UNIVERSIDADE ESTADUAL DE CAMPINAS

FACULDADE DE TECNOLOGIA

NILO GIANNECCHINI

INTEGRAÇÃO CONTÍNUA COM APLICAÇÃO DE TESTES DE REGRESSÃO

Limeira, 2018

NILO GIANNECCHINI

INTEGRAÇÃO CONTÍNUA COM APLICAÇÃO DE TESTES DE REGRESSÃO

Dissertação apresentada à Faculdade de Tecnologia da Universidade Estadual de Campinas como parte dos requisitos exigidos para a obtenção do título de Mestre em Tecnologia, na Área de Sistemas de Informação e Comunicação

Orientadora: Profa. Dra. Regina Lúcia de Oliveira Moraes

ESTE EXEMPLAR CORRESPONDE À VERSÃO FINAL DA DISSERTAÇÃO DEFENDIDA PELO ALUNO NILO GIANNECCHINI, E ORIENTADA PELA PROFA. DRA. REGINA LÚCIA DE OLIVEIRA MORAES

Limeira, 2018

Agência(s) de fomento e nº(s) de processo(s): Não se aplica.
ORCID: https://orcid.org/0000-0002-6461-9481

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca da Faculdade de Tecnologia
Felipe de Souza Bueno - CRB 8/8577

Giannecchini, Nilo, 1988-
G348i Integração contínua com aplicação de testes de regressão / Nilo Giannecchini. – Limeira, SP : [s.n.], 2018.

Orientador: Regina Lúcia de Oliveira Moraes.
Dissertação (mestrado) – Universidade Estadual de Campinas, Faculdade de Tecnologia.

1. Integração contínua. 2. Testes de regressão. I. Moraes, Regina Lúcia de Oliveira, 1956-. II. Universidade Estadual de Campinas. Faculdade de Tecnologia. III. Título.

Informações para Biblioteca Digital

Título em outro idioma: Continuous integration with regression test application
Palavras-chave em inglês:
Continuous integration
Regression tests
Área de concentração: Sistemas de Informação e Comunicação
Titulação: Mestre em Tecnologia
Banca examinadora:
Regina Lúcia de Oliveira Moraes [Orientador]
Tânia Basso
Paulo Sérgio Martins Pedro
Data de defesa: 09-05-2018
Programa de Pós-Graduação: Tecnologia

FOLHA DE APROVAÇÃO

Abaixo se apresentam os membros da comissão julgadora da sessão pública de defesa de dissertação para o Título de Mestre em Tecnologia na área de concentração de Sistemas de Informação e Comunicação, a que submeteu a (o) aluna (o) Nilo Giannecchini, em 09 de maio de 2018 na Faculdade de Tecnologia- FT/ UNICAMP, em Limeira/SP.

Prof. (a). Dr. (a) Regina Lucia de Oliveira Moraes

FT – UNICAMP - Presidente da Comissão Julgadora

Prof. (a). Dr. (a) Tania Basso

ISCA - Instituto Superior de Ciências Aplicadas

Prof. Dr. Paulo Sérgio Martins Pedro

FT – UNICAMP

A Ata da defesa com as respectivas assinaturas dos membros encontra-se no processo de vida acadêmica da aluna na Universidade.

RESUMO

À medida que a complexidade de um projeto aumenta, a necessidade de integrar códigos de diferentes membros de uma equipe cresce. No ciclo de desenvolvimento de sistemas de *software* complexos, um dos pontos de maior risco é a integração de modificações independentes feitas por diferentes membros de uma equipe. Nesse sentido, a adoção de técnicas de teste em um ambiente de integração, visando minimizar falhas no momento da entrega do sistema de *software* tem emergido. Além disso, um maior número de aplicações e soluções para apoiar o desenvolvimento de sistemas em equipe tem sido disponibilizadas, facilitando o desenvolvimento colaborativo.

Esta dissertação de mestrado apresenta uma arquitetura para incluir a execução de testes em um ambiente de integração contínua, para facilitar a construção dos produtos de *software* por uma equipe composta por vários membros. O objetivo é propor uma solução simples para aplicar uma bateria de diversos tipos de testes em um ambiente integrado, capaz de identificar de maneira contínua, quando alterações no sistema ocorrerem. Esta dissertação incluiu na arquitetura apresentada os testes unitários (por exemplo, teste de classes), testes de integração (por exemplo, testes de componentes e de combinação de componentes) e testes de regressão (por exemplo, teste que verifica se novas modificações interferem nos requisitos já implementados).

A aplicação da arquitetura em um ambiente real de desenvolvimento mostrou, como resultado, uma diminuição significativa de falhas identificadas no momento de entrega de um sistema, quando comparamos com uma cópia do mesmo sistema que não foi desenvolvido com apoio da arquitetura. Essa melhoria ocorreu porque a maioria das falhas foram identificadas e corrigidas ao longo do processo, aumentando a confiabilidade do sistema.

Espera-se que as equipes que adotem a solução proposta tenham como resultado uma melhora na detecção de falhas decorrentes das alterações realizadas pelos vários desenvolvedores. Além disso, um guia de boas práticas, que possam reduzir as dificuldades de implementação, adaptação e manutenção desse ambiente no futuro foi disponibilizado com livre acesso pelos desenvolvedores interessados na adoção da solução.

ABSTRACT

As the complexity of a project increases, the need to integrate code from different members of a team grows. In the development cycle of complex software systems, one of the riskiest points is the integration of independent modifications made by different members of a team. In this sense, the adoption of testing techniques in an integration environment, aiming to minimize failures in the delivery of the software system has emerged. In addition, a greater number of applications and solutions to support the development of team systems have been made available, facilitating collaborative development.

This research presents an architecture to include the execution of tests in a continuous integration environment, to facilitate the construction of the software products by a team composed of several members. The goal is to propose a simple solution to apply a battery of several types of tests that is able to continuously identify when changes in the system occur. The focus of this dissertation is an architecture of an environment that includes the unit tests (for example, test of classes), integration tests (for example, component and component combination tests) and regression tests (for example, test that verifies if new modifications interfere in the Requirements already implemented).

The application of the architecture in a real environment of development showed, as a result, a significant decrease of failures that was identified at the time of delivery of a system, when compared to a copy of the same system that was not developed with the support of the architecture. This improvement occurred because most faults were identified and corrected throughout the process, increasing the system reliability.

It is expected that teams adopting the proposed solution will get improved failure detection due to changes made by various developers. In addition, a guide to best practices that could reduce the difficulties of implementing, adapting and maintaining this environment in the future was made freely available by developers interested in adopting the solution.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Em um ambiente de desenvolvimento em equipe, cada desenvolvedor realiza um conjunto de tarefas que, normalmente, tem relação com as tarefas que outros desenvolvedores estão realizando. Integrar a nova versão de todos os envolvidos pode ser uma tarefa complexa. Quanto maior a equipe, maior é o desafio. Em um projeto de grande porte, o processo de integração do trabalho de várias pessoas e equipes pode consumir um tempo significativo do projeto, impactando cronograma, custo e recursos humanos.

Esperar até o final do projeto para fazer essa integração pode levar a todo tipo de problemas de qualidade. Para evitar esses problemas, a integração deve ser realizada continuamente, fazendo com que os riscos sejam gerenciados mais rapidamente e em pequenos incrementos. Caso ocorra algum problema, a equipe deve ser alertada rapidamente e procedimentos podem ser configurados para evitar que códigos incorretos sejam incluídos na versão de produção [1].

De acordo com a empresa Gartner [2] as empresas investem em torno de 40% de seus orçamentos de TI em projetos de integração. Quando bem-sucedidos, há uma diminuição dos custos e tempo de desenvolvimento, um retorno sobre investimento mais rápido e maior produtividade. Além disso, colaboram ainda para atingir as principais metas de *e-business* e aprimoram relacionamentos com os clientes, parceiros e funcionários.

Atualmente, 50% das empresas utilizam alguma forma de integração contínua no desenvolvimento de software, número baixo levando-se em conta a redução de custos que toda automação traz [3]. Além da baixa adoção, a baixa maturidade também impede a completa capitalização das vantagens da Integração Contínua, não havendo dados que garantam a evolução desse número. Embora o mercado esteja se tornando mais ágil no desenvolvimento de software, a utilização da Integração Contínua não é uma realidade, mesmo sendo ela uma vantagem competitiva.

A Integração Contínua foi definida por Fowler [4] como uma prática de desenvolvimento de *software* onde os membros de um time integram seu trabalho frequentemente. Essa prática pode ser feita através de processos manuais ou automatizados [5]. Geralmente, o desenvolvedor integra o código alterado e/ou desenvolvido ao projeto principal na mesma frequência com que as funcionalidades são desenvolvidas, podendo ser feita muitas vezes ao invés de apenas uma vez ao dia. Cada integração é verificada por uma construção do *software* (incluindo testes) para detectar defeitos de integração o mais rápido possível. O objetivo principal é verificar se as alterações ou novas funcionalidades não criaram novos defeitos no projeto já existente. Muitos

times acham que essa abordagem leva a uma significativa redução dos problemas de integração e permite que um time desenvolva *software* coeso mais rapidamente.

Como resultado da execução de uma integração bem-sucedida, consegue-se garantir que o código compila e que as funcionalidades testadas mantêm sua integridade e funcionamento conforme o esperado, além de uma potencial versão do *software* estável, que pode ser utilizada [6].

Para que a integração seja efetiva, é necessário verificar o funcionamento do código após as modificações. Para atender a essa necessidade, o desenvolvimento em conjunto com a realização de testes é fundamental [4]. Dessa forma, um processo de integração contínua passa, obrigatoriamente, pela integração de testes nesse processo e, se possível, que eles sejam automaticamente invocados quando alterações nos componentes do sistema acontecem. Essa abordagem está em evolução e esta proposta vem contribuir para que os diferentes tipos de testes sejam integrados no processo, de forma que todos sejam executados automaticamente a cada alteração de código integrada.

1.1 JUSTIFICATIVA

A importância do desenvolvimento de um ambiente de integração contínua adaptável e capaz de executar um conjunto de testes durante o processo de desenvolvimento do sistema é importante para os ambientes de desenvolvimento modernos. Definir e implementar tal ambiente, onde testes unitários, testes de integração e testes de regressão são disparados automaticamente quando uma alteração é efetivada, motivou essa pesquisa. Não foram encontrados um processo integrado e um conjunto de ferramentas que se adaptem às novas tecnologias de desenvolvimento. Dessa forma, a adoção de um ambiente integrado que acomode as linguagens utilizadas e os testes desenvolvidos fica dificultada. Particularmente, para o desenvolvimento em plataforma web utilizando-se Java Script, não foi encontrado uma definição adequada e este trabalho vem contribuir nessa direção.

Com a proposta desta arquitetura de referência para a integração contínua procurou-se trazer benefícios aos desenvolvedores e organizações que desejam melhorar a velocidade de desenvolvimento e antecipar a correção de falhas. Um dos requisitos é manter alto desempenho na integração da produção da equipe e na execução dos testes automatizados.

Segundo nossa avaliação, este trabalho contribui para aumentar e facilitar a utilização da integração contínua, conforme a arquitetura aqui proposta que inclui a execução dos testes automáticos.

1.2 OBJETIVO

Identificado o problema, o objetivo do presente trabalho foi propor uma arquitetura de referência para a integração contínua em um ambiente de desenvolvimento colaborativo, permitindo que se realize uma bateria dos diversos tipos de testes em um ambiente integrado. Fazendo a monitoração da versão dos componentes, o ambiente deve ser capaz de identificar quando houver alterações no sistema, seja por evolução, ou por manutenção, e aplicar o conjunto de testes de forma automática.

Embora ambientes de integração contínua já existam e sejam utilizados nas empresas que desenvolvem software, a proposta deste trabalho formaliza esse ambiente e complementa a automatização para ambientes de aplicações Web.

Para alcançar este objetivo foram traçados alguns objetivos secundários tais como:

- Estudar técnicas de *Test Driven Development* (TDD) [7] onde o desenvolvedor escreve um caso de teste automatizado que define uma melhoria ou uma nova funcionalidade e então, é produzido código que possa ser validado pelo teste;
- Estudar técnicas de *Behavior-driven development* (BDD) [8] que é uma técnica de desenvolvimento que encoraja a colaboração entre desenvolvedores, setores de qualidade e pessoas não-técnicas ou de negócios em um projeto de *software* visando o desenvolvimento dos testes;
- Explorar ferramentas de integração contínua com o intuito de estender o mecanismo de realização dos testes automáticos;
- Estudar as ferramentas existentes que auxiliam no desenvolvimento e execução dos testes automáticos, uma vez que a ferramenta deve ser executada sobre o sistema em tempo de desenvolvimento;
- Estudar como melhorar o desempenho dos testes regressivos em um ambiente de integração contínua;
- Definir formas de validação da arquitetura como, por exemplo, a utilização do tempo na integração contínua (quanto menor o tempo maior a eficiência) e o aumento na detecção de falhas (quanto mais cedo se detectam as falhas mais eficiente é o processo), além de um questionário para colher *feedback* e melhorar o processo de adoção;

Para a validação da arquitetura proposta neste trabalho foram feitas comparações de indicadores extraídos do ambiente proposto e um ambiente tradicional que não utilizou integração contínua.

