD, TDD e BDD para criação de sistemas com alta testabilidade.

10.3.2 Quando Utilizar

Testabilidade pode ser útil para analisar riscos e estabelecer quais pontos são mais críticos para se testar, principalmente para equipes que estão começando a aplicar testes automatizados em sistemas legados. Contudo, ela também é importante para acompanhar a qualidade dos testes automatizados, pois baixa testabilidade (ou alto custo para se testar) implica testes com muitos antipadrões.

Essa métrica também auxilia na identificação dos módulos do sistema em teste que precisam ser refatorados, isto é, módulos que não possuem um bom *design*. Dessa forma, a análise da testabilidade do sistema, antes de adicionar novas funcionalidades, é importante, pois ajuda a prevenir que uma nova porção de código seja inserida sobre uma arquitetura confusa, que pode tornar a implementação mais complicada, além de piorar o *design* do sistema.

TDD e TFD não só proporcionam alta cobertura dos testes, como também favorecem para que o sistema seja testável, pois a criação dos testes a priori implica que o código do sistema se adapte aos testes, e não o contrário [16]. Portanto, testabilidade também é útil para acompanhar e verificar se o desenvolvimento com TDD ou TFD está sendo feito apropriadamente.

10.4 Outras métricas

As métricas de cobertura e testabilidade podem ser úteis para todos os contextos de desenvolvimento de sistemas de software, mesmo quando estamos seguindo a abordagem PDCA ou GQM. No entanto, existem muitas outras métricas que são úteis para contextos específicos e que podem ajudar a encontrar defeitos, melhorar o código do sistema e dos testes, além serem métricas valiosas para acompanhar a automação de testes, tanto para novos projetos quanto para projetos legados [144]. A seguir é apresentada uma lista com algumas destas métricas:

1. **Fator de Teste**: É o total de linhas dos testes pelo total de linhas do sistema. É útil para comparar módulos de um mesmo projeto e ajudar a determinar quais são os módulos mais testados e quais precisam de maior atenção. Esta métrica não é recomendada para o acompanhamento da evolução dos testes de um projeto, pois a lógica e o tamanho do código dos testes não possuem qualquer relação com a lógica do código do sistema.

2. **Número de testes por linha de código do sistema**: Esta métrica pode ser útil para acompanhar a evolução dos testes automatizados de um projeto, desde que o sistema cresça sem alterações drásticas quanto a sua complexidade.

3. **Número de linhas de testes**: Análogo ao número de linhas de um sistema, esta métrica dá uma pequena dimensão do código dos testes e pode ser utilizada para o planejamento de tarefas de estudo e manutenção do código. A avaliação dessa métrica em uma amostra do código, como classes ou métodos, pode identificar testes que precisam de refatoração.

4. **Número de testes**: Métrica para acompanhar a evolução do desenvolvimento de testes. Útil, principalmente, em projetos que estão começando a ter testes automatizados.

5. **Número de asserções**: É uma métrica que ajuda a detectar se os testes estão realmente testando o sistema, isto é, se estão fazendo verificações.

6. **Número de testes pendentes**: É comum escrever testes marcados como pendentes (se o arcabouço de teste fornecer essa funcionalidade) que serão implementados no momento apropriado. Por exemplo, se o acompanhamento da quantidade dos testes ao longo de uma iteração indicar

que o número de pendências não está diminuindo, pode ser um sinal de que os prazos estão curtos e que os testes estão sendo sacrificados.

7. **Número de testes falhando**: Esta métrica é útil para detectar a fragilidade dos testes, além de servir de acompanhamento para o conserto dos mesmos.

8. **Número de asserções por método**: Indica métodos que talvez precisam ser refatorados caso o número seja alto, pois podem ser responsáveis por testar mais de uma ideia.

9. **Replicação de código dos testes**: Identifica trechos do código dos testes que precisam ser refatorados, assim como pode indicar que o código do sistema também possui replicação de código.

10. **Quantidade de defeitos encontrados**: Pode indicar a qualidade do sistema e também a falta de testes automatizados.

11. **Tempo de execução da bateria dos testes**: Métrica para determinar se o programa ou os casos de testes possuem gargalos de desempenho que precisam ser otimizados ou refatorados.

10.5 Conclusões

Os valores obtidos das métricas estão diretamente relacionados ao contexto do projeto e do sistema, pois elas dependem de muitos fatores como a linguagem de programação e a complexidade do produto. Dessa forma, é inviável utilizá-las isoladamente para definir o estado de um projeto ou mesmo para comparar projetos distintos.

Para estes fins, sempre é necessário uma análise que fará a interpretação das informações, principalmente porque os resultados podem apresentar valores que não são esperados para a realidade do projeto, isto porque as métricas são, muitas vezes, facilmente burladas, propositalmente ou por falta de experiência com o desenvolvimento dos testes, como demonstram os exemplos a seguir:

- **Exemplo 1**: Um número alto de testes e de fator de teste pode aparentar que um sistema possui poucos defeitos, já que passa a impressão de que foram verificados diferentes cenários de testes, mas também pode indicar que o código do sistema possui uma grande replicação de código, e, portanto, os testes também são replicados.

- **Exemplo 2**: Uma alta cobertura do código provavelmente indica que o sistema está muito bem testado, já que não há trechos de código esquecidos, mas também pode mostrar que o sistema foi muito mal testado, caso os testes não possuam verificações, apenas chamadas dos métodos do sistema.

Estas métricas relacionadas a testes automatizados ajudam a estabelecer objetivos de melhoria da qualidade e da produtividade da automação dos testes, dentre outros objetivos que são comuns a diversos projetos. A Tabela 10.1 aponta alguns destes objetivos e as métricas mais recomendadas para ajudar no gerenciamento.

É importante lembrar que outras métricas podem ser geradas a partir da combinação de uma ou mais destas métricas, pois cada uma delas utiliza uma medida diferente. A busca de novas métricas sempre é útil, pois quanto mais evidências, mais fácil é a análise dos dados e também a definição de estratégias. No entanto, a coleta de métricas não deve prejudicar a agilidade do processo, assim como o excesso de informações não deve tirar o foco do que é realmente necessário melhorar.

Objetivo \ Métrica	1	2	3	4	5	6	7	8	9	10	11	12	13
Encontrar defeitos		o	o									o	
Melhorar o código do sistema	o	o									o		o
Melhorar o código dos testes		o	o		o		o	o	o	o	o	o	o
Introduzir testes automatizados em novos projetos						o							
Introduzir testes automatizados em sistemas legados	o											o	
Acompanhar a automação dos testes		o		o	o	o		o	o			o	

Tabela 10.1: Objetivo *vs.* Métrica (*Goal vs. Metric*). Legenda: (1) Testabilidade; (2) Cobertura; (3) Fator de Teste; (4) Número de testes por linha de código do sistema; (5) Número de linhas de testes; (6) Número de testes; (7) Número de asserções; (8) Número de testes pendentes; (9) Número de testes falhando; (10) Número de asserções por método; (11) Replicação de código dos testes; (12) Quantidade de defeitos encontrados; e (13) Tempo de execução da bateria dos testes.

Capítulo 11

Considerações Finais

Desenvolvimento de software é uma tarefa complexa que exige conhecimento técnico, organização, atenção, criatividade e também muita comunicação. É previsível que durante o desenvolvimento alguns destes requisitos falhe, mas é imprevisível o momento que irão falhar. Por isso, é imprescindível que exista uma maneira fácil e ágil de executar todos os testes a qualquer momento, e isso é viável com o auxílio de testes automatizados.

A automação dos testes traz segurança para fazer alterações no código, seja por manutenção, refatoração ou até mesmo para adição de novas funcionalidades. Além disso, um teste programático permite criar testes mais elaborados e complexos, que poderão ser repetidos identicamente inúmeras vezes.