1.3 CONTRIBUIÇÕES DO TRABALHO

As contribuições deste trabalho se insere na melhoria dos processos de desenvolvimento em um ambiente colaborativo para plataforma web, utilizando-se Java Script. São elas:

- Uma arquitetura de referência que define os componentes de um ambiente de integração que incorpore testes unitários, de integração e de regressão que são executados de forma automática quando alterações são integradas ao sistema;
- Uma implementação do ambiente de integração que apresenta, com base em um conjunto de ferramentas escolhidas, a arquitetura de referência proposta;
- Uma avaliação de uma instância da arquitetura de referência, para efeito de comparação de resultados, utilizando-se dois sistemas reais, que foram desenvolvidos no ambiente integrado e no tradicional respectivamente;
- Um guia de boas práticas que, ao ser utilizado, auxilie na redução das dificuldades de implementação, adaptação e manutenção desse ambiente no futuro.

1.4 ORGANIZAÇÃO DO TEXTO

Este trabalho foi dividido em capítulos e encontra-se estruturado da seguinte forma:

Além do **Capítulo 1** que traz a *Introdução* e apresentou o contexto em que se insere a pesquisa, ressaltou sua importância, apresentando os objetivos e justificativas para o desenvolvimento do trabalho, o **Capítulo 2** - *Background* apresenta conceitos fundamentais necessários para desenvolver a pesquisa, sendo abordados temas como: teste de *software*, integração contínua, metodologia de desenvolvimento de teste e arquitetura de integração contínua. Já o **Capítulo 3** - *Trabalhos Relacionados e Ferramentas de Apoio* apresenta trabalhos que tenham intersecção com os objetivos da pesquisa que foi desenvolvida. O **Capítulo 4** – *Integração contínua com testes de regressão* aborda o conceito de integração contínua e as ferramentas que se relacionam com o tema. Também foi descrita a arquitetura para o desenvolvimento da abordagem proposta. O **Capítulo 5** - *Resultados e Discussão* apresenta os resultados da pesquisa e as evidencias dos testes realizados. Também traz um guia de boas práticas para apoiar a implantação de um ambiente semelhante em outras organizações. Finalmente, o **Capítulo 6** – *Conclusões e Trabalhos Futuros* apresenta nossas conclusões e propostas para trabalhos futuros.

2 BACKGROUND

Ao longo deste capítulo serão descritos conceitos importantes para facilitar o entendimento deste trabalho. Os conceitos que serão apresentados são: (i) integração contínua; (ii) teste de *software*, apresentando esta fase do ciclo de desenvolvimento de *software*; (iii) metodologia de desenvolvimento de teste e técnicas para auxiliar no processo de teste de *software*.

2.1 INTEGRAÇÃO CONTÍNUA

A integração contínua (IC) é uma técnica que visa fazer com que o *software* esteja finalizado no momento em que o último pedaço de código é concluído [4]. Entretanto, a implementação dessa técnica não é das mais simples. Exige um conhecimento profundo por parte de quem a está implementando, além de uma boa vontade da equipe de desenvolvimento em adota-la.

O principal fundamento da IC é facilitar a manutenção do código. Entretanto, muito mais pode ser feito com a IC. A IC, basicamente, deve construir um executável (*build*), testar, analisar e implementar uma aplicação para garantir que o processo inteiro esteja funcionando e seguindo as melhores práticas de desenvolvimento [9].

Esse processo deve ser executado a cada alteração no código (após cada *commit* efetuado por um membro da equipe), permitindo uma resposta imediata aos membros do time. Além disso, pode ser usado como uma ferramenta de monitoração do desenvolvimento, possibilitando acompanhar o que outros membros da equipe estão realizando e se estão realizando as tarefas da maneira correta.

Esse tipo de procedimento permite também que a IC seja utilizada como uma forma de gerenciamento, mensurando o tempo gasto pelo time de desenvolvimento em alterações no código, ajudando a definir custos e melhorias nos processos envolvidos [9].

A implementação da integração contínua pode ser dividida em sete ciclos, começando por uma empresa que não possui nenhum tipo de servidor centralizado até uma em que a integração contínua está implementada seguindo todas as boas práticas. Esses ciclos são determinados por um processo de maturação da empresa, são eles [10]:

1. Nesse ciclo, não há um servidor centralizado de código, e todos os builds são realizados na máquina local do desenvolvedor;
2. A criação de um servidor centralizado foi realizada, e agora os desenvolvedores podem realizar *commits* de suas alterações nesse servidor;

3. Já existem alguns testes automatizados e o time começa a tirar proveito de algumas características da integração contínua. O servidor também possui a capacidade de alertar os membros da equipe sobre alterações realizadas;
4. A verificação de qualidade começa a ser implementada nesse ciclo, seguindo métricas comprovadamente eficientes, sem que isso acarrete grandes problemas de integração de novos membros;
5. A confiança no resultado dos *builds* automatizados é bastante grande, e é possível a utilização de técnicas como a TDD (*Test-Driven Development* – Desenvolvimento Orientado a Testes) [7];
6. A implementação do sexto ciclo envolve a implementação dos servidores para testes de aceitação (teste feito pelo cliente) e implementação, preparando a empresa para a filosofia da entrega contínua;
7. A confiança em todos os testes automatizados é completa, e pode ser permitido ao servidor de IC implementar técnicas automatizadas diretamente na produção.

Vemos que a integração contínua envolve a adaptação da equipe de desenvolvimento a uma série de processos. Entretanto, segundo a Forrester Consulting [10], uma vez que a equipe está adaptada, as vantagens que essa técnica traz são muito maiores que eventuais problemas encontrados.

A qualidade do *software* estará garantida ao final do desenvolvimento, o que significa um ganho muito grande em produtividade. Esse ganho está relacionado ao fato de que a aplicação de testes de aceitação ocupa boa parte do esforço de desenvolvimento de *software*. Se esses testes são aplicados continuamente, parte desse tempo será poupado ao final, fazendo cumprir ou até diminuindo o tempo de projeto [10].

2.2 TESTE DE *SOFTWARE*

Os testes de *software* têm a finalidade de garantir a qualidade do produto desenvolvido. Mesmo realizando testes, é impossível garantir que o produto que foi alvo dos testes fique isento de falhas [11], porém os testes aumentam as chances de que isso aconteça. De acordo com Meyers et al. [11] um caso de teste bem-sucedido é aquele que é capaz de revelar a presença de defeitos no produto. Assim, definir um conjunto de casos de teste que tenha alta probabilidade de encontrar defeitos é crucial para o processo de teste de *software*. As falhas podem surgir por diversos motivos, a especificação do projeto / da atividade pode estar incompleta ou incorreta, o tamanho do projeto pode exigir uma equipe grande e gerar conflitos no desenvolvimento ou limitações específicas que podem tornar mais complexo o produto, entre outros motivos. O

objetivo dos testes de *software* é identificar os defeitos no produto, sendo necessário procedimentos complementares para rastrear falhas que são responsáveis pelos defeitos apontados [12, Cap. 1].

Uma classificação utilizada é baseada na visibilidade que o testador tem da estrutura interna do *software* que está sendo testado. A técnica caixa branca, também conhecida como teste estrutural, utiliza a estrutura interna do programa para derivar os casos de teste. Deve garantir que os caminhos lógicos do *software* foram testados, exercitando decisões lógicas, loops, definições e usos de variáveis [13]. Pode ser executado em todas as etapas de desenvolvimento, mas é mais comum nas etapas iniciais quando são feitos os testes de unidades.

A técnica conhecida como teste caixa preta considera o sistema como uma caixa em que não é possível conhecer detalhes internos e só é possível visualizar as entradas e saídas. Nessa técnica, os casos de teste são projetados de acordo com a especificação do sistema, sem se preocupar com a implementação [13].

A técnica de teste de caixa cinza é uma mescla do uso das técnicas de caixa branca e de caixa preta. Isso significa que se tem acesso parcial às estruturas de dados e algoritmos do componente para que se possa desenvolver os casos de teste, que são executados como na técnica de teste caixa preta [14].

As próximas subseções apresentam os testes de *software* utilizados em função do estágio do ciclo de vida do *software* e que estão separados pelo nível dos componentes testados.

2.2.1 TESTE UNITÁRIO

O teste unitário tem por objetivo validar funções, classes ou pequenos componentes [15]. Normalmente, são utilizadas técnicas de teste caixa branca, isso é, testes estruturais. Corresponde ao primeiro nível de teste e deve ser executado durante e após o termino da codificação das unidades de *software*. Este nível de teste tem como objetivo garantir que o item sob teste cumpra a função descrita na sua especificação e aponte defeitos, ou seja, comportamentos que desviem daquele que é esperado e que são causados por falhas na lógica de programação.

Meszaros [16] se refere ao termo *Test Double* como um conjunto de objetos que pode ser utilizado para substituir uma classe de produção ou um conjunto delas durante os testes. Sendo assim, para garantir que uma classe está apresentando um comportamento correto pode-se testar seus métodos de maneira isolada com o uso de *Test Double*, podendo assim remover a dependência de recursos externos. Isto significa que o uso de *Test Doubles* permite que

funcionalidades que até então deveriam ser avaliadas por testes de integração sejam avaliadas por testes unitários. Esta afirmação é especialmente relevante dentro do contexto da integração contínua, onde a execução de testes unitários e de integração precisa ser separada, principalmente, devido ao tempo de execução.

Os *Test Doubles* são categorizados em diferentes tipos, a saber:

- ***Dummy***: São objetos que servem simplesmente para preencher parâmetros de métodos. Atendem aos cenários onde deseja-se chamar um dado método, mas os objetos sendo passados por parâmetro são irrelevantes, não precisando ser corretamente construídos;
- ***Fake***: É uma implementação do código real, mas que não atende aos propósitos de produção, servindo apenas para teste (por exemplo, um serviço que obtenha uma lista de todas as contas de um usuário);
- ***Stub***: Um trecho de código usado para substituir algumas outras funcionalidades podendo simular o comportamento de um código existente (como um procedimento em uma máquina remota) ou ser um substituto temporário para o código ainda a ser desenvolvido, sempre retornando o mesmo valor;
- ***Driver***: Operações que automatizam testes de acordo com casos de teste, similar ao *stub*;
- ***Spy***: Provê um retorno determinístico apresentando valores diferentes dependendo da forma pela qual foi chamado, ou seja, pode se utilizar uma chamada de integração real onde essa função está com *mock* para um determinado retorno específico;
- ***Mock***: Esta opção permite que um dado método em teste possa ser programado para, durante a execução do teste, realizar um comportamento diferente do original (simular um comportamento real), ou seja, pode-se programar diversos cenários de resposta do *mock* para testar diversos cenários reais, como por exemplo, integrações off-line, latência na rede, etc. Isso é útil, por exemplo, para ignorar chamadas a recursos externos quando se está testando um código que não faz uso de tais recursos, mas cujo acoplamento existe por questões de design da classe.

O uso de *Test Doubles* é comum para alcançar o isolamento entre classes, necessário para os testes unitários e de integração. Porém, em sistemas construídos utilizando-se linguagens orientadas a objetos, diversas classes se relacionam e seus métodos dependem de serviços

realizados por métodos de outras classes, o que dificulta o isolamento da mesma de dependências externas. Essas dependências causam as chamadas entradas e saídas indiretas [17].

Os testes unitários são comumente empregados na checagem de métodos, classes e transições de estados do SUT (*System Under Test*), sendo que [16]:

- São automatizados e repetíveis;
- Podem ser implementados facilmente;
- Uma vez escritos, os testes devem ser mantidos para reuso futuro;
- Qualquer profissional envolvido com o desenvolvimento de uma aplicação deve ser capaz de executá-los;
- Facilmente acionáveis, geralmente a partir de um botão ou item de menu dentro de uma IDE (*Integrated Development Environment*);
- São de execução rápida.

As técnicas do teste unitário têm como prerrogativa o uso da estrutura do código implementado para avaliar o comportamento do componente de *software*, ou seja, elabora-se casos de teste que cubram todos os caminhos de execução com base na estrutura do código gerado. Dessa maneira, todas as variações originadas por estruturas de condições ou repetições são exercitadas [13], visando atender critérios de cobertura, tais como:

i. Teste de condição, onde o teste examina os lados verdadeiros ou falsos de condições booleanas [18];
ii. Teste de fluxo de dados, verifica o comportamento das variáveis até localizar algum defeito em sua propagação durante a execução do código fonte [18];
iii. Teste de ciclos, onde concentra-se na validação da construção de ciclos [15] que podem ser classificados como: ciclos simples, concatenados, aninhados e desestruturados;
iv. Teste de caminhos lógicos, onde calcula-se a complexidade lógica do *software* e utiliza esta medida como base para descobrir os caminhos básicos do *software*, exercendo o teste de modo que todos os caminhos sejam exercitados [15].

Os testes unitários trazem como benefícios [12, Cap. 3]:

- A garantia de que problemas sejam descobertos ainda durante o processo de desenvolvimento das unidades;
- A facilidade na manutenção de código, especialmente na refatoração;
- A simplificação da integração de módulos;

- A utilização como artefato do projeto, apresentando os detalhes da funcionalidade.

Os testes unitários representam, inclusive, a base para o "Desenvolvimento Guiado por Testes" (*Test Driven Development* - TDD) [7], forma de desenvolvimento que tem ganho bastante espaço, sobretudo com a crescente popularidade de metodologias ágeis como *Scrum*.

TDD é um processo para desenvolvimento de *software* que enfatiza a construção de soluções com base em casos de testes previamente desenvolvidos. Dessa forma, fica fácil a integração com uma ferramenta de automação de testes unitários [7].