Ainda, a automação aumenta a quantidade de tempo gasto com a verificação do sistema e diminui o tempo gasto com a identificação e correção de erros (tempo perdido). Todos os testes podem ser executados a qualquer momento e, por consequência, os erros tendem a ser encontrados mais cedo. É possível até automatizar a execução dos testes, com ferramentas que ficam constantemente verificando se um código foi alterado ou com aquelas que obtêm o código de um repositório automaticamente e rodam a bateria de testes por meio de um *script*.

No entanto, a automação de testes é um processo complexo, sujeito a erros e que precisa de manutenção. Por isso, é fundamental que as baterias de testes sejam de alta qualidade, ou seja, organizadas, legíveis, rápidas etc. Para isso, é essencial o conhecimento de boas práticas, padrões, antipadrões, e indícios de problemas.

Além disso, os testes automatizados possuem influência na forma que um software é modelado. Os sistemas que são implementados sem testes automatizados tendem a possuir uma baixa testabilidade, mesmo que o código seja de alta qualidade. Por isso, é aconselhável utilizar abordagens que forçam os desenvolvedores a criarem código com alta testabilidade.

Apesar dos testes automatizados ajudarem na criação de uma modelagem coesa e pouco acoplada do sistema, o objetivo principal desta prática é verificar a qualidade de diferentes características que são importantes para o projeto. Portanto, é fundamental a utilização das soluções propostas pela área de Teste de Software, que são completamente compatíveis com as abordagens sugeridas pela área de Metodologias Ágeis. Por exemplo, é possível integrar boas práticas de verificação de código em conjunto com desenvolvimento dirigido por testes, além de que o progresso da automação de testes de um projeto pode ser acompanhado por meio de métricas de software.

11.1 Pontos para Pesquisa

Esta área de pesquisa está em crescimento, existem muitas pesquisas a serem feitas, muitas ferramentas ainda não produzidas e muitas técnicas ainda não evidenciadas. Testes automatizados já têm trazido benefícios significativos para muitos projetos, mas pesquisas que comparam as técnicas de escrita ou que comprovam a eficácia dessa prática ainda podem ser úteis. Todavia, a tendência é facilitar a escrita

dos testes para baixar o custo de implementação e manutenção. Esta tendência é comprovada por ferramentas com APIs mais fáceis de usar, que geram código de teste, e outras que até geram casos de testes pertinentes.

Abaixo, segue uma lista de propostas de ferramentas para serem implementadas e de estudos empíricos que são difíceis de serem realizados, já que é complicado isolar outras variáveis do desenvolvimento de software que atrapalham a interpretação dos resultados.

Sugestões de pesquisas:

- Pesquisa em que quatro grupos pequenos com o mesmo nível de experiência de programação irão implementar um mesmo sistema. Uma das equipes utilizará TDD, outra TFD, outra TAD e outra irá fazer apenas testes manuais. O tempo de desenvolvimento e a qualidade do código e do produto gerado serão analisados para buscar evidências de vantagens e de desvantagens de cada prática.
- Pesquisa em que dois grupos com o mesmo nível de experiência de programação e de TDD irão implementar um mesmo sistema. Uma das equipes utilizará TDD e com o uso exaustivo de Objetos Dublês, enquanto o segundo grupo irá fazer testes contendo certa integração dos módulos e só utilizará Objetos Dublês para casos críticos. O tempo de desenvolvimento e a qualidade do código, dos testes e do produto gerado serão analisados para buscar evidências de vantagens e de desvantagens de cada prática.
- Pesquisa em que dois grupos com o mesmo nível de experiência de programação e de TDD irão implementar um mesmo sistema que tenha um linguajar não conhecido pelas equipes. Uma das equipes utilizará TDD enquanto a outra, o BDD. O tempo de desenvolvimento e a qualidade do código, dos testes e do produto gerado serão analisados para buscar evidências de vantagens e de desvantagens de cada prática.

Sugestões de Estudos:

- Continuar a documentar padrões, antipadrões e indícios de problemas.
- Encontrar padrões ao se testar os Padrões de Projetos e os Arquiteturais.
- Encontrar padrões de testes para Programação Funcional.
- Encontrar padrões de testes de *Web Services*.

Sugestões de Ferramentas:

- Criar arcabouços para testes com aspectos. A partir de *pointcuts*, protótipos de classes podem ser gerados automaticamente para a realização dos testes.
- Aperfeiçoar as ferramentas de relatórios de testes para torná-los mais legíveis e terem maior utilidade para documentação. Por exemplo, as ferramentas podem analisar os nomes dos métodos que utilizam a convenção *camel case* ou o caractere *underline* para formatar de uma maneira mais legível, com espaço. Já existem ferramentas com esse propósito, mas ainda falta integração com as ferramentas mais populares.
- Complemento da ferramenta Python-QAssertions e conversão da ferramenta para outras linguagens. Outras asserções podem ser adicionadas, como uma que produza casos úteis de testes para expressões regulares, ou, então, asserções que gerem testes úteis para padrões de projetos e operações comuns em banco de dados (CRUD).
- Criar ferramentas que facilitem e incentivem o uso de padrões identificados.
- Criar frameworks de testes automatizados próprios para testarem sistemas paralelos e distribuídos.

Sugestões de Ferramentas para Testes com Persistência de Dados:

- Adaptar as ferramentas de testes com persistência de dados de arcabouços Web para que os testes sejam executados em diversas instâncias de banco de dados em memória, assim, os testes poderão ser executados em paralelo.

Sugestões de Ferramentas para Testes de Interface de Usuário:

- Ferramenta que gera uma Camada de Abstração das Funcionalidades da interface de usuário para facilitar a escrita dos testes de interface.
- Aperfeiçoamento das ferramentas de gravação de testes de interface, de tal modo que facilite a criação de módulos, evitando a repetição de código. Também pode-se evidenciar os pontos que precisam ser refatorados.
- Estudos e ferramentas para testes de usabilidade, baseadas em heurísticas recomendadas pela área de Interação Homem-Computador. Para testes de interface Web, podem ser analisados documentos CSS para identificar o contraste das cores dos componentes, assim como o tamanho das fontes utilizadas.
- Ferramentas para facilitar testes de leiaute. Elas podem detectar componentes que não estão visíveis assim como irregularidades do leiaute. Em aplicações Web, podem ser analisados os componentes que possuam a propriedade de invisibilidade (*display*), assim como as propriedades de localização tridimensional (*z-index*).

Sugestões de Ferramentas de Métricas de Testes:

- Criar métricas de padrões de qualidade que se baseiam padrões do qualidade descritos na dissertação.
- Criar ferramentas que detectem antipadrões nos testes, assim como o Testability-Explorer encontra antipadrões de testabilidade no código do sistema.
- Ferramentas para coleta e exibição de métricas de testes automatizados. Converter a ferramenta Testability-explorer (para Java) para outras linguagens.
- Criar um robô que explore repositórios de código e ferramentas de administração de defeitos para se obter métricas que relacionam quantidade de defeitos com a quantidade de testes.

Apêndices

Apêndice A

Teste de Carga com JMeter

JMeter é uma ferramenta livre para *Desktop*, implementada em Java/Swing e que facilita a criação de testes de carga, estresse, desempenho e longevidade. Os testes são criados com auxílio da interface de usuário, dispensando o uso de código-fonte. O teste é definido por intermédio de uma árvore de comandos a ser executados (Plano de Teste), sendo que cada comando é representado por um elemento fornecido pela interface. As figuras a seguir mostram um exemplo simples de um teste de carga para uma aplicação Web.

Na Figura A.1 é possível ver as informações globais do teste (lado direito da figura) e a árvore de comandos a ser executados (lado esquerdo). No Plano de Teste, é possível definir inúmeras variáveis (a tabela da figura), as quais são visíveis a todos os comandos do teste. Já em relação a árvore de elementos, cada tipo de comando é representado por um ícone e um nome. Alguns desses comandos serão descritos nas próximas figuras.