A escolha do TDD como uma prática de desenvolvimento implica, obrigatoriamente, na codificação dos testes unitários antes mesmo da escrita das partes da aplicação que serão submetidas a esses testes. Por mais que um teste possa ser formulado após a codificação de uma funcionalidade, isto não é adotado em projetos em conformidade com os princípios de TDD. A justificativa é que se tal teste fosse elaborado após o desenvolvimento do recurso a ser verificado, o mesmo poderia ser concebido de uma maneira incorreta, considerando apenas a possibilidade de execução com sucesso da função.

A implementação de um projeto baseado em técnicas de TDD é feita seguindo um ciclo conhecido como *Red-Green-Refactor*, em que cada um destes termos correspondendo a uma fase na construção dos testes [19]:

Red: o teste é escrito logo no início do desenvolvimento, com a funcionalidade-alvo sequer implementada (normalmente, apenas um "esqueleto" existirá, procurando se aproximar da estrutura esperada para a função em questão). O objetivo principal é que esse teste realmente falhe (daí o nome "*Red*", um sinal "vermelho" para um problema), cumprindo a meta de evitar uma verificação "viciada", que sempre resultaria em sucesso na sua execução;

Green: nesta fase é efetuada a codificação da forma mais simples possível, atendendo àquilo que se espera para uma funcionalidade, além de garantir que os testes associados serão executados com sucesso (sinal "verde" indicando que não existem problemas);

Refactor: com os testes tendo passado na etapa anterior, é possível refatorar o item sob análise, eliminando prováveis duplicações e melhorando a estrutura do código. Se este não for o caso, o comum é que se passe para os próximos testes unitários.

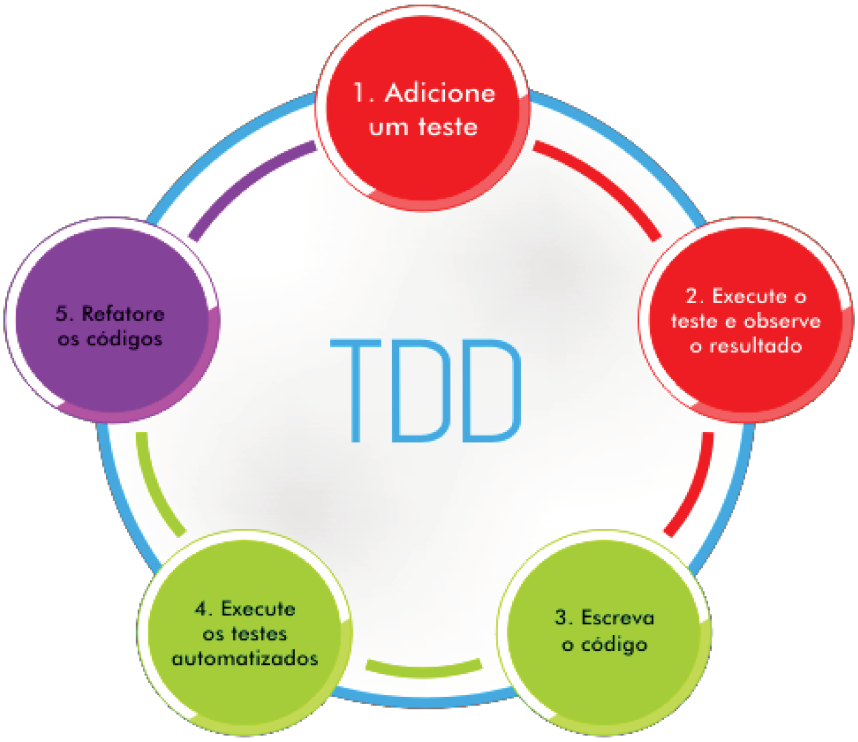

Figura 1: Ciclo *Red-Green-Refactor* [7]

Inúmeros são os benefícios decorrentes das práticas de TDD [7]:

- Um código mais claro, uma vez que os testes unitários geralmente efetuam checagens em porções menos extensas;
- Quando bem estruturado, um teste acaba servindo como documentação do código, visto que, a partir da leitura da especificação do teste, é possível ter uma noção do funcionamento de uma classe, método e/ou objeto;
- Um rápido *feedback* com alertas para problemas encontrados, uma vez que os testes são repetidos a cada novo ciclo de desenvolvimento/manutenção em uma aplicação;
- Uma cobertura adequada de diferentes trechos de código, algo que poderia não se conseguir somente através de testes de sistema ou de aceitação;
- Maior economia de tempo e de recursos financeiros na manutenção de uma aplicação, uma vez que diversas falhas acabam sendo apontadas e solucionadas ainda durante a etapa de desenvolvimento.

Assim, as técnicas de TDD, ao favorecer um código mais simples e de fácil manutenção, acabam contribuindo para uma melhor assimilação das boas práticas de desenvolvimento de *software*, uma vez que [7]:

- Desenvolver guiado por testes permite separar a lógica de negócios ou de acesso a dados das camadas gráficas de uma aplicação. A este tipo de princípio se dá o nome de Separação de Responsabilidades, utilizando-se uma série de recomendações e diretrizes que conduzam à obtenção de aplicações formadas por componentes mais coesos;
- Quanto à noção de coesão, este conceito deve ser compreendido como a medida com a qual uma estrutura de *software* (classe ou componente) atende o objetivo inicial para o qual foi concebida. Uma alta coesão indica que o item considerado não acumula responsabilidades além daquelas para as quais foi especificado, uma característica perseguida por todos os projetos de *software* que procuram ser bem estruturados. Projetar testes unitários de uma maneira simples é também uma ação que contribui para a obtenção de classes mais coesas;
- Baixo acoplamento fazendo com que haja pouca dependência entre os diversos componentes envolvidos. Um alto acoplamento resulta em um nível de complexidade devido a alta dependência, fato este que pode levar a dificuldades na manutenção futura das funcionalidades.

2.2.2 TESTE DE INTEGRAÇÃO

Testes de integração têm como objetivo encontrar defeitos na integração das diferentes unidades do *software*. Consiste em testar as integrações entre componentes, observando a interação de cada componente com outras unidades, seguindo a arquitetura definida para o *software* [20]. Normalmente, para os testes de integração, são utilizadas técnicas caixa cinza de teste, pois pelo menos a arquitetura do software a ser testado é conhecida. Considerando a maneira como é feita a integração de componentes, ela pode seguir abordagem incremental ou não. Enquanto na abordagem não-incremental (também chamada abordagem *Big-Bang*) o sistema é agrupado por completo, na abordagem incremental o sistema é agrupado em etapas, facilitando assim o isolamento das falhas.

Abordagens incrementais diferem entre si na forma em que se constroem as etapas de agrupamento de módulos. Algumas estratégias buscam isolar as falhas para facilitar a correção. Cada uma dessas estratégias tem suas vantagens e desvantagens. São elas [21]

- ***Bottom-up***: essa estratégia, também conhecida como ascendente, é iniciada pelos componentes mais simples (responsáveis pela computação) seguindo em direção aos mais altos da hierarquia do sistema (responsáveis pelo controle). Neste caso,

devem-se construir os *drivers* (tipo de *Test Doubles* que faz a chamada a módulos de serviço e recebe os resultados da computação) para substituir os componentes (unidades, módulos ou subsistemas) que ainda não foram construídos ou integrados. Entre as desvantagens para essa estratégia estão: (i) há uma grande necessidade de controladores; (ii) muitas unidades devem ser integradas antes de conseguir operacionalizar qualquer sistema; (iii) falhas / defeitos de interfaces de visualização são detectados mais tarde;

- ***Top-down***: essa estratégia, também conhecida como descendente, é iniciada a partir dos níveis mais altos da hierarquia do sistema. Durante a integração são usados *stubs* (tipo de *Test Doubles* que recebe uma chamada de módulos de controle e devolve resultados similares à computação real) para substituir os componentes (unidades, módulos ou subsistemas) que ainda não foram construídos ou integrados. Entre as desvantagens dessa estratégia estão: (i) grande necessidade de simuladores; (ii) falhas do projeto de arquitetura são identificados muito tarde; (iii) testes de funcionalidades são tardios pela baixa evolução e liberação de pacotes;
- ***Sandwich***: é a combinação das duas estratégias anteriores, sendo também conhecida como mista;

Na estratégia *Big-Bang* os testes de integração acontecerão com todas as unidades do sistema sendo testados de uma única vez. Entre as desvantagens dessa estratégia estão: (i) falhas de interfaces serão detectados mais tarde; (ii) falhas no projeto de arquitetura serão detectados mais tarde; (iii) a identificação das falhas tem um custo alto.

O teste de integração precisa ser independente e determinístico, ou seja, ao ser executado múltiplas vezes, deve apresentar o mesmo resultado. Martin Fowler [22] analisa problemas relacionados ao não determinismo de forma clara e sucinta, onde sugere que os testes deixam de ser confiáveis quando param de ser determinísticos, o que leva ao abandono posteriormente. O não-determinismo surge por cinco causas principais, que precisam ser evitadas [22]:

1. **Falta de isolamento**: Trata da questão em que o estado de um banco de dados (ou outros recursos externos) interfere na correta execução do teste;
2. **Comportamento assíncrono**: A utilização de *sleep* em *threads* pode tanto tornar os testes lentos quanto interromper a execução do método por causa do *timeout*. Portanto, em cenários com processamento assíncrono, isto é, com *threads* executando em paralelo, deve-se usar *callbacks* para garantir que o teste seja executado no momento correto;

3. **Serviços remotos**: Em alguns casos é comum o uso de serviços remotos reais para testes de integração entre sistemas, já que nem sempre certos serviços estarão disponíveis para teste;
4. **Tempo**: Depender do relógio é algo claramente não-determinístico. Testes que dependem de tempo (hora corrente) não irão gerar resultados iguais. Portanto, não se deve depender de horário em cenários de teste;
5. **Vazamento de recursos**: Quando os recursos são mal gerenciados, os testes podem passar a falhar por motivos distintos, apresentando comportamento não-determinístico, como por exemplo, falta de memória.

2.2.3 TESTE DE REGRESSÃO

Teste de regressão consiste em aplicar a cada nova versão do *software* ou a cada ciclo de desenvolvimento, todos os testes que foram aplicados anteriormente ao sistema. Esses testes devem ser atualizados para adaptar às mudanças ocorridas na nova versão.

Podemos destacar diversas vantagens na utilização destes testes, tais como: melhoria na qualidade do produto final; evolução no processo de desenvolvimento; maior competitividade; ganho de produtividade; redução de retrabalho; rapidez no atendimento aos clientes; melhoria da satisfação dos clientes [12].

O teste de regressão é um processo caro usado para validar novas versões do *software* e para detectar se novas falhas foram adicionadas a porções de código que não sofreram alterações. Uma maneira de aplica-lo é reutilizar todos os casos de testes no teste de regressão. Porém, a reexecução de todos os casos de teste pode ser considerada uma atividade inviável ou extremamente custosa. Visando aumentar o custo-benefício do teste de regressão, muitos pesquisadores têm se dedicado a definir técnicas de seleção de testes. Rothermel et al. [23] considera quatro técnicas para o reuso da suíte de testes da versão original do *software*:

1. Reteste total, reutilização na execução do *software* modificado dos casos de teste previamente desenvolvidos. Casos de testes obsoletos devem ser reformulados ou descartados. Porém, essa técnica pode ser muito cara, já que executar todas as situações de teste exige muito tempo e grande esforço humano;
2. Seleção de testes de regressão: também há o reuso de casos de teste, porém seletivamente, centrando-se sobre subconjuntos de suítes de teste existentes. Uma técnica de seleção de testes de regressão segura deve garantir que os casos de testes

descartados não revelem falhas na nova versão. Encontrar o equilíbrio entre segurança e eficiência da técnica é um grande desafio;

3. Redução do custo do teste: dada pela eliminação de casos de teste da suíte de testes de forma permanente. À medida que o *software* evolui, novos casos de testes são criados para validar novas funcionalidades. Com isso, alguns testes podem se tornar redundantes. A técnica de redução da suíte de testes aumenta a eficiência do conjunto de testes eliminando os testes redundantes. Esta técnica traz bastante economia no custo de detecção de falhas, porém ela pode igualmente reduzir a eficácia dos testes;
4. Priorização do caso de teste: estabelece uma ordenação dos casos de testes de modo que aqueles com prioridade mais elevada, de acordo com algum critério, sejam executados mais cedo no ciclo de teste de regressão em comparação aos casos de testes com baixa prioridade. Por exemplo, os testadores podem querer priorizar os testes utilizando critérios de cobertura de código, tempo de execução, funcionalidades mais utilizadas, probabilidade de detecção de defeitos, entre outros. A priorização de caso de teste pode ser usada no teste inicial do *software* ou durante a fase de teste de regressão. O benefício de usar a priorização durante o teste de regressão é que os resultados de análises precedentes podem ser usados para dar a melhor prioridade às situações de teste futuras.

Mesmo utilizando as técnicas citadas acima é recomendada a utilização de ferramentas de automação para aumentar a produtividade e viabilidade desse tipo de teste, permitindo resoluções com maior agilidade. Antes de começar a automatizar todos os casos de testes existentes, devemos planejar e avaliar se realmente a funcionalidade em questão requer ou não sua adoção. Os principais fatores que devemos considerar são [24]:

- **Rastreabilidade**: Manter mapeada a rastreabilidade do *software* permitirá facilmente identificar quais as funcionalidades podem ser afetadas por uma mudança no *software*. Mudanças no *software* podem incluir novas funcionalidades ou alterar funcionalidades existentes em função de alguma necessidade do cliente;
- **Rotinas críticas**: Identificar junto ao cliente quais são as rotinas mais utilizadas por ele, dentre essas definir quais delas são críticas; normalmente, críticas são aquelas que não podem de maneira alguma apresentar defeitos;
- **Testes redundantes**: Casos de testes que provocam a mesma execução do *software* não revelam falhas distintas quando executados. Identificando-se a existência de

teste redundantes não se deve planejar sua execução na etapa de testes de regressão, apenas testes que possam revelar novas falhas devem ser executados;

- **Testes que falharam**: Identificar as execuções de testes que em algum momento falharam, pode ser uma boa estratégia de priorização. Testes que falharam nas versões anteriores e também em chamados abertos pelos clientes, podem ser planejados para serem reexecutados na etapa de testes de regressão e assim evitar a sua recorrência;
- **Relevância do teste**: Testes de funcionalidades que apresentam frequência alta de utilização devem ser priorizados para se elaborar casos de testes automáticos;
- **Dificuldade nos testes manuais**: existem funcionalidades muito complicadas de serem testadas, pois envolvem diferentes perfis de usuários, regras de negócio diferenciadas, integrações diversas, ou seja, devido a complexidade, uma automatização deve ser feita;
- **Frequência de execução**: se o caso de teste for executado somente uma vez, não existe a necessidade de automatizá-lo, contrariamente testes frequentes devem ser priorizados na automatização.

2.2.4 TESTE DE SISTEMA

O teste de sistema, analisa o item a ser testado com base na sua especificação. Utiliza, normalmente, técnica caixa preta e tem como objetivo verificar o processo de entrada dos dados, o processamento e sua resposta. É com esse tipo de teste que se pode validar a implementação apropriada das regras de negócio e demonstrar ao desenvolvedor e ao cliente que o *software* atende aos requisitos estabelecidos [20].

A execução dos testes deverá seguir os procedimentos definidos nos casos de teste [14]. O ambiente de teste é, normalmente, um ambiente controlado que apresente características similares ao ambiente de produção [25]. Este tipo de teste não se limita a testar somente requisitos funcionais, mas também requisitos não funcionais, tais como a expectativa do cliente quanto ao desempenho, testes de segurança, teste de robustez, entre outros. Esse conjunto de testes procura garantir a operação correta do sistema [14].

O teste de aceitação é um tipo de teste de sistema associado às necessidades dos usuários e aos requisitos e processos de negócios. É realizado para verificar se o sistema satisfaz os critérios de aceite e para que o cliente obtenha as informações geradas pelo sistema e decida

por aprovar (ou não) o sistema. O objetivo desse teste é estabelecer a confiança do cliente no sistema.

Neste nível apenas algumas funcionalidades chaves são testadas pelo cliente e/ou usuários. Existem três abordagens para o teste de aceitação [14]:

- Teste de aceitação de usuário (UAT): Corresponde a um teste caixa-preta realizado por um grupo restrito de usuários finais antes da disponibilização do sistema. Tem por função verificar se o sistema atende aos seus requisitos originais e às necessidades do usuário;
- Teste de aceitação operacional (OAT): Também conhecido como teste de preparação. Procura garantir que os processos e procedimentos estejam prontos para que o usuário/testador possa utilizá-los;
- Alfa-teste e Beta-teste: Alfa-teste é realizado pelo cliente no ambiente de desenvolvimento no período entre o término do desenvolvimento e a entrega, enquanto que o Beta-teste é realizado pelo usuário final nas instalações do cliente após o alfa-teste.

2.3 METODOLOGIA DE DESENVOLVIMENTO DE TESTE

As metodologias de desenvolvimento de teste de *software* especificam padrões sobre como os testes devem ser planejados e executados e como devem ser criadas a documentação necessária. Procura definir como projetar, da melhor maneira, todo o processo de teste [25]. As empresas em geral, principalmente as de pequeno e médio porte, sentem dificuldade de associar metodologias de teste de *software* a seus processos de desenvolvimento. A dificuldade se dá, muitas vezes, pelos recursos financeiros e também pelo fato de ser limitado o recurso de pessoal.

A utilização de uma metodologia de teste pode reduzir custos com retrabalho e com manutenção, levando a minimizar os defeitos na versão do produto entregue ao cliente e, consequentemente, refletindo em um aumento do nível de confiança por parte dos clientes. Com mais tempo disponível, a equipe pode investir em melhorias e novos planejamentos [26].

Um exemplo de metodologia de testes é a desenvolvida pelo CenPRA (Centro de Pesquisas Renato Archer), sendo fundamentada nos modelos sugeridos pela norma IEEE 829 [27], que descreve os documentos que devem ser gerados na atividade de gerência dos testes de *software* [28]. A metodologia utiliza-se de técnicas, procedimentos e ferramentas para melhorar o processo de teste, capacitando as empresas produtoras a desenvolver produtos com maior

qualidade. Foi desenvolvida de maneira a que as empresas pudessem aplicá-la a vários tipos de sistema, tais como sistemas de informação e sistemas específicos. Também atende diferentes tipos de organização desde microempresas até as de grande porte, já que cada uma pode selecionar a melhor maneira de aplica-la.

O processo de teste padrão não contempla a automação dos testes visando, com isso, simplificar o processo inicial. Essa constatação corrobora com trabalhos da literatura [30, 31] que afirmam que um processo só deve ser suportado por ferramentas quando estiver convenientemente definido e consistentemente adotado. O processo de automação de teste tem maior probabilidade de ser bem-sucedido em organizações que contem com uma equipe de teste bem definida e com um processo padrão de documentação a ser seguido [25].

A metodologia do CenPRA baseia-se em três componentes [28]:

- Treinamento: capacitação em conceitos básicos sobre teste, técnicas de teste, documentação e processo de teste;
- Processo de teste: realização das atividades de planejamento, projeto, execução e acompanhamento dos testes;
- Suporte para geração de documentos: aplicação de técnica para criação dos documentos baseado na norma IEEE.

O processo de teste proposto na metodologia do CenPRA está baseado em alguns pressupostos básicos [28]:

- Os testes de sistema e aceitação são projetados e executados sob a responsabilidade da equipe de teste;
- Os testes de sistema e, eventualmente, o de aceitação são realizados de forma iterativa, havendo, antes do início de cada ciclo de teste, uma avaliação rápida do produto;
- Plano de Teste: apresenta o planejamento incluindo a abrangência, abordagem, recursos e cronograma. Identifica os itens e as funcionalidades a serem testados, as tarefas e riscos associados;
- Especificação de projeto de teste: identifica as funcionalidades e características a serem testadas, os casos e procedimento de teste a serem utilizados e apresenta os critérios de aprovação; define os casos de teste, incluindo dados de entrada, resultados esperados das ações e condições gerais para a execução do teste; especifica os passos para executar um conjunto de casos de teste;

- Relatórios de Teste: identifica os itens encaminhados para teste (no caso de equipes distintas serem responsáveis pelas tarefas de desenvolvimento e de teste); apresenta registros cronológicos dos detalhes relevantes relacionados com a execução dos testes; documenta qualquer evento que ocorra durante a atividade de teste e que requeira análise posterior; apresenta de forma resumida os resultados das atividades de teste associadas a uma ou mais especificações de projeto de teste e provê avaliações baseadas nesses resultados.

2.4 CONSIDERAÇÕES FINAIS SOBRE O CAPÍTULO

Nesta seção foram apresentados conceitos que serviram como base para a presente dissertação. Inicialmente, foi apresentado o conceito de Integração Contínua (IC), assunto focal desta dissertação e a seguir foi abordado conceitos envolvidos nas diversas atividades de teste de *software*.

Os tópicos aqui descritos serão usados durante todo o texto. A próxima seção apresenta trabalhos relacionados ao presente estudo, atualizando o conhecimento sobre os estudos correlatos existentes. Abrange trabalhos sobre integração contínua, metodologia de testes, arquiteturas de referência e ferramentas de apoio a essas atividades.

3 TRABALHOS RELACIONADOS E FERRAMENTAS DE APOIO

Nesta seção são apresentados os trabalhados relacionados ao tema proposto. Esta seção é composta pelas seguintes subseções: 3.1 Práticas de Integração Contínua, onde se apresenta o trabalho de Fowler descrevendo as práticas comumente adotadas em um ambiente de Integração Contínua; 3.2 Avaliação da Integração Contínua, onde são apresentados trabalhos que abordam metodologias de avaliação para ambientes integrados de desenvolvimento; 3.3 Ferramentas de Apoio à Integração Contínua, onde são apresentadas ferramentas existentes que apoiam o processo de Integração Contínua; 3.4 Metodologia de teste, onde são apresentados estudos de ferramentas e métricas para um caso de teste bem-sucedido; 3.5 Arquitetura de integração contínua, onde arquiteturas de referências são apresentadas.

3.1 PRÁTICAS DE INTEGRAÇÃO CONTÍNUA

O trabalho desenvolvido por Fowler [31] apresenta 11 práticas da integração contínua:

1. **Manter um único repositório:** As equipes de desenvolvimento de *software* têm buscado ferramentas para gerenciar o código fonte dos sistemas. Essas ferramentas, chamadas ferramentas de gerenciamento, são recursos integrantes da maioria dos projetos de desenvolvimento. Essas ferramentas devem apontar para um único repositório que contenha todas as informações do projeto.

2. **Automatizar a compilação:** O uso de ambientes automatizados de *builds* é prática comum das equipes de desenvolvimento. Porém, um erro comum dessas equipes é não incluir todos os artefatos na compilação automatizada. A compilação automática deve incluir desde alterações na estrutura do banco de dados até a publicação no ambiente produtivo, passando pela compilação do executável e execução bem-sucedida dos testes.

3. **Faça o seu *build* ser testado automaticamente:** Uma boa maneira de revelar as falhas mais rapidamente e eficientemente é incluir testes automatizados no processo de compilação. O teste não é perfeito, mas pode revelar uma quantidade relevante de falhas, normalmente o suficiente para a melhoria necessária. Em particular, a ascensão do *Extreme Programming* (XP) [32] e TDD [7] têm feito muito para popularizar o código de auto teste e como resultado, muitas pessoas têm percebido o valor da técnica.

4. **Compartilhamento (*commit*) constante de código:** Integração permite que cada desenvolvedor saiba quais foram as alterações que outros desenvolvedores fizeram. Como acontece com qualquer ciclo, o desenvolvedor atualiza sua cópia de trabalho para coincidir com

a linha principal, resolve eventuais conflitos que surjam e, em seguida, constroem a parte do produto a ser trabalhada na sua máquina local. No momento do compartilhamento, ao integrar o código alterado, descobrem se há conflitos entre este código e o código já existente na linha principal. Esse compartilhamento (*commit*) deve ser constante (ao menos uma vez por dia). Revelar falhas prematuramente facilita a correção do código. Conflitos que permanecem despercebidos por semanas podem ser muito difíceis de se resolver. Os desenvolvedores devem sempre compartilhar os testes desenvolvidos e caso haja uma dependência de alguma integração o desenvolvedor deverá compartilhar o teste utilizando *mock*.

5. **Cada *commit* deve construir o projeto em uma máquina de integração:** Deve se assegurar que a construção ocorreu em uma máquina de integração e somente se esta compilação de integração for bem-sucedida, deve ser considerado o *commit* um sucesso. Um servidor de integração contínua age como um monitor para o repositório. Toda vez que um *commit* no repositório termina, o servidor automaticamente verifica as fontes na máquina de integração, inicia uma compilação, e notifica o desenvolvedor do resultado da compilação.

6. **Corrigir imediatamente a construção de executável que falhar:** Uma parte fundamental ao se fazer uma compilação contínua é que se o *build* da versão principal falhar, ele precisa ser corrigido imediatamente. Muitas vezes, a maneira mais rápida de se corrigir a compilação é reverter o último *commit* da linha principal, assim o sistema volta para a última versão estável.

7. **Mantenha a construção rápida:** O principal foco da Integração Contínua é fornecer um *feedback* rápido. Cada minuto reduzido no tempo de construção, é um minuto a mais para cada desenvolvedor a cada *commit* realizado.

8. **Teste em um ambiente igual ao ambiente de produção:** O objetivo do teste é detectar qualquer problema que o sistema terá na produção. Para se obter um bom resultado é necessário que as mesmas dependências existentes no ambiente real sejam reproduzidas no ambiente de teste. Se o teste for executado em um ambiente diferente, cada diferença resulta em um risco, uma vez que em um ambiente distinto o comportamento do sistema durante os testes pode não refletir o que pode acontecer na produção. O uso de virtualização pode tornar mais fácil montar ambientes de teste semelhantes. Além disso, também pode permitir executar vários testes em uma máquina, ou simular várias máquinas em uma rede.

9. **Tornar o último executável de fácil acesso para todos:** Certifique-se de que há um lugar conhecido, onde todos podem encontrar as últimas versões do executável do sistema, como por exemplo, uma loja online ou um repositório de artefatos. Assim, desenvolvedores e clientes podem atualizar para a última versão estável do sistema quando necessário.