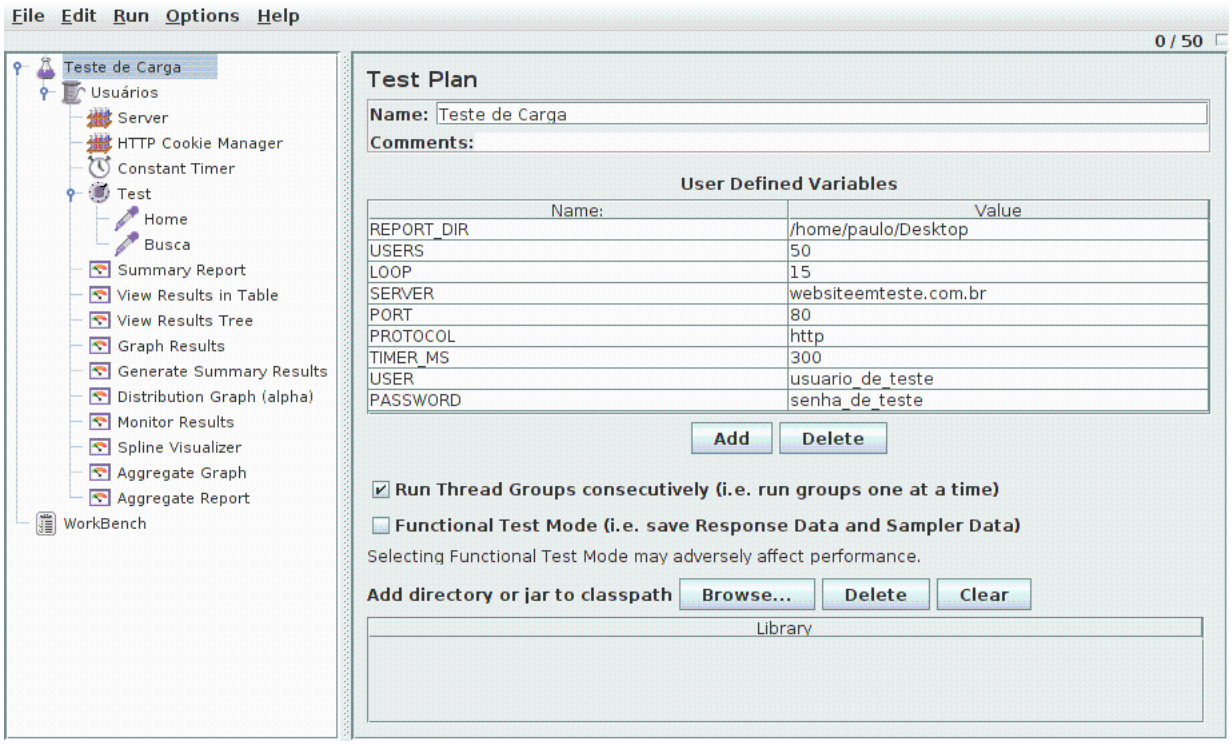


Figura A.1: Configurações do Plano de Teste com JMeter.

O comando `Usuários` é do tipo *Thread Group* (Grupo de *Threads*, Figura A.2), que define a quan-

tidade de usuários que serão simulados acessando o sistema (*Number of Threads*). Ainda, é possível configurar algumas opções de como esses usuários irão fazer as requisições (as outras opções da parte direita da janela). Por exemplo, a *Loop Count* define quantas vezes cada usuário irá repetir os passos do teste.

Note que os valores definidos nessa janela utilizam algumas das variáveis definidas no comando Plano de Teste. Isso foi feito para centralizar as configurações mais importantes em um só lugar. Assim, uma mesma árvore de comandos pode ser facilmente aproveitada para realizar os testes em diferentes ambientes, bastando alterar as variáveis pertinentes.

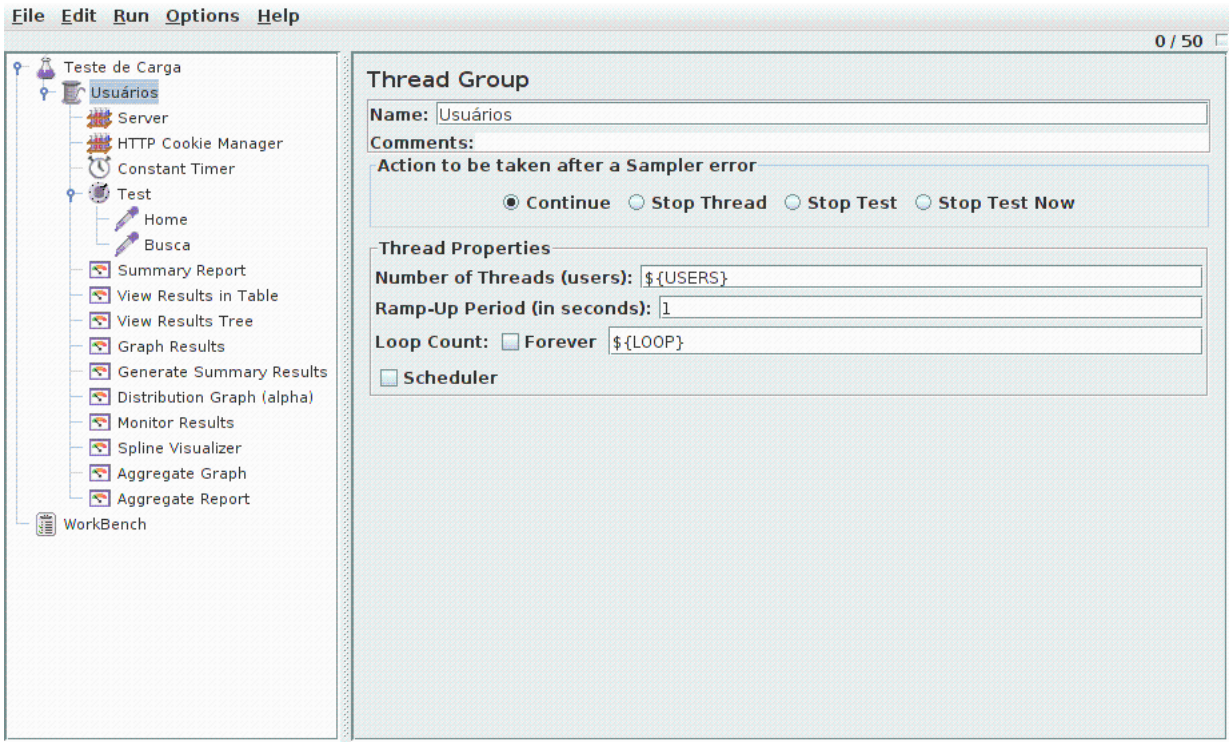


Figura A.2: Configurações dos Usuários que serão simulados pelo JMeter.

Outro comando de configuração, que é essencial para testes de aplicações Web, é o do tipo *HTTP Request Defaults* (Figura A.3). Nele é possível definir o protocolo de acesso, o servidor e a porta, assim como o tipo de codificação (*encoding*) e os tempos máximos de conexão e resposta (*timeouts*) das requisições, além de outras configurações.

Tendo definido as configurações centrais do teste, é momento de definir quais páginas os usuários simulados irão acessar, representado pelo comando `Test`. Esse comando serve para agrupar um conjunto de ações para melhorar a organização e reutilização dos elementos. Existem ainda outros comandos lógicos, tais como condicionais e de laços.

Nesse teste simples, o usuário irá apenas acessar a página inicial (*Home*, Figura A.4) e fazer uma busca (`Busca`, Figura A.5), que são comandos do tipo *HTTP Request* (Requisição HTTP). Para acessar a página inicial, basta a execução de uma requisição HTTP do tipo GET no caminho `/`, enquanto, para executar a busca, é necessário fazer um POST para `/search` contendo o texto a ser buscado.

O comando de requisição HTTP ainda possui outras opções, além de que é possível sobrescrever as configurações definidas no comando *HTTP Request Defaults*. Para definir essas opções, é preciso conhecer em detalhes como funciona o sistema.

Quando o Plano de Teste é executado, o JMeter carrega as configurações, cria as *threads* que simularão os usuários e executa as requisições previamente definidas. Entretanto, um teste não é completa-

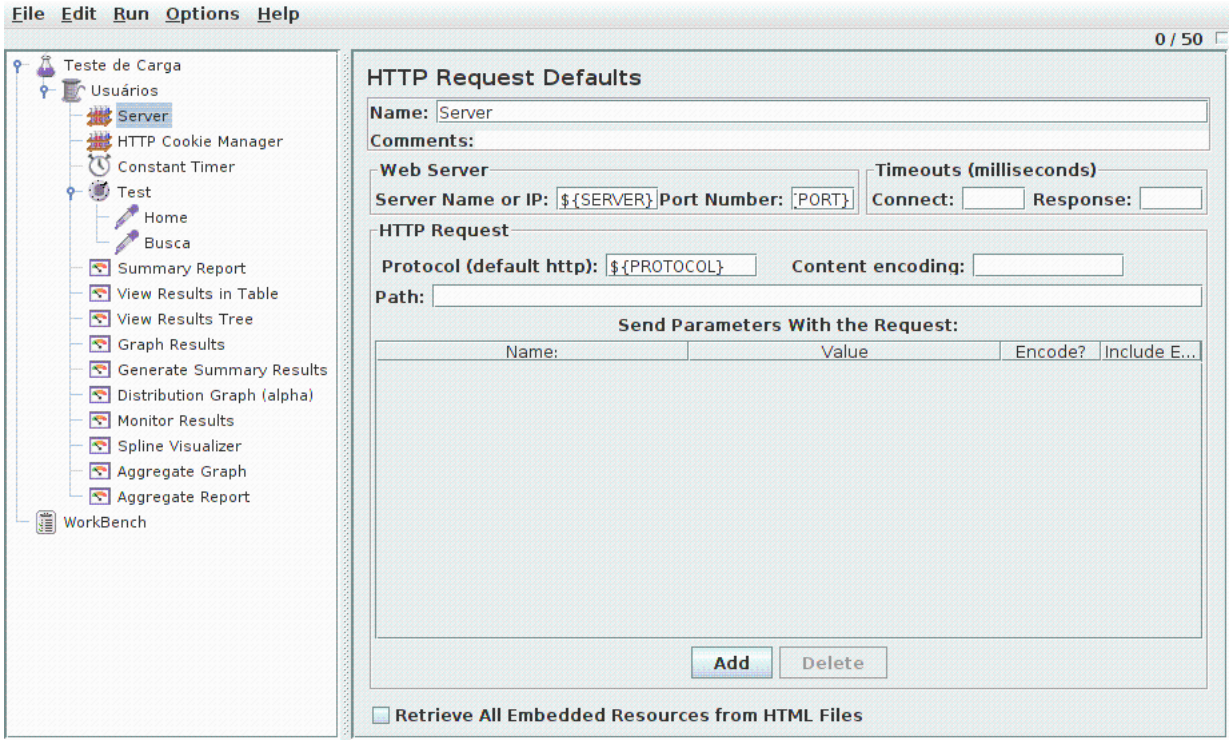


Figura A.3: Configurações padrões do servidor.

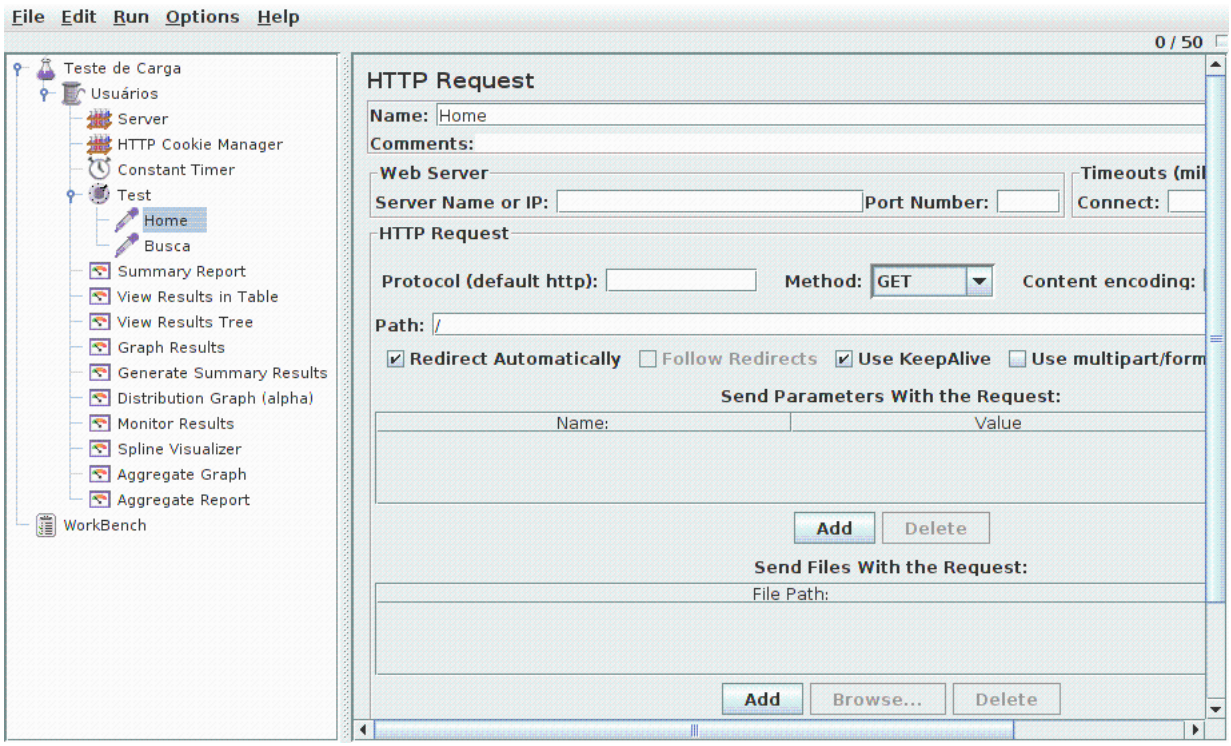


Figura A.4: Requisição HTTP GET na página inicial do sistema em teste.

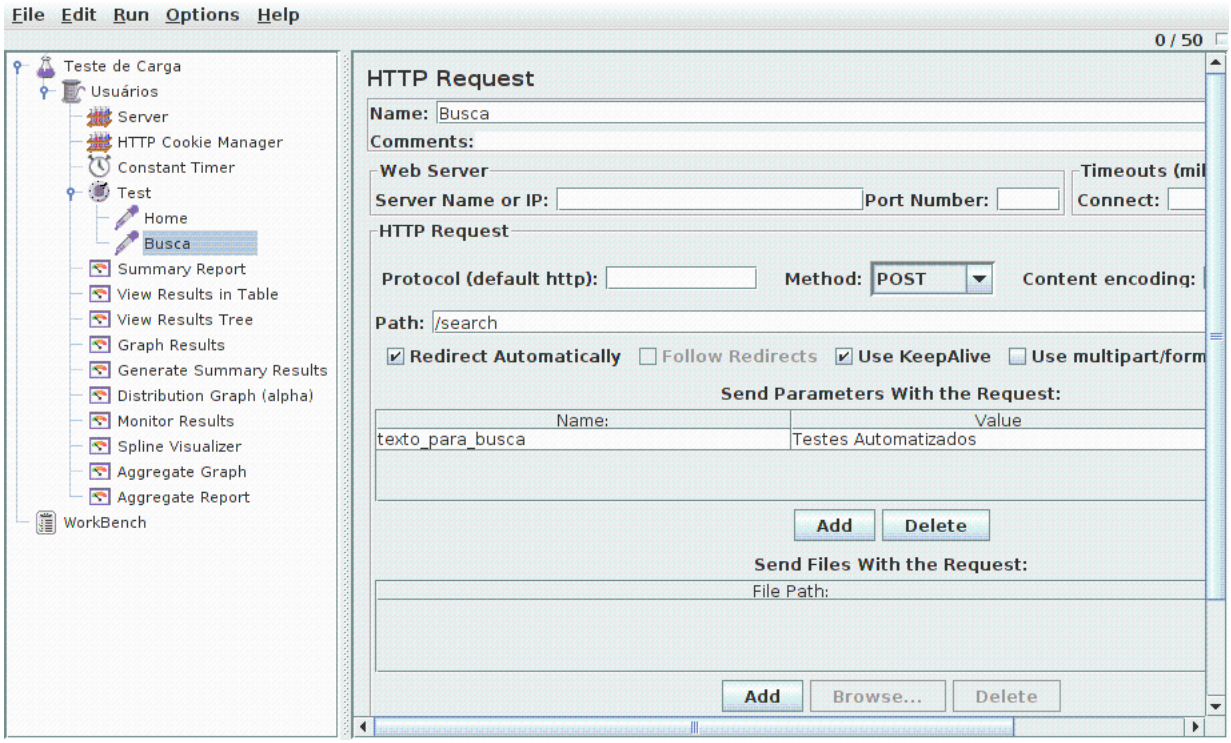


Figura A.5: Requisição HTTP POST para realizar uma busca no sistema.