10. **Todos os membros da equipe podem ter acesso à última versão do código fonte:** Integração Contínua visa a comunicação, deve-se garantir que todos os envolvidos possam ter acesso ao estado atual do código fonte do sistema e o histórico das mudanças efetuadas. Uma das coisas mais importantes para se comunicar é o estado da compilação da versão principal.

11. **Automatizar a implantação:** A implantação automática ajuda a acelerar o processo de publicação do sistema e a reduzir os defeitos. Uma vez que a última versão se encontra estável, é necessário a intervenção de um desenvolvedor para publica-la, ou em uma loja ou no ambiente produtivo. Automatizar essa tarefa evita erros humanos, como por exemplo, publicação equivocada de pacotes.

Para que essas práticas possam ser atendidas, ferramentas de auxilio são disponibilizadas para desenvolvedores e profissionais de teste. Essas práticas apresentadas por Fowler foram tomadas como base para se propor a arquitetura no ambiente de integração contínua melhorando o processo de teste em conjunto com a compilação na máquina do desenvolvedor e um melhor *feedback* do resultado dos testes. Os passos 1, 2, 3, 4, 6, 7, 9 e 10 foram implementados na instancia da arquitetura de referência, estendendo as práticas ao adicionar como um novo passo a cobertura de testes.

A próxima seção apresenta trabalhos que se relacionam com a tarefa de avaliação no ambiente de integração contínua.

3.2 AVALIAÇÃO DA INTEGRAÇÃO CONTÍNUA

Lu, Yang e Qian [29] descrevem uma análise dos conceitos e os benefícios de se utilizar um ambiente de integração contínua. Além disso, mostram a necessidade de se introduzir este conceito no desenvolvimento do projeto, uma vez que essa prática pode melhorar o tempo de resposta da equipe ao se detectar falhas e permitir que os desenvolvedores foquem na inovação e na produção do sistema com qualidade. Como experimento, é utilizado a plataforma Jenkins [33] que, atualmente segundo os autores, é a mais utilizada para integração contínua. Essa plataforma é composta pela automação de testes unitários verificando problemas de sintaxe, tarefas concorrentes e executando vários testes com resultados previamente definidos para garantir as funcionalidades do sistema. Os autores citam as ferramentas e *plug-ins* utilizados na integração do sistema, tais como o Sikuli [34] para teste de interface gráfica. Realizaram um treinamento com a equipe sobre os conceitos e a utilização das ferramentas correspondentes, monitorando um projeto piloto. Perceberam que o número de falhas encontradas e resolvidas em tempo de desenvolvimento foi maior quando comparado a projetos que não utilizam um

ambiente de integração contínua. Assim concluíram que um ambiente contínuo com testes automatizados melhora a produtividade dos desenvolvedores, a qualidade do código, antecipando a correção das falhas e diminui os riscos de execução do projeto. O exemplo de práticas de integração contínua embasou as utilizadas nesta proposta.

Yüksel et al. [35] realizaram um trabalho que descreve como utilizar um ambiente de integração contínua com técnicas automáticas de testes para garantir um sistema C4ISR (*Command, Control, Communications, Computers, Intelligence, Surveillance, and Reconnaissance*). O foco era um C4ISR integrado, multi plataforma, multi linguagem, robusto e livre de falhas geradas por alterações decorrentes de melhorias ou refatoramentos. Foi utilizado a plataforma TeamCity [36] por ser *OpenSource*, e um *framework* de testes baseado no modelo XML desenvolvido pelos autores. Os testes foram executados utilizando-se o JUnit [37] por ser uma plataforma de testes em Java que executa testes em outras linguagens de programação. Durante os experimentos, foi executado um arquivo XML para cada cenário de teste descrito. Foi observado uma melhora significativa na quantidade de testes executados e na qualidade das compilações, aumentando a taxa do sucesso da compilação de 72% para 93%. Esse trabalho colabora com um modelo de arquitetura de teste, separando a compilação e os testes com o objetivo de serem executados paralelamente, quando necessário.

Brandtner, Giger e Gall [30] realizaram uma prova de conceito integrando as diversas plataformas de ambientes integrados existentes no mercado, e criaram uma interface intuitiva para mostrar os dados coletados. Os resultados demonstraram que a integração realizada pelos autores, trouxe melhores benefícios se comparado com o uso individual de uma plataforma. Além disso, houve uma melhora na visibilidade do status do projeto e na qualidade da informação fornecida pelas ferramentas, mas a quantidade de informações gerada e com isso a dificuldade de filtrar essas informações, aumentou a cada plataforma adicionada ao ambiente integrado. Com base nisso, neste trabalho optou-se por criar uma plataforma web para mostrar os dados coletados, a fim de simplificar a integração com o *software* a ser desenvolvido. Essa abordagem, ajudou a definição da ferramenta de integração contínua utilizada na atual proposta, pois a ferramenta escolhida segue o padrão simplificado web, facilitando a leitura das informações geradas pela metodologia proposta.

Uma revisão sistemática de abordagens, ferramentas, desafios e práticas sobre a integração contínua foi realizada por Shahin, Babar e Zhu [38], mostrando que a investigação e profissionais de engenharia de *software* atuando na área de práticas de integração contínua estão crescendo, principalmente em abordagens empresariais, onde o foco principal da utilização está em sistemas web. Além disso, a abordagem proposta neste trabalho corrobora com as práticas

revisadas, sendo possível notar que as ferramentas que facilitam a implementação de práticas contínuas reduzem o tempo de construção de testes automáticos com aumento da visibilidade da equipe na detecção de falhas.

M. Hilton et al. [39] realizaram um estudo sobre a utilização de integração contínua em projetos de código aberto. Foram levantadas 14 perguntas separadas em 3 temas sobre o uso da IC, o custo da IC e os benefícios da IC. Esse estudo envolveu uma análise técnica de mais 34 mil projetos abertos no GitHub [40] e mais de 400 desenvolvedores que responderam às perguntas. Os resultados comprovam o aumento na utilização da IC, sendo que comprovadamente foram constatadas entregas de pacotes de projetos duas vezes mais rápida e com a análise dos códigos realizada, foi observado que 70% dos projetos no GitHub utilizam IC. Esse estudo incentivou o desenvolvimento da metodologia aqui proposta, frente a constatação dos benefícios trazidos pelo uso e evolução da IC em diversas plataformas de desenvolvimento, porém o nosso processo automático é realizado logo após o *commit*.

3.3 FERRAMENTAS DE APOIO AO PROCESSO DE INTEGRAÇÃO CONTÍNUA

Para os usuários de JavaScript, uma das bibliotecas de automação de tarefas mais usadas é o GruntJS [41]. Ela nasceu com o objetivo de aplicar na prática os conceitos de produtividade no desenvolvimento e integração de atividades às bibliotecas e frameworks que o projeto fizer uso. Em uma definição simplificada, o GruntJS é uma ferramenta de linha de comando baseada em tarefas para projetos em JavaScript.

De uma forma mais detalhada, podemos dizer que quando trabalhamos com um projeto em JavaScript, existe uma série de tarefas que devem ser feitas regularmente, tais como concatenar arquivos, executar um processo de detecção de falhas em alguma ferramenta e problemas no código (como o JSHint), executar testes (unitários, de integração) ou modificar seus scripts. Com o GruntJS, podemos nos ater às regras de negócio e código das aplicações e sem a preocupação com as tarefas repetidas de automatização, geração ou gerenciamento.

É importante notar que o GruntJS segue a especificação do CommonJS, um projeto desenvolvido para normatizar e padronizar convenções e estilos de JavaScript. Para isso, o GruntJS exporta a si mesmo como um módulo que contém suas configurações e tarefas [41].

A fim de facilitar o processo de configuração deve-se, idealmente, armazenar quaisquer portas, funções e outras constantes, que são utilizadas na parte superior do arquivo, como variáveis globais. Isso garante que se houver mudança em uma função ou constante, a edição

da variável global no topo seria muito mais simples do que alterar seu valor em cada localização no arquivo. Além disso, as constantes ajudam a fornecer informações através da atribuição de um nome de variável para cada valor desconhecido.

Por padrão, todos os plug-ins do GruntJS têm uma tarefa respectiva que pode ser chamada para realizar a saída desejada. Tarefas definidas pelo usuário podem ser referenciadas no final do Gruntfile para serem executadas assincronamente.

A ferramenta Cruise Control [42], é uma das aplicações mais famosas para integração contínua e é mantida e desenvolvida por voluntários dispostos a melhorar e disponibilizar novas funcionalidades. A aplicação permite que *builds* sejam executados a partir de diferentes ferramentas, integrem com diferentes gerenciadores de controle de versão, além de realizar notificações de diferentes formas, quando, por exemplo, algum defeito acontece durante a execução de um *job*.

Outra ferramenta bem conhecida é o Continuum [43]. Funciona como um servidor para integração contínua, facilitando a gerência dos projetos que terão seus *builds* executados. Ele oferece uma página *web*, que pode ser executada em diversos sistemas operacionais e a partir dela podem ser usados diferentes projetos de *build*, assim como diferentes ferramentas de controles de versão.

O SCons [44] é uma ferramenta de integração contínua cujos arquivos de configuração são implementados como um script Python, acrescido de um conjunto de módulos. Isso dá aos SCons muitas capacidades poderosas que não são encontradas em outras ferramentas, porém é necessário customizar os *scripts* e *plug-ins* para atingir sua total capacidade, além da curva de aprendizado ser maior que as demais ferramentas.

A ferramenta Jenkins [33], ou também conhecida como Hudson, é uma ferramenta para integração contínua que monitora a execução de trabalhos repetidos, como o a criação de *build* de um projeto, assim como fazem também o Cruise Control e o Continuum. Através de sua interface, torna-se fácil acompanhar o estado de cada *build*. Além disso, permite o trabalho colaborativo com diversos tipos de sistemas de controle de versão e sistemas de *build*. Também provê suporte para notificações das execuções dos *jobs* através de e-mail, RSS e IM Integration. A Tabela 1 apresenta algumas das ferramentas mais utilizadas para integração contínua e as comparações de suas características.

Tabela 1: Comparativo das ferramentas de integração contínua

	Cruise Control [42]	Continuum [43]	Jenkins [33]	SCons [44]	GruntJS [41]

Linguagens de programação que oferece suporte.	Java, Ruby/Rails e .Net	Java	Java	C, C++, D, Java, Fortran, Yacc, Lex, Qt, SWIG, .Net, TeX e LaTeX	Java, JavaScript, Ruby/Rails e .Net
Sistemas Operacionais.	Windows e Unix	Windows, Debian, Fedora Core, Mac OS X e Solaris.	Windows, Solaris e Unix.	Windows, Unix, Mac OS X e Solaris.	Windows, Unix, Mac OS X e Solaris
Ferramentas de *build*.	Ant, NAnt, Maven 1, Maven 2, Shell Script	Maven 2, Maven 1, Ant e Shell Script.	Maven 2, Maven 1, Ant, Shell Script e Batch *command* do Windows.	Script Python	Ant, NAnt, Maven 1, Maven 2, Shell Script e Batch *command* do Windows.
Suporte para testes.	Realiza um *merge* dos testes gerados pelo JUnit de um Ant com um arquivo de log gerado pelo Cruise Control.	Gera documentação de testes para projetos que usam Maven 1 e Maven 2.	Oferece suporte para JUnit e TestNG; Gera relatórios de testes, que podem ser tabulados, sumarizados e indicados com informações de histórico; Cria um gráfico contando detalhes de cada teste na documentação.	Oferece suporte para QUnit, YUI Test e o JSTestDriver	Oferece suporte para JUnit, QUnit, YUI Test e o JSTestDriver

Principais funcionalidades disponíveis.	É possível: (i). Especificar se ao encontrar uma falha no *build*, deve-se continuar ou parar a execução; (ii). Atualizar o arquivo de *build* de um projeto através de um repositório, antes de sua execução; (iii). Definir onde arquivos de log gerados pelo Cruise Control serão armazenados.	É possível: (i). Disponibilizar relatórios de testes unitários; (ii). Criar grupos de acesso; (iii). Acompanhar, em tempo real, a execução de um *build* através de um console, etc.	É possível: (i). Integrar com geração de Javadoc; (ii). Definir onde serão armazenados os arquivos empacotados (ex: .war, .jar) criados a partir da execução de algum *build*; (iii). Acompanhar, em tempo real, a execução de um *build* através de um console, etc.	É possível: (i). Disponibilizar relatórios de testes unitários; (ii). Definir onde serão armazenados os arquivos empacotados (ex: .war, .jar) criados a partir da execução de algum *build*; (iii). Acompanhar, em tempo real, a execução de um *build* através de um console, etc.	É possível: (i). Disponibilizar relatórios de testes unitários; (ii). Especificar se ao encontrar uma falha no *build*, deve-se continuar ou parar a execução; (iii). Definir onde serão armazenados os arquivos empacotados (ex: .war, .jar) criados a partir da execução de algum *build*; (iv). Definir onde arquivos de log gerados pelo GruntJS serão armazenados; (v). Acompanhar, em tempo real, a execução de um *build* através de um console, etc.
Configuração dos projetos.	Configuração realizada em um arquivo .xml. Curva de aprendizado inicial pode ser significativa.	Configuração realizada em uma interface.	Configuração realizada em uma interface.	Configuração realizada através de um script python. Curva de aprendizado inicial pode ser significativa.	Configuração realizada em um arquivo .gruntfile. Curva de aprendizado inicial pode ser significativa.
Ferramentas de controle de versão.	AccuRev, AlienBrain, ClearCase, CVS, Perforce, PVCS, StarTeam, Subversion, Visual Source Safe.	ClearCase, CVS, Local, Perforce, StarTeam, Subversion, Visual Source Safe.	CVS e Subversion.	ClearCase, CVS, SVN, TFS, Subversion.	ClearCase, CVS, SVN, TFS, Subversion.
Formas de extensão.	Código fonte fornecido para extensões. Possibilidade de utilizar *plug-ins*.	Não permite extensões.	Código fonte fornecido para extensões Possibilidade de utilizar *plug-ins*.	Código fonte fornecido para extensões Possibilidade de utilizar *plug-ins*.	Código fonte fornecido para extensões Possibilidade de utilizar *plug-ins*.
Formas de notificação.	E-mail, Weblog, Yahoo IM message, Html, JSP.	E-mail, IM (IRC, Jabber, MSN).	E-mail, RSS, IM Integration.	E-mail, RSS, IM Integration e Html.	E-mail, RSS, IM Integration e Html.

Analisando a Tabela 1, observa-se que o GruntJS possui todas as funcionalidades citadas nas outras ferramentas, com a vantagem de ser modularizada, podendo-se escolher exatamente o que se quer utilizar sem a necessidade de grandes instalações.

Das ferramentas apresentadas, foi escolhida a GruntJS pois se adapta às novas tecnologias, tais como HTML5 e JavaScript, além de cumprir o requisito de compatibilidade com várias linguagens. Oferece, também, suporte a testes de regressão e testes de navegação e uma variedade de *plug-ins* e *add-ons* capazes de otimizar a ferramenta para qualquer caso especifico. Além disso, a GruntJS [41], por se tratar de uma ferramenta de código aberto, está disponível para a pesquisa a ser desenvolvida no âmbito deste trabalho.

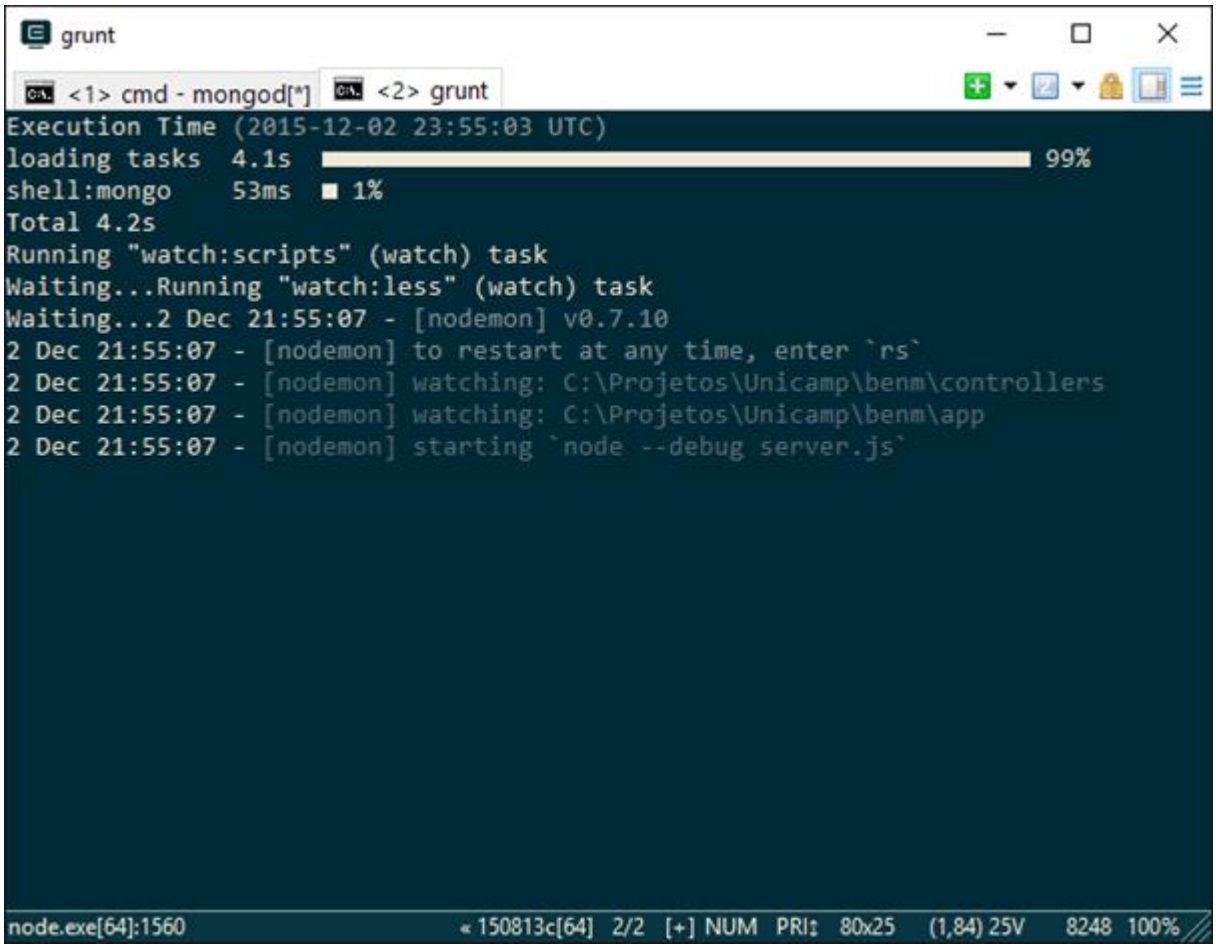

Figura 2: Demonstração da execução da ferramenta GruntJS

A Figura 2 apresenta uma demonstração simplificada do funcionamento da ferramenta GruntJS, juntamente com a execução de um teste, escrito em Jasmine, de conexão ao banco de dados não relacional MongoDB. Nas primeiras linhas da Figura 2, pode-se observar o tempo que foi gasto pelo sistema para carregar as tarefas relacionadas aos testes (*loading* de

bibliotecas) e o tempo para executar o teste de conexão ao banco de dados, sendo gastos 4,2 segundos totais para a execução do teste. As demais linhas mostram o log da execução e o resultado do teste, onde a conexão foi bem-sucedida e foram encontrados três contatos na base de teste "*Contacts*". A próxima seção traz os trabalhos relacionados a metodologias de testes integrados.

3.4 METODOLOGIA DE TESTES

Modelos conceituais de testes integrados, são apresentados por Xiang e Jiang [45], onde são identificadas as entidades de todos os tipos de usuários do sistema, tais como pessoa, equipamento, ambiente e organização e os atributos dessas entidades. Essas informações são premissas para os cenários de testes. O modelo segue definindo as relações das entidades e as interações que elas sofrem entre si. Os autores abordam também, a arquitetura do teste integrado, que é concebida a partir de um ambiente real ou simulado. Assim, concluem que a plataforma de teste integrado otimiza os testes, diminui o custo, minimiza o tempo de desenvolvimento do teste e transforma os dados do teste em informação relevante. Os modelos conceituais de teste integrado apresentados foram utilizados como base para estruturar o modelo a ser utilizado neste trabalho, uma vez que nem todas as integrações estarão disponíveis, será utilizado o modelo com *mocks,* pois é o que melhor se ajusta à metodologia proposta.

Holmes e Kellogg [46] fizeram um estudo de como poderiam automatizar testes funcionais em aplicações web devido à dificuldade do teste em arquiteturas de múltiplas camadas, múltiplos browsers e tecnologias utilizadas na camada do cliente como o JavaScript. Escolheram a Selenium [47] por ser uma ferramenta de testes web, que utiliza scripts simples para embutir o teste no mecanismo do browser. Assim, verificaram que a Selenium lida muito bem com os problemas encontrados em testes funcionais de sistemas web sem adicionar novos problemas. Nesta presente proposta não foram contemplados testes funcionais automáticos, mas foi verificado que a ferramenta GruntJS integra com a Selenium sendo uma abordagem para o futuro dessa integração e automatização dos testes funcionais.

Brajnik, Baruzzo e Fabbro [48] adotaram uma abordagem baseada em modelos que representam os dados manipulados pela interface gráfica e seu comportamento para testar aplicações web responsivas (site flexível quando automaticamente se encaixa no dispositivo do usuário, podendo ser computadores, celulares, *tablets* e outros) utilizando diferentes navegadores, sistemas operacionais e dispositivos. Um compilador lê esses modelos, integra anotações do modelo fornecendo detalhes sobre a implementação da interface do usuário e

produz um código de teste. O engenheiro de teste pode utilizar esse código gerado para escrever testes de alto nível que podem então ser executados em todas as plataformas desejadas, usando Selenium como ferramenta. Essa abordagem apoia a realização de trabalhos que tenham foco em testes de usabilidade não apoiando os demais tipos de testes.

As técnicas de priorização de casos de teste (TCP) reordenam os casos de teste de modo que os objetivos de teste possam ser atendidos mais rapidamente. Um objetivo em potencial envolve a revelação de falhas, e as técnicas TCP têm se mostrado capazes de revelar falhas mais rapidamente como mostra Elbaum, Malishevsky, e Rothermel [49]. Nos casos em que descartar casos de teste é aceitável, a priorização de casos de teste pode ser usada em conjunto com a seleção de teste de regressão para priorizar os casos de teste no conjunto de testes selecionado. Essa integração entre descartar casos de testes com a seleção de teste de regressão foi realizada por Elbaum et al. [49] sendo utilizada na metodologia proposta de acordo com as definições do projeto. Além disso, a priorização de casos de teste pode aumentar a probabilidade de que, se as atividades de teste de regressão forem inesperadamente terminadas, o tempo de teste terá sido gasto de forma mais benéfica do que se os casos de teste não fossem priorizados.

Amrit e Meijberg [50] descrevem um estudo de caso com base em uma empresa holandesa que utilizou a técnica TDD combinado com IC. Com uma avaliação da qualidade técnica (também realizada em estudos anteriores), foi proposto uma avaliação quantitativa do impacto da implementação TDD e IC e foi fornecida uma descrição detalhada da configuração do caso. Os membros da equipe holandesa perceberam um aumento no foco na qualidade e na aplicação de testes, considerando a aceitação do cliente. As estatísticas descritivas apontam para uma melhora geral, não só encontrando menos defeitos, mas também reduzindo o tempo necessário para corrigir os defeitos (*Lead* de defeito e *throughput*). Uma das limitações deste artigo pode ser o uso de KLOC (*Kilo Lines of Code*) na maioria das métricas, bem como o uso de KLOC para medir a produtividade do desenvolvimento. As informações da utilização do TDD com IC foram utilizadas nesta proposta, onde o desenvolvedor escreve o teste unitário antes ou em conjunto com o código do sistema, porém não foi utilizado a métrica de KLOC na metodologia proposta pois o autor não acredita que uma quantidade de código desenvolvida por hora possa ser parâmetro de produtividade. As métricas utilizadas na proposta foram quantidades de defeitos encontrados e classificação dos defeitos.

3.5 ARQUITETURA DE INTEGRAÇÃO CONTÍNUA

Angelov, Grefen e Greefhorst [51] estudaram 16 arquiteturas de referência e suas aplicações para determinar seus níveis de sucesso. Foi apresentado uma estrutura que permite classificar arquiteturas de referência em três dimensões, de acordo com seu contexto, objetivos e design. Uma investigação sobre quais valores nas três dimensões podem ser combinados para o projeto de uma arquitetura de referência foi desenvolvida, sendo que cada combinação de valores nas dimensões de uma arquitetura foi chamada de tipo. Arquiteturas de referência que se encaixam em um desses tipos têm maiores chances de sucesso e aplicação efetiva. Os resultados deste trabalho contribuíram com uma base para a arquitetura de referência mais bem-sucedida.

Nakagawa, Oquendo e Becker [52] apresentaram a última versão de um processo, denominado ProSARA, para a construção de arquiteturas de referência, focando em como projetar, representar e avaliar tais arquiteturas. Para ilustrar um uso do ProSARA, foi apresentado o estabelecimento do SiMuS, uma arquitetura de referência para o domínio robótico. Como resultado, o desenvolvimento do SiMuS e outras arquiteturas de referência deu evidências de que o ProSARA pode ser considerado um processo eficiente e sistemático para construir arquiteturas de referência. Contribuindo com a montagem da arquitetura de referência proposta neste trabalho.

Nakagawa et al. [53] contribuíram propondo um modelo de referência para arquiteturas de referência, chamado RAModel (Modelo de Arquitetura de Referência), que apresenta possivelmente todos os elementos, organizados por tipos e relacionamentos, que poderiam estar contidos em arquiteturas de referência. Ao adotar o RAModel, os autores acreditam que o modelo orienta e melhora a produtividade durante a construção de novas arquiteturas de referência e podendo se tornar um suporte eficaz para o uso e a evolução das arquiteturas existentes. Ao usar o RAModel como base deste trabalho, houve uma redução no esforço e no tempo, portanto, uma melhoria na produtividade da construção da arquitetura de referência.

Riley [54] apresenta duas abordagens de arquitetura referência para automação de testes. A primeira abordagem refere-se ao ambiente dividido em componentes isolados podendo gerar um número maior de pontos de falhas, uma vez que cada equipe fica responsável somente pelo seu componente arquitetural, ou seja, uma equipe para testes, uma equipe para qualidade do código, etc. A segunda abordagem, chamada de centralizada, une todos os componentes arquiteturais em um mesmo ambiente facilitando a comunicação dos times e a responsabilidade da entrega como um todo, desde o código fonte, passando pelos testes e pela cobertura e qualidade do mesmo. Essa abordagem foi escolhida como base para a montagem da arquitetura de referência proposta neste trabalho.