mente automatizado se ele não coleta e exibe as informações pertinentes para análise, por isso, o JMeter também fornece diversos elementos que geram relatórios.

Por exemplo, a Figura A.6 mostra um gráfico onde é possível visualizar todos os tempos de resposta das requisições feitas pelos usuários. A partir desses valores, são calculados a média, mediana, desvio padrão e, também, a vazão que representa a quantidade de requisições que são suportadas por minuto.

Esse exemplo é bem simples, mas que pode ser bem útil para avaliar a capacidade da infraestrutura da sua aplicação. Para transformá-lo em um teste de estresse, basta aumentar a quantidade de usuários e de requisições até que o servidor caia ou o desempenho se torne insuportavelmente lento.

Os testes podem ser feitos para cada funcionalidade do sistema, ou, então, pode-se criar uma sequência de passos que é comum dos usuários fazerem. Apenas é importante ressaltar que as funcionalidades mais populares e mais pesadas do sistema devem ser priorizadas, ou seja, as que possuem mais risco de derrubar os servidores.

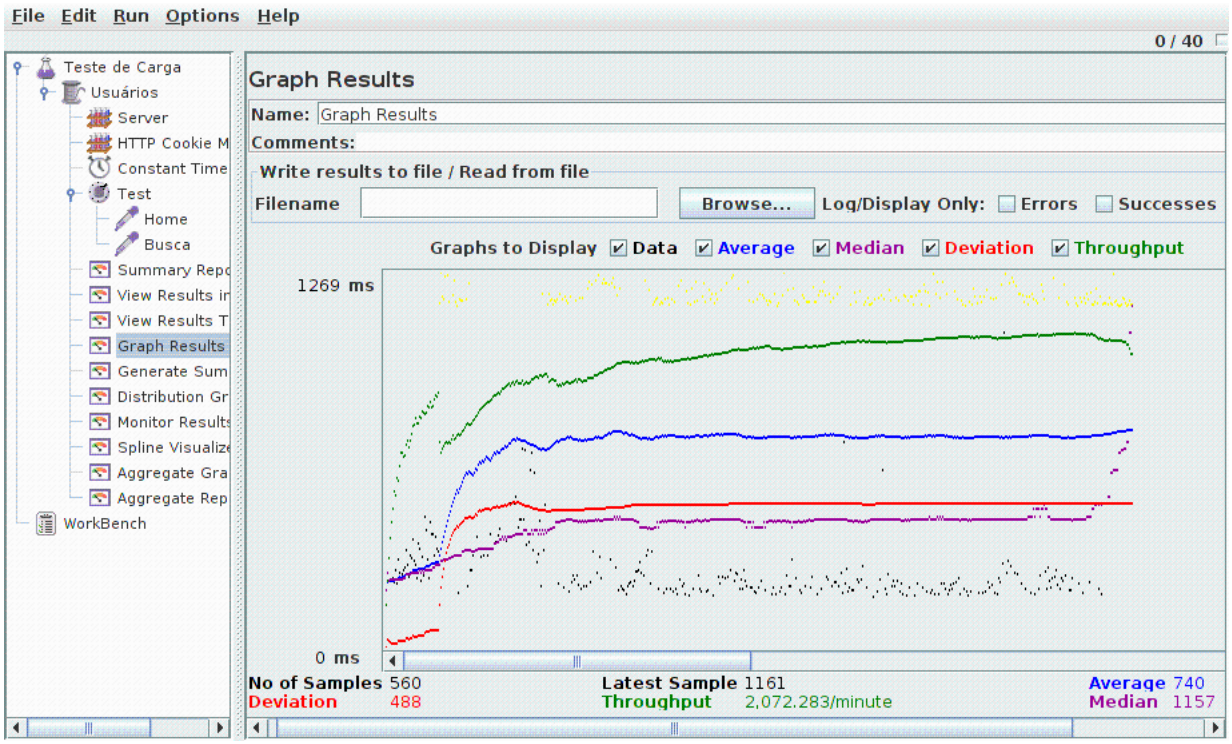


Figura A.6: Um dos gráficos que pode ser gerado pelo JMeter.

Apêndice B

Biblioteca CUnit

Na Seção 6.4.12 há um exemplo de testes criados com o arcabouço CUNit, agora, a Figura B.1 apresenta um esqueleto de como criar uma bateria (*suite*) de testes, ou seja, como cadastrar as funções que devem ser executadas pelo arcabouço (linhas 13 a 33).

O CUnit fornece várias maneiras de executar as baterias de testes. O modo convencional é a Interface Automatizada (linhas 42 a 44), que executa os testes sem intervenção humana e imprime os resultados em um arquivo XML. A Interface Básica (linhas 35 a 40) também inicia os testes automaticamente, mas permite executar individualmente baterias ou testes. Quanto aos resultados, eles são impressos no console, com quantidade de detalhes que pode ser configurada. Ainda há a Interface Interativa, a qual permite que o usuário controle o fluxo de execução dos testes.

```c
#include <stdio.h>
/* Referências do CUnit */
#include <CUnit/CUnit.h>
#include <CUnit/Basic.h>
#include <CUnit/Automated.h>

void test_um(void) { /* ... */ }
void test_dois(void) { /* ... */ }
void test_tres(void) { /* ... */ }

/* Execução dos testes com CUnit */
int main() {
   CU_pSuite suite = NULL;

   /* Inicializa registro de testes do CUnit */
   if (CUE_SUCCESS != CU_initialize_registry())
      return CU_get_error();

   /* Adiciona a suite de testes ao registro */
   suite = CU_add_suite("Suite", NULL, NULL);
   if (NULL == suite) {
      CU_cleanup_registry();
      return CU_get_error();
   }

   /* Adiciona casos de testes à suite de testes */
   if ((NULL == CU_add_test(suite, "test_um", test_um)) ||
       (NULL == CU_add_test(suite, "test_dois", test_dois)) ||
       (NULL == CU_add_test(suite, "test_tres", test_tres))
   ) {
      CU_cleanup_registry();
      return CU_get_error();
   }

   /* Executa todos os testes usando Interface Básica */
   CU_basic_set_mode(CU_BRM_VERBOSE);
   CU_basic_run_tests();
   printf("\n");
   CU_basic_show_failures(CU_get_failure_list());
   printf("\n\n");

   /* Executa todos os testes usando Interface Automatizada */
   CU_automated_run_tests();
   CU_list_tests_to_file();

   return CU_get_error();
}
```

Figura B.1: Biblioteca CUnit.

Referências Bibliográficas

[1] Gojko Adzic. *Bridging the Communication Gap: Specification by Example and Agile Acceptance Testing*. Neuri Limited, 2009.

[2] Scott W. Ambler. Test driven database design. *TASS Quarterly magazine*, page 4, September 2006. Toronto Association of Systems and Software Quality.

[3] Scott W. Ambler and Ron Jeffries. *Agile Modeling: Effective Practices for Extreme Programming and the Unified Process*. Wiley, 2002.