Elberzhager [55] demonstra sete práticas ao se montar uma arquitetura de testes integrados em um ambiente de integração contínua, onde padrões de integração predefinidos podem ser usados diretamente uma vez que a arquitetura tenha sido definida e o teste de integração tenha sido planejado. Os padrões

de defeitos capturados em tal padrão de integração podem ser usados durante a integração teste para identificar defeitos de integração típicos, o que resulta em maior qualidade do produto. Essas práticas foram utilizadas na construção dos testes de integração na arquitetura de referência proposta neste trabalho.

3.6 CONSIDERAÇÕES FINAIS SOBRE O CAPÍTULO

Neste capítulo foram apresentados trabalhos que, de alguma forma, se relacionam com o foco da presente dissertação. Trabalhos relacionados que avaliaram práticas de integração contínua, processos de integração contínua, metodologias de teste e arquitetura de integração contínua foram relatados, destacando sua contribuição à proposta desta dissertação. Também foram expostas as ferramentas que apoiam a integração contínua, exemplos de arquiteturas de referência em ambientes de integração contínua e sua utilização para a proposta desta dissertação. O próximo capítulo apresenta conceitos e ferramentas que foram utilizadas para definir a arquitetura de referência de Integração Contínua proposta nesta dissertação.

4 INTEGRAÇÃO CONTÍNUA COM TESTES DE REGRESSÃO

Nesta seção serão apresentados a arquitetura proposta. Também serão apresentadas ferramentas e o fluxo de trabalho que fazem parte da integração contínua e auxiliam nos testes regressivos.

4.1 INTEGRAÇÃO CONTÍNUA

Observada a importância de se melhorar a qualidade dos processos de desenvolvimento de software em ambientes colaborativos, desenvolveu-se uma arquitetura para o ambiente de IC, que contempla os tipos de testes mais essenciais, isto é, testes unitários, de integração e de regressão.

A abordagem utilizada na criação da arquitetura de testes teve duas fases principais, a Análise e o Desenvolvimento:

i. Análise: esta fase estudou as técnicas de desenvolvimento dos testes em busca de melhores práticas na escrita do teste, bem como, ferramentas disponíveis para apoiar a arquitetura. O objetivo foi definir métricas e etapas para a montagem da arquitetura, mapeando as informações obtidas.

ii. Desenvolvimento: esta fase trabalhou um conjunto de ferramentas para compor o ambiente de integração contínua e a validação da arquitetura do ambiente de IC proposta.

A Figura 3 apresenta o conjunto de atividades expresso em um diagrama de atividades da UML, que contempla uma configuração do ambiente, o *commit*, de parte ou de todo o componente desenvolvido no gerenciador de versões pelo desenvolvedor, a atuação do detector de alterações identificando essa alteração e iniciando o gerenciador de tarefas. O gerenciador de tarefas, por sua vez, lê e executa os arquivos de testes escritos em linguagem de teste e retorna um relatório com as informações dos testes, podendo ser configurado em formato html ou no próprio console de execução conforme Figura 2. Esse relatório informa em qual arquivo e em qual linha de código ocorreu a falha no teste.

Nessa arquitetura, o desenvolvedor é obrigado a sempre atualizar os testes, uma vez que o processo automático executa os testes a cada *commit*, os testes irão acusar falhas quando as alterações do código fonte forem submetidas e o *build* não continuará para o *deplo.* A arquitetura prevê uma priorização dos casos de teste de acordo com as definições do projeto. Os testes que exigem muito tempo para serem executados podem ser retirados do conjunto de IC e executados no momento adequado (ou seja, em um intervalo maior de tempo). Uma vez

identificado a demora na execução do teste, esse teste fica armazenado no repositório e é executado pelo menos uma vez antes de cada entrega. Esse tempo é diferente para cada caso e deve ser definido no desenvolvimento do teste em conjunto com a equipe de negócio. Além disso, para novos desenvolvimentos, a arquitetura prevê a utilização de uma porcentagem de cobertura de testes garantindo a criação de novos casos de teste, quando necessário. Uma arquitetura do ambiente de IC é apresentada na Figura 4.

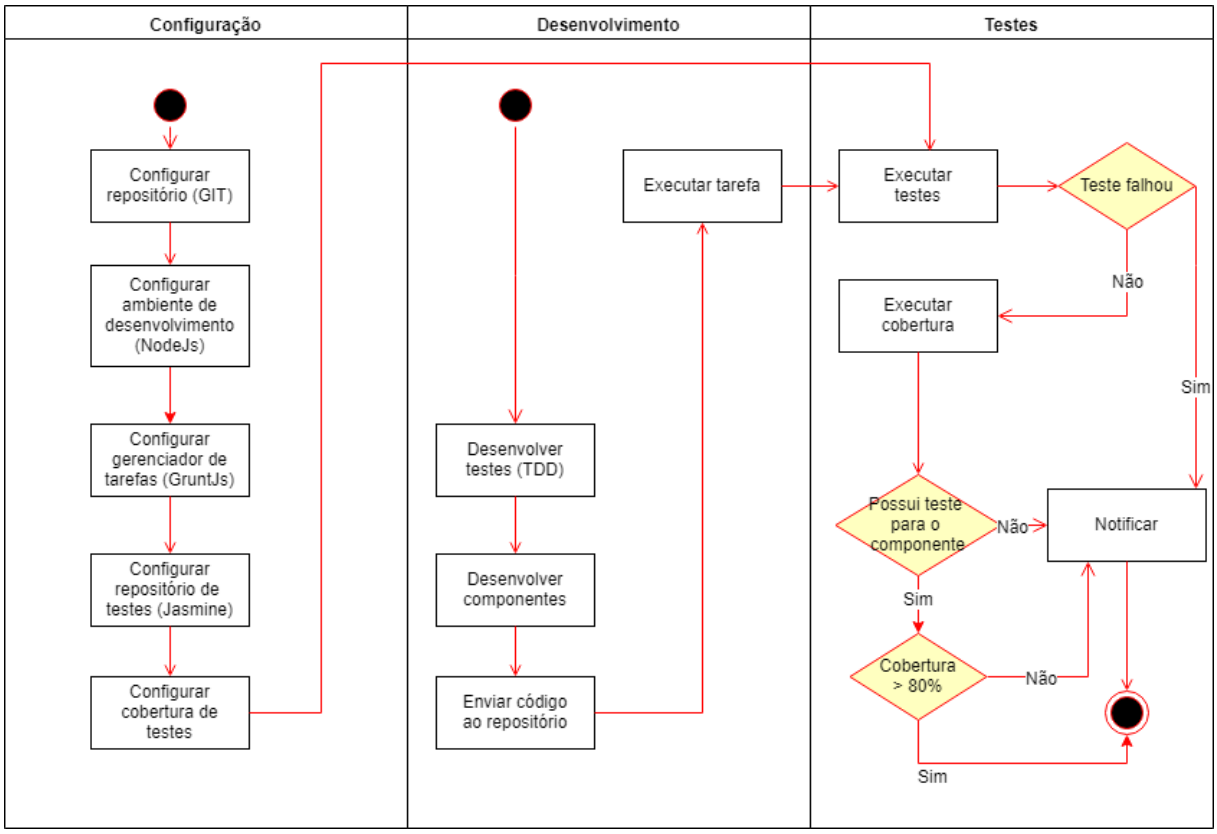

Figura 3: Diagrama de atividades

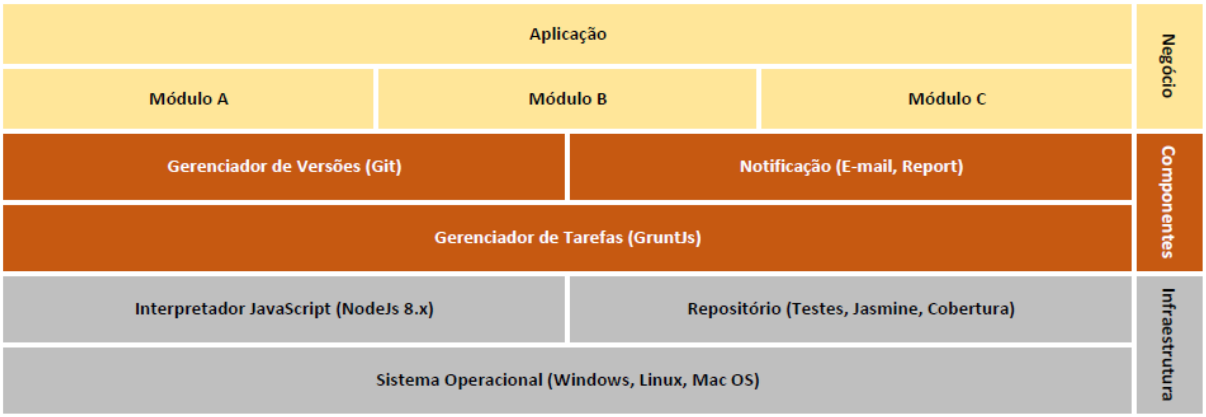

Figura 4: Arquitetura da solução proposta

O gerenciador de versão é responsável por controlar as versões dos artefatos produzidos, registrando as mudanças feitas em um arquivo ou um conjunto de arquivos ao longo do tempo, de forma que se possa recuperar versões específicas. Sendo assim, quando o ambiente detectar falha durante a aplicação dos testes, é possível identificar de forma rápida quais arquivos foram alterados, indicando as possíveis causas dessa nova falha.

O detector de alterações, é responsável por monitorar as alterações feitas nos componentes sob controle do gerenciador de versão e iniciar o gerenciador de tarefas de acordo com sua necessidade. Além disso, uma biblioteca de *plug-ins* que ajudam na configuração da integração dos componentes é integrado no ambiente, sendo que poderá ser necessário uma biblioteca específica para cada ambiente.

O gerenciador de tarefas é responsável por executar todas as tarefas configuradas, ou seja, iniciar os testes, coletar os resultados e gerar os relatórios.

Os testes devem ser escritos em uma linguagem que disponibilize um ambiente e uma estrutura para os testes, funções de assertivas e o tratamento de recursos de *mocks*, *spies*. Por exemplo, as linguagens Jasmine e Cucumber são adequadas para essa etapa.

A Figura 5 apresenta o diagrama de sequência UML com as atividades de configuração e integração continua na arquitetura proposta. Primeiramente, o usuário dispara um processo de configuração do ambiente, fazendo com que repositórios e gerenciadores sejam adequados ao sistema que será testado. Iniciando o desenvolvimento, os casos de teste são escritos e inseridos no ambiente antes mesmo do desenvolvimento das funcionalidades seguindo a prática TDD, e após a elaboração dos casos de testes é iniciado o desenvolvimento funcional da aplicação. Quando o desenvolvedor submete o código escrito no ambiente é iniciado um processo automático de execução dos testes previamente elaborados, sendo que se os testes falharem na execução é disparado um alerta ao desenvolvedor para que corrija o código submetido. Se os testes executarem com sucesso, será verificado se a cobertura dos testes está válida para aquele cenário, ou seja, se os testes que foram executados cobrem uma quantidade de linhas de código previamente estipulada, disparando uma notificação positiva em caso de sucesso e uma notificação de alerta em caso de falha. Nos casos de falhas, os desenvolvedores sabem exatamente onde devem atuar, ou na correção da funcionalidade, ou na criação de casos de testes. Nos casos de sucesso, o desenvolvedor segue para a próxima funcionalidade a ser desenvolvida repetindo o diagrama.