[4] Scott W. Ambler and Pramod J. Sadalage. *Refactoring Databases: Evolutionary Database Design*. Addison-Wesley, 2006.

[5] Prasanth Anbalagan and Tao Xie. Apte: automated pointcut testing for aspectj programs. In *WTAOP '06: Proceedings of the 2nd workshop on Testing aspect-oriented programs*, pages 27–32, New York, NY, USA, 2006. ACM.

[6] Ann Anderson, Ralph Beattie, Kent Beck, David Bryant, Marie DeArment, Martin Fowler, Margaret Fronczak, Rich Garzaniti, Dennis Gore, Brian Hacker, Chet Hen-drickson, Ron Jeffries, Doug Joppie, David Kim, Paul Kowalsky, Debbie Mueller, Tom Murasky, Richard Nutter, Adrian Pantea, and Don Thomas. Chrysler goes to extremes. *Distributed Computing*, 1(10):24–28, October 1998.

[7] Susan G. Archer, Laurel Allender, and Celine Richer. Software durability - is it important? can it be achieved? In *Proceedings of the Seventh International Conference on Human-Computer Interaction*, pages 593–596, 1997.

[8] Abel Avram and Floyd Marinescu. *Domain-Driven Design Quickly*. Lulu.com, 2007.

[9] Alberto Avritzer and Elaine J. Weyuker. Generating test suites for software load testing. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 44–57, 1994.

[10] Alberto Avritzer and Elaine J. Weyuker. The automatic generation of load test suites and the assessment of the resulting software. *IEEE Transactions on Software Engineering*, 21(9):705–716, September 1995.

[11] Liane Ribeiro Pinto Bandeira. Metodologia baseada em métricas de teste para indicação de testes a serem melhorados. Dissertação eletrônica, Biblioteca Digital de Teses e Dissertações da UFPE, Setembro 2008.

[12] Vitor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. The goal question metric. In *Encyclopedia of Software Engineering*, pages 528–532, 1996.

[13] Kent Beck. Simple smalltalk testing: With patterns. *First Class Software, Inc.*, 1994.

[14] Kent Beck. Make it run, make it right: Design through refactoring. *The Smalltalk Report*, 6(4):19–24, January 1997.

[15] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[16] Kent Beck. *Test-Driven Development: By Example*. Addison-Wesley, 2002.

[17] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change, 2nd Edition*. Addison-Wesley, 2004.

[18] Kent Beck et al. Manifesto for Agile Software Development. Home page: `http://agilemanifesto.org`, 2001.

[19] Kent Beck and Martin Fowler. *Planning Extreme Programming*. Addison-Wesley, 2001.

[20] Kent Beck and Mike Potel. *Kent Beck's Guide to Better Smalltalk*. Cambridge University Press, 1998.

[21] Boris Beizer. *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. Wiley, 1995.

[22] Yochai Benkler. Coase's Penguin, or Linux and the Nature of the Firm. *Computing Research Repository (CoRR)*, 2001.

[23] Yochai Benkler. *The Wealth of Networks: How Social Production Transforms Markets and Freedom*. Yale University Press, 2006.

[24] Mario Luca Bernardi and Giuseppe Antonio Di Lucca. Testing aspect oriented programs: an approach based on the coverage of the interactions among advices and methods. In *Quality of Information and Communications Technology, 2007. QUATIC 2007. 6th International Conference on the*, pages 65–76. IEEE Computer Society, 2007.

[25] Thirumalesh Bhat and Nachiappan Nagappan. Evaluating the efficacy of test-driven development: industrial case studies. In *ISESE '06: Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, pages 356–363, New York, NY, USA, 2006. ACM.

[26] Randolph Bias. Walkthroughs: Efficient collaborative testing. *IEEE Software*, 8(5):94–95, September 1991.

[27] Robert V. Binder. Design for testability in object-oriented systems. *CACM: Communications of the ACM*, 37(9):87–101, 1994.

[28] Robert V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Professional, 1999.

[29] Rex Black. *Pragmatic Software Testing: Becoming an Effective and Efficient Test Professional*. Wiley, 2007.

[30] Joshua Bloch. *Effective Java*. Prentice Hall PTR, 2008.

[31] Barry W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, pages 61–72, May 1988.

[32] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture, Volume 1, A System of Patterns*. Hardcover, 1996.

[33] Gerardo Canfora, Aniello Cimitile, Felix Garcia, Mario Piattini, and Corrado Aaron Visaggio. Evaluating advantages of test driven development: a controlled experiment with professionals. In *ISESE '06: Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, pages 364–371, New York, NY, USA, 2006. ACM.

[34] David Chelimsky, Dave Astels, Bryan Helmkamp, Dan North, Zach Dennis, and Aslak Hellesoy. *The RSpec Book: Behaviour Driven Development with RSpec, Cucumber, and Friends*. Pragmatic Bookshelf, 2009.

[35] Paulo Cheque and Fabio Kon. Desenvolvendo com agilidade: Experiências na reimplementação de um sistema de grande porte. In *Primeiro Workshop de Desenvolvimento Rápido de Aplicações (WDRA), realizado em conjunto com o VI Simpósio Brasileiro de Qualidade de Software*, 2007.

[36] Paulo Cheque and Fabio Kon. A importância dos testes automatizados: Controle ágil, rápido e confiável de qualidade. *Engenharia de Software Magazine*, 1(3):54–57, 2008.

[37] Tony Clear. The waterfall is dead: long live the waterfall!! *ACM SIGCSE (Special Interest Group on Computer Science Education) Bulletin*, 35(4):13–14, 2003.

[38] Alistair Cockburn. *Agile Software Development*. Addison-Wesley Longman, 2002.

[39] Alistair Cockburn and Laurie Williams. The costs and benefits of pair programming. In *Proceedings of the First International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2000)*, Cagliari, Sardinia, Italy, June 2000.

[40] Mike Cohn. *User Stories Applied: For Agile Software Development*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[41] Mike Cohn. *Agile Estimating and Planning*. Prentice Hall PTR, 2005.

[42] Microsoft Corporation. *Engineering Software for Accessibility*. Microsoft Press, 2009.

[43] Lisa Crispin and Janet Gregory. *Agile Testing: A Practical Guide for Testers and Agile Teams (Addison-Wesley Signature Series)*. Addison-Wesley, 2009.

[44] Lisa Crispin and Tip House. *Testing Extreme Programming*. Addison-Wesley, 2002.

[45] Philip B. Crosby. *Quality Is Free*. Mentor, 1980.

[46] Alexandre Freire da Silva. Reflexões sobre o ensino de metodologias ágeis na academia, na indústria e no governo. Master's thesis, Departamento de Ciência da Computação, Instituto de Matemática e Estatística - Universidade de São Paulo, Setembro 2007.

[47] Thomas H. Davenport and Jeanne G. Harris. *Competing on Analytics: The New Science of Winning*. Harvard Business School Press, 2007.

[48] Vieri del Bianco, Luigi Lavazza, Sandro Morasca, Davide Taibi, and Davide Tosi. The qualispo approach to oss product quality evaluation. In *FLOSS '10: Proceedings of the 3rd International Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development*, pages 23–28, New York, NY, USA, 2010. ACM.

[49] Esther Derby and Diana Larsen. *Agile Retrospectives: Making Good Teams Great*. Pragmatic Bookshelf, 2006.

[50] Edsger W. Dijkstra. The humble programmer. *CACM: Communications of the ACM*, 15, 1972.

[51] M. E. Drummond, Jr. A perspective on system performance evaluation. *IBM Systems Journal*, 8(4):252–263, 1969.

[52] Paul Duvall, Steve Matyas, and Andrew Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley Professional, 2007.

[53] Eduardo Martins Guerra. Um estudo sobre refatoração de código de teste. Master's thesis, Instituto Técnológico de Aeronáutica, 2005.

[54] Gerald D. Everett and Raymond McLeod Jr. *Software Testing*. John Wiley and Sons, Inc, 2007.

[55] Michael Feathers. *Working Effectively with Legacy Code*. Prentice Hall, 2008.

[56] Mark Fewster and Dorothy Graham. *Software Test Automation*. Addison-Wesley Professional, 1999.

[57] Dairton Luiz Bassi Filho. Experiências com desenvolvimento ágil. Master's thesis, Departamento de Ciência da Computação, Instituto de Matemática e Estatística - Universidade de São Paulo, Março 2008.

[58] Ira R. Forman and Nate Forman. *Java Reflection in Action*. Manning Publications, 2004.

[59] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[60] Steve Freeman and Nat Pryce. *Growing Object-Oriented Software, Guided by Tests*. Addison-Wesley Professional, 2009.

[61] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, 1995.

[62] David Gelperin and Bill Hetzel. The growth of software testing. *CACM: Communications of the ACM*, 31(6):687–695, 1988.

[63] Tom Gilb and Dorothy Graham. *Software Inspection*. Addison Wesley, 1993.

[64] Robert L. Glass. Persistent software errors. *IEEE Transactions on Software Engineering*, 7(2):162–168, March 1981.

[65] Robert L. Glass. The standish report: does it really describe a software crisis? 49(8):15–16, 2006.

[66] The Standish Group. The CHAOS report, 1994.

[67] The Standish Group. The CHAOS report, 2003.

[68] Atul Gupta and Pankaj Jalote. Test inspected unit or inspect unit tested code? In *Empirical Software Engineering and Measurement (ESEM)*, pages 51–60. IEEE Computer Society, 2007.

[69] Misko Hevery. Testability explorer: using byte-code analysis to engineer lasting social changes in an organization's software development process. In *OOPSLA Companion '08: Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 747–748, New York, NY, USA, 2008. ACM.

[70] Dorota Huizinqa and Adam Kolawa. *Automated Defect Prevention: Best Practices in Software Management*. Wiley-IEEE Computer Society Press, 2007.

[71] Andy Hunt and Dave Thomas. *Pragmatic Unit Testing in Java with JUnit*. The Pragmatic Programmers, 2003.

[72] The IEEE. IEEE standard for software reviews and audits. ANSI/IEEE STD 1028-1988, IEEE Computer Society, 1988.

[73] Melody Y. Ivory and Marti A Hearst. The state of the art in automating usability evaluation of user interfaces. *ACM Comput. Surv.*, 33(4):470–516, 2001.

[74] David Janzen. Software architecture improvement through test-driven development. In Ralph E. Johnson and Richard P. Gabriel, editors, *OOPSLA Companion*, pages 240–241. ACM, 2005.

[75] David S. Janzen and Hossein Saiedian. On the influence of test-driven development on software design. In *CSEET '06: Proceedings of the 19th Conference on Software Engineering Education & Training*, pages 141–148, Washington, DC, USA, 2006. IEEE Computer Society.

[76] Cem Kaner. Improving the maintainability of automated test suites. *Proceedings of the Tenth International Quality Week*, 1997.

[77] Cem Kaner, Jack Falk, and Hung Q. Nguyen. *Testing Computer Software*. Wiley, 1999.

[78] R. A. Khan and K. Mustafa. Metric based testability model for object oriented design (mtmood). *SIGSOFT Softw. Eng. Notes*, 34(2):1–6, 2009.

[79] Joshua Kierievsky. *Refactoring to Patterns*. Addison-Wesley Professional, 2001.

[80] Taeksu Kim, Chanjin Park, and Chisu Wu. Mock object models for test driven development. In *Software Engineering Research, Management and Applications, 2006. Fourth International Conference on*, pages 221–228. IEEE Computer Society, 2006.

[81] Donald Knuth. Structured programming with go to statements. *ACM Journal Computing Surveys*, 6(4), 1974.

[82] Lasse Koskela. *Test Driven: Practical TDD and Acceptance TDD for Java Developers*. Manning Publications, 2007.

[83] Mohit kumar, Akashdeep sharma, and Sushil Garg. A study of aspect oriented testing techniques. In *Industrial Electronics & Applications, 2009. ISIEA 2009. IEEE Symposium on*, pages 996–1001. IEEE Computer Society, 2009.

[84] Craig Larman and Victor R. Basili. Iterative and incremental development: a brief history. *IEEE Computer*, pages 47–56, July 2003.

[85] Otávio Augusto Lazzarini Lemos, Fabiano Cutigi Ferrari, Paulo Cesar Masiero, and Cristina Videira Lopes. Testing aspect-oriented programming pointcut descriptors. In Roger T. Alexander, Stephan Herrmann, and Dehla Sokenou, editors, *Workshop on Testing Aspect-Oriented Programs (WTAOP)*, pages 33–38. ACM, 2006.

[86] Otávio A. L. Lemos, José Carlos Maldonado, and Paulo Cesar Masiero. Teste de unidades de programas orientados a aspectos. In *Simpósio Brasileiro de Engenharia de Software*, 2004.

[87] Karl R. P. H. Leung and Wing Lok Yeung. Generating user acceptance test plans from test cases. In *COMPSAC*, pages 737–742. IEEE Computer Society, 2007.

[88] K. J. Lienberherr. Formulations and benefits of the law of demeter. 1989.

[89] Barbara Liskov. Keynote address - data abstraction and hierarchy. In *OOPSLA '87: Addendum to the proceedings on Object-oriented programming systems, languages and applications (Addendum)*, pages 17–34, New York, NY, USA, 1987. ACM.

[90] Henry H. Liu. *Software Performance and Scalability: A Quantitative Approach (Quantitative Software Engineering Series)*. Wiley, 2009.

[91] Cristina Videira Lopes and Trung Chi Ngo. Unit-testing aspectual behavior. In *In proc. of Workshop on Testing Aspect-Oriented Programs (WTAOP), held in conjunction with the 4th International Conference on Aspect-Oriented Software Development (AOSD'05)*, 2005.

[92] Kim Man Lui and Keith C.C. Chan. Test-driven development and software process improvement in china. In *Proceedings of the 5th International Conference on eXtreme Programming and Agile Processes in Software Engineering (XP 2004)*, volume 3092 of *Lecture Notes on Computer Science*, pages 219–222, 2004.

[93] Tim Mackinnon, Steve Freeman, and Philip Craig. *Endo-testing: unit testing with mock objects*, pages 287–301. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[94] Lech Madeyski. *Test-Driven Development: An Empirical Evaluation of Agile Practice*. Springer, 2009.

[95] José Carlos Maldonado, Márcio Eduardo Delamaro, and Mario Jino. *Introdução ao Teste de Software*. Campus, 2007.

[96] Robert C. Martin. The test bus imperative: Architectures that support automated acceptance testing. *IEEE Software*, 22(4):65–67, 2005.

[97] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, 2008.

[98] Deborah J. Mayhew. *The Usability Engineering Lifecycle: A Practitioner's Handbook for User Interface Design (Interactive Technologies)*. Morgan Kaufmann, 1999.

[99] Gerard Meszaros. *XUnit Test Patterns: Refactoring Test Code*. Addison-Wesley, 2007.

[100] Bertrand Meyer. *Object-Oriented Software Construction, Second Edition*. The Object-Oriented Series. Prentice-Hall, Englewood Cliffs (NJ), USA, 1997.

[101] James B. Michael, Bernard J. Bossuyt, and Byron B. Snyder. Metrics for measuring the effectiveness of software-testing tools. In *International Symposium on Software Reliability Engineering (ISSRE)*, pages 117–128. IEEE Computer Society, 2002.

[102] Rodrigo M. L. M. Moreira, Ana C. R. Paiva, and Ademar Aguiar. Testing aspect-oriented programs. In *Information Systems and Technologies (CISTI), 2010 5th Iberian Conference on*, pages 1–6. IEEE Computer Society, 2010.

[103] Tomer Moscovich and John F. Hughes. Indirect mappings of multi-touch input using one and two hands. pages 1275–1284. ACM, 2008.

[104] Rick Mugridge and Ward Cunningham. *Fit for Developing Software: Framework for Integrated Tests*. Prentice Hall, 2005.

[105] Glenford J. Myers. *The Art of Software Testing*. John Wiley and Sons, New York, 1979.

[106] Syed Asad Ali Naqvi, Shaukat Ali, and M. Uzair Khan. An evaluation of aspect oriented testing techniques. In *Emerging Technologies, 2005. Proceedings of the IEEE Symposium on*, pages 461–466. IEEE Computer Society, 2005.

[107] NIST. National institute of standards and technology, 2002.

[108] H. Ohba. Software quality = test accuracy * test coverage. In *International Conference on Software Engineering (ICSE)*, pages 287–295, 1982.

[109] Taiichi Ohno. *Toyota Production System: Beyond Large-Scale Production*. Productivity Press, 1998.

[110] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. Ph.D. thesis, University of Illinois, 1992.

[111] Behrooz Parhami. Defect, fault, error,..., or failure? In *Reliability, IEEE Transactions on*, volume 46, pages 450–451. IEEE Reliability Society, 1997.

[112] Mary Poppendieck and Tom Poppendieck. *Lean Software Development: An Agile Toolkit*. Addison-Wesley Professional, 2003.

[113] Dhanji Prasanna. *Dependency Injection*. Manning Publications, 2009.

[114] IEEE Press. Standard 610.12. IEEE standard glossary of software engineering terminology, 1990.

[115] Roger Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Science/Engineering/Math, 2009.

[116] Viera K. Proulx. Test-driven design for introductory oo programming. In *SIGCSE '09: Proceedings of the 40th ACM technical symposium on Computer science education*, pages 138–142, New York, NY, USA, 2009. ACM.

[117] Nat Pryce. *Growing Object-Oriented Software, Guided by Tests*. Addison-Wesley Professional, 2009.

[118] Vaclav Rajlich. Changing the paradigm of software engineering. *Communications of the ACM*, 49(8):67–70, August 2006.

[119] Reginaldo Ré, Otávio Augusto Lazzarini Lemos, and Paulo Cesar Masiero. Minimizing stub creation during integration test of aspect-oriented programs. In *WTAOP '07: Proceedings of the 3rd workshop on Testing aspect-oriented programs*, pages 1–6, New York, NY, USA, 2007. ACM.

[120] Stuart Reid. The art of software testing, second edition. glenford J. myers. *Softw. Test, Verif. Reliab*, 15(2):136–137, 2005.

[121] André Restivo and Ademar Aguiar. Towards detecting and solving aspect conflicts and interferences using unit tests. In *SPLAT '07: Proceedings of the 5th workshop on Software engineering properties of languages and aspect technologies*, page 7, New York, NY, USA, 2007. ACM.

[122] Don Roberts, John Brant, and Ralph E. Johnson. A refactoring tool for Smalltalk. *Theory and Practice of Object Systems (TAPOS)*, 3(4):253–263, 1997.

[123] Winston W. Royce. Managing the development of large software systems: concepts and techniques. In *ICSE '87: Proceedings of the 9th international conference on Software Engineering*, pages 328–338, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[124] David Saff and Michael D. Ernst. Can continuous testing speed software development? In *Fourteenth International Symposium on Software Reliability Engineering (ISSRE)*, pages 281–292, 2003.

[125] Goutam Kumar Saha. Understanding software testing concepts. *Ubiquity*, 2008(1):1, February 2008.

[126] Joc Sanders and Eugene Curran. *Software Quality*. Addison-Wesley, 1994.

[127] Danilo Sato, Alfredo Goldman, and Fabio Kon. Tracking the Evolution of Object Oriented Quality Metrics. In *Proceedings of the 8th International Conference on Extreme Programming and Agile Processes in Software Engineering (XP'2007)*, pages 84–92, 2007.

[128] Danilo Toshiaki Sato. Uso eficaz de métricas em métodos Ágeis de desenvolvimento de software. Master's thesis, Departamento de Ciência da Computação, Instituto de Matemática e Estatística - Universidade de São Paulo, Agosto 2007.

[129] Ulrich Schoettmer and Toshiyuki Minami. Challenging the 'high performance — high cost paradigm' in test. In *International Test Conference (ITC '95)*, pages 870–879, Altoona, Pa., USA, October 1995. IEEE Computer Society Press.

[130] Ken Schwaber. *Agile Project Management with Scrum*. Microsoft Press, 2004.

[131] Ken Schwaber and Mike Beedle. *Agile Software Development with SCRUM*. Prentice Hall, 2001.

[132] Mike Potel Sean Cotter. *Inside Taligent Technology*. Taligent Press, 1995.

[133] Helen Sharp, Yvonne Rogers, and Jenny Preece. *Interaction Design: Beyond Human-Computer Interaction*. Wiley, 2007.

[134] Walter Andrew Shewhart. Statistical method from the viewpoint of quality control. In *Dover Publications*, 1939.

[135] Sandro Silvestre. *Desenvolvimento de software robusto*. PhD thesis, Instituto de Pesquisas Tecnológicas do Estado de São Paulo - Universidade de São Paulo, Dezembro 2006.

[136] Diomidis Spinellis. *Code Quality: The Open Source Perspective*. Addison-Wesley Professional, 2006.

[137] Susan H. Strauss and Robert G. Ebenau. *Software Inspection Process*. McGraw-Hill, 1994.

[138] Dave Thomas and Andy Hunt. Mock objects. In *Software, IEEE*, volume 19, pages 22–24. IEEE Computer Society, 2002.

[139] Jenifer Tidwell. *Designing Interfaces*. O'Reilly Media, 2005.

[140] James E. Tomayko. A comparison of pair programming to inspections for software defect reduction. *Computer Science Education*, 12(3):213–222, 2002.

[141] Richard Torkar. Towards automated software testing - techniques, classifications and frameworks. Master's thesis, School of Engineering - Dept. of Systems and Software Engineering/Blekinge Institute of Technology, 2006.

[142] K. Vahidi and A. Orailoglu. Testability metrics for synthesis of self-testable designs and effective test plans. In *VTS '95: Proceedings of the 13th IEEE VLSI Test Symposium*, page 170, Washington, DC, USA, 1995. IEEE Computer Society.

[143] Arie van Deursen, Leon M. F. Moonen, Alexander van den Bergh, and Gerard Kok. Refactoring test code. Preprint, Centrum voor Wiskunde en Informatica, department Software Engineering (SEN), 2001.

[144] Auri Marcelo Rizzo Vincenzi, José Carlos Maldonado, Eric W. Wong, and Márcio Eduardo Delamaro. Coverage testing of java programs and components. *Sci. Comput. Program.*, 56(1-2):211–230, 2005.

[145] James A. Whittaker and Mike Andrews. *How to break Web software: functional and security testing of Web applications and Web services*. Addison-Wesley, 2006.

[146] Laurie Williams and Robert Kessler. *Pair Programming Illuminated*. Addison-Wesley, 2002.

[147] Laurie A. Williams, E. Michael Maximilien, and Mladen A. Vouk. Test-driven development as a defect-reduction practice. In *ISSRE*, pages 34–48. IEEE Computer Society, 2003.

[148] Stephen Withall. *Software Requirement Patterns (Best Practices)*. Microsoft Press, 2007.

[149] Yuk Kuen Wong. *Modern Software Review: Techniques and Technologies*. IRM Press, 2006.

[150] Tao Xie and Jianjun Zhao. Perspectives on automated testing of aspect-oriented programs. In *WTAOP '07: Proceedings of the 3rd workshop on Testing aspect-oriented programs*, pages 7–12, New York, NY, USA, 2007. ACM.

[151] Edward Yourdon. *Structured Walkthrough*. Prentice-Hall, 4th edition, 1989.

[152] Chuan Zhao and Roger T. Alexander. Testing aspect-oriented programs as object-oriented programs. In *WTAOP '07: Proceedings of the 3rd workshop on Testing aspect-oriented programs*, pages 23–27, New York, NY, USA, 2007. ACM.

[153] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *CSURV: Computing Surveys*, 29, 1997.