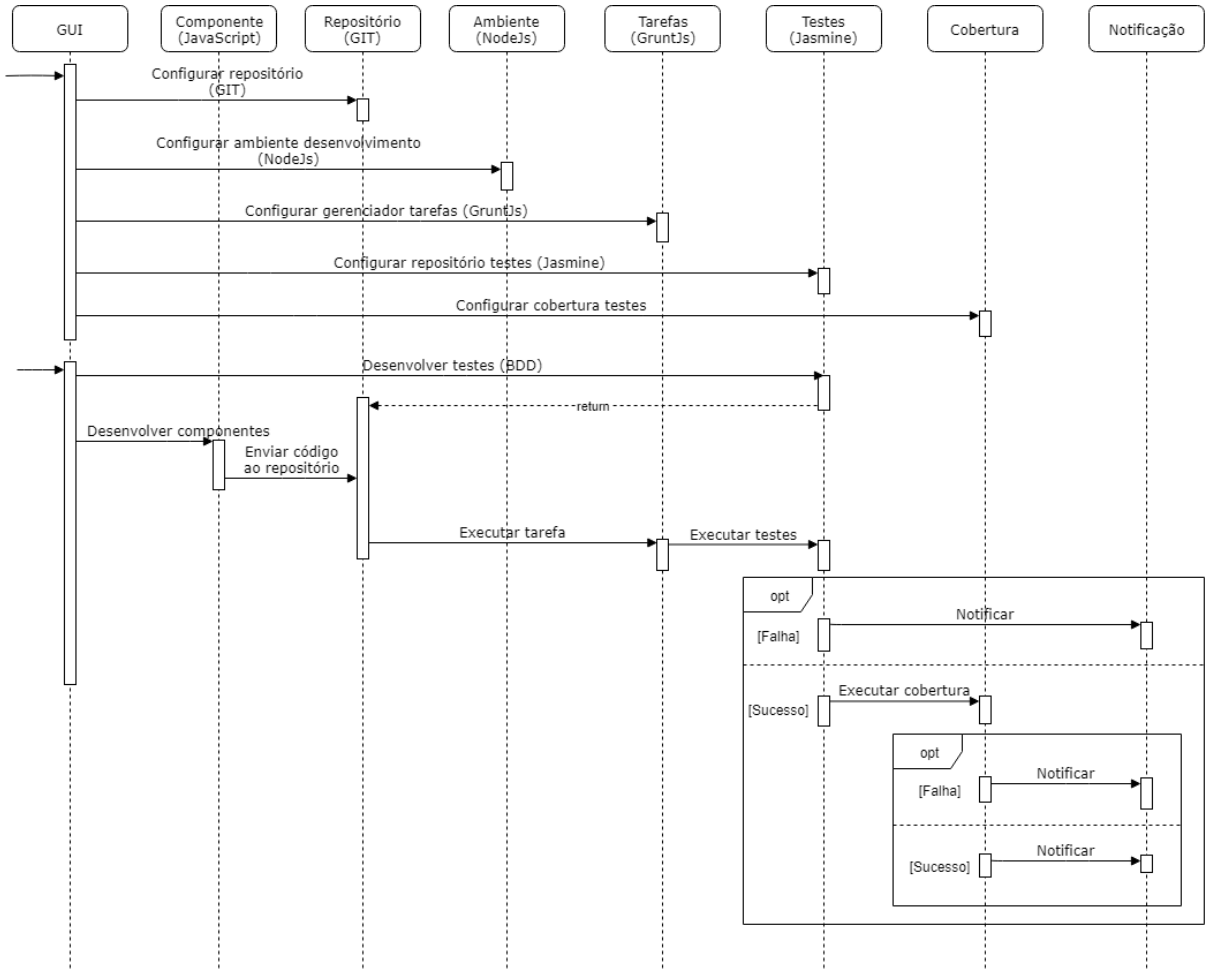

Figura 5: Diagrama de sequência

4.2 IMPLEMENTAÇÃO DA ARQUITETURA

Baseado nos estudos da literatura, como também nos estudos sobre algumas ferramentas disponíveis no mercado, foi feita uma comparação da capacidade das ferramentas que pudessem apoiar as etapas previstas na arquitetura do ambiente, levando-se em conta a adaptação às novas tecnologias (por favor, ref. à Seção 3 do Capítulo 3).

O repositório Git foi escolhido por ser um repositório leve, rápido e de fácil aprendizado.

O ambiente de integração contínua foi criado com base na plataforma NodeJs [56], sendo atualmente, a plataforma mais utilizada para sistemas Web em JavaScript [57] e por conter uma grande biblioteca de *plug-ins*.

A ferramenta GruntJS se adapta de maneira adequada às novas tecnologias, como também, contempla os diferentes tipos de teste, motivo pelo qual a ferramenta foi escolhida como gerenciador de tarefas da execução contínua dos testes.

A linguagem Jasmine foi escolhida por se tratar de uma linguagem de teste que está crescendo entre a comunidade de teste, por ser facilmente integrada a qualquer plataforma web, por ser de fácil escrita e por dar suporte a vários tipos de teste.

As próximas subseções apresentam os componentes utilizados na implementação da arquitetura proposta e o papel desempenhado por eles.

4.3 REPOSITÓRIO GIT

Com o gerenciador de pacotes do NodeJs, instalou-se as demais ferramentas utilizadas, como o GruntJS e seus *plug-ins*. O *plug-in grunt-git-hooks* do NodeJs, monitora a pasta dos arquivos fontes no repositório GIT e permite que quando as alterações forem enviadas ao repositório (passo 2), o *plug-in* comande o início do processo de execução dos *scripts* de teste automaticamente, fazendo a regressão no sistema (passo 4).

A Figura 6 apresenta um exemplo de configuração do *plug-in grunt-git-hooks,* para atender as necessidades do ambiente de IC.

No exemplo apresentado na Figura 6, a linha 6 importa o *plug-in grunt-git-hooks* que permite criar uma tarefa de execução. Neste exemplo, o arquivo de configuração é indicado (linha 11) e a partir da linha 18 define-se o bloco de execução. A linha 23 do bloco de execução indica que quando o desenvolvedor realizar o *commit*, executam-se os testes configurados no arquivo *karma.conf.js*, uma vez que ele foi indicado no arquivo de configuração (linha 11).

```
module.exports = function(grunt) {

    require('time-grunt')(grunt);
    require('load-grunt-tasks')(grunt);

    grunt.loadNpmTasks('grunt-githooks');

    grunt.initConfig({
        karma: {
            options: {
                configFile: 'karma.conf.js'
            },
            test: {
                singleRun: true
            }
        },

        githooks: {
            all: {
                options: {
                    template: '../../.git/hooks/pos-commit.js'
                },
                'pos-commit': 'karma'
            }
        }
    });

    grunt.registerTask('githook', ['githooks', 'karma:test']);
};
```

Figura 6: Exemplo do uso do *plug-in grunt-git-hooks*

4.4 LINGUAGEM JASMINE

Na implementação da arquitetura proposta, os testes foram escritos em uma linguagem chamada Jasmine [58]. O Jasmine é um *framework* de teste para JavaScript que não exige navegadores ou outros *frameworks* para ser executado. Possui uma sintaxe intuitiva facilitando a criação dos testes e é ideal para qualquer projeto Web que utilize JavaScript. Os testes poderiam ser escritos em outras linguagens, tais como o Mocha [59], Jest [60], Chai [61], Unexpected [62], Sinon [63], Enzyme [64], Testdouble [65] e Ava [66], mas optou-se pelo Jasmine pois a linguagem disponibiliza um ambiente de testes, uma estrutura para os testes, funções de assertivas, evidencia os resultados e trata os recursos de *mocks*, *spies* [67].

```
1  module.exports = function(config) {
2    config.set({
3      // Linguagem assertiva que será utilizada
4      frameworks: ['jasmine'],
5
6      // Lista de arquivos de testes
7      files: [
8        'www/js/object.js',
9        'www/js/tests/*.js'
10     ],
11
12     // Porta web que será utilizada para executar os testes
13     port: 9876,
14
15     // Nível de log utilizado
16     // config.LOG_DISABLE || config.LOG_ERROR || config.LOG_WARN
17     // || config.LOG_INFO || config.LOG_DEBUG
18     logLevel: config.LOG_INFO,
19
20     // Continuous Integration mode
21     singleRun: true
22   });
23 };
```

Figura 7: Exemplo do arquivo *karma.conf.js*

Os testes criados em Jasmine usam assertivas de comportamento, ou seja, são testes que verificam se o comportamento da funcionalidade está correto. Pode ser utilizado para verificar conteúdo de variáveis, retorno de métodos e funções, existência de tabelas ou registros no banco de dados e leitura e escrita desses registros. O arquivo *karma.conf.js* define o caminho dos arquivos de testes escritos em Jasmine, como apresentado na Figura 6 (linhas 8 e 9). Assim, o *plug-in* consegue identificar quais testes serão executados no fluxo automático.

Ainda na Figura 7, a linha 4 define qual linguagem assertiva será utilizada. Na linha 13 define-se em qual porta web será executado o teste (por exemplo http://localhost:9876), onde será simulado um navegador conectado nessa porta para execução dos testes. Na linha 21 define-se que o teste será executado apenas uma vez por *commit*.

4.5 GERENCIADOR GRUNTJS

O gerenciamento pelo GruntJS é realizado através de tarefas configuradas no arquivo *gruntfile*. Com o *plug-in grunt-git-hooks*, configurado conforme imagem no item 2 da seção 4 (refere-se à Figura 6), ao se efetuar um *commit* o GruntJS interpreta o arquivo *gruntfile* verificando as tarefas que estão configuradas e, se houver, a sequência de tarefas que deverá ser

executada. No exemplo utilizado, após o *git-hook*, será executado a tarefa [*karma:test*] (linha 28 da Figura 6) que inicia a execução dos testes jasmine de acordo com a configuração do arquivo *karma.conf.js*.

Para verificar o funcionamento do gerenciamento proposto, foi realizado o teste com uma classe de teste hipotética "*myFunction*" que contém uma função para concatenar texto, conforme é apresentado na Figura 8.

```
1  if (typeof NS == 'undefined') { NS = {}; }
2
3  NS.myFunction = {
4      //replace "//will add new functionality here later" with the following:
5      append: function append(string1, string2) {
6          return string1 + string2;
7      }
8  };
```

Figura 8: Classe "*myFunction*" em JavaScript

O arquivo com os testes unitários para a classe *myFunction*, escritos em Jasmine, é apresentado na Figura 9. Observa-se dois casos de teste para a função de concatenar texto. O primeiro caso de teste, na linha 6, verifica se a função existe na classe e o segundo caso de teste, na linha 9, verifica se a saída da função corresponde ao resultado esperado ao se passar duas variáveis de entrada.

```
1  describe("myFunction", function() {
2      var myfunc = NS.myFunction;
3
4      //replace "//will insert additional tests here later" with the following:
5      describe("appending strings", function() {
6          it("should be able to append 2 strings", function() {
7              expect(myfunc.append).toBeDefined();
8          });
9          it("should append 2 strings", function() {
10             expect(myfunc.append('hello ','world')).toEqual('hello world');
11         });
12     });
13 });
```

Figura 9: Teste unitário escrito em Jasmine

```
1  var models = require ('../../app/models');
2
3  describe('Models', function() {
4
5      describe('Contact', function() {
6          var schema = models.Contact.schema.paths;
7
8          // verifica se o modelo de dados exite no banco
9          it('should exist', function() {
10             expect(models.Contact).to.exist;
11         });
12
13         // verifica se o modelo de dados possui
14         // o campo de email em formato string
15         it('should have email string field', function() {
16             expect(schema.email).to.exist;
17             expect(schema.email.instance).to.equal('String');
18         });
19
20         // verifica se o modelo de dados possui
21         // o campo de nome e sobrenome em formato string
22         it('should have name string field', function() {
23             expect(schema['name.first']).to.exist;
24             expect(schema['name.first'].instance).to.equal('String');
25             expect(schema['name.last']).to.exist;
26             expect(schema['name.last'].instance).to.equal('String');
27         });
28
29         // verifica se o modelo de dados possui
30         // o campo de telefone em formato string
31         it('should have phone string field', function() {
32             expect(schema.phone).to.exist;
33             expect(schema.phone.instance).to.equal('String');
34         });
35
36         // verifica se o modelo de dados possui
37         // o campo de avatar em formato string
38         it('should have gravatar string field', function() {
39             expect(schema.gravatar).to.exist;
40             expect(schema.gravatar.instance).to.equal('String');
41         });
42     });
43 });
```

Figura 10: Teste do Modelo de Dados escrito em Jasmine

Para os casos de teste de integração foi utilizado um modelo de dados hipotético "*Contact*", para o qual se verifica a estrutura do esquema de dados e a correta inserção de instâncias nessa base de dados. A estrutura contém e-mail, nome, sobrenome, telefone e uma imagem de perfil, sendo que os testes são realizados sobre essa estrutura. A Figura 10 mostra o caso de teste escrito em Jasmine para esse modelo de dados. Na linha 8, verifica-se se a base existe no banco de dados e na linha 15, verifica-se se o campo e-mail se encontra no formato texto. Já na linha 22, verifica-se se o objeto do tipo "*name*" possui os campos de primeiro nome e último nome em formato texto na base. Na linha 31 temos o teste que verifica se o campo telefone existe e

está no formato texto e por fim na linha 38 o teste verifica se o campo avatar existe na base e se o formato do campo é texto, uma vez que salvamos imagens convertidas em texto no banco de dados.

Para verificar a correta inserção dos dados nesse banco de dados foi criado um caso de teste que deve retornar uma mensagem positiva em caso de sucesso e uma mensagem negativa caso contrário. Esse caso de teste é apresentado na Figura 11. Na linha 2, criamos um objeto contato antes da execução do teste para ser utilizado no teste de inserção. Na linha 12, verificamos se a função "*add*" existe no escopo da classe "*Contacts*". Na linha 17 é realizado a chamada da função "*add*" passando como parâmetro o objeto contato de teste (req) e o objeto de resposta esperado (res). Já na linha 18, testamos se o objeto de retorno é um formato json semelhante ao objeto json de entrada do contato teste.

```
1  describe('add', function() {
2      beforeEach(function() {
3          req.body = {
4              name: 'testing',
5              email: 'test@testing.com',
6              phone: '123-456-7890'
7          };
8      });
9
10     // verifica se a função de adicionar contato existe
11     it('should be defined', function() {
12         expect(contacts.add).to.be.a('function');
13     });
14
15     // verifica se o contato foi salvo com sucesso
16     it('should return json on save', function() {
17         contacts.add(req, res);
18         expect(res.json).calledWith(req.body);
19     });
20 });
```

Figura 11: Teste de Inserção de Dados na Base escrito em Jasmine

4.6 CONSIDERAÇÕES FINAIS SOBRE O CAPÍTULO

Neste capítulo foram apresentados a arquitetura e proposta para o ambiente de IC. Além disso foi exemplificado a conexão do NodeJS com o GruntJS, Git e os diversos exemplos de testes utilizados na arquitetura. O próximo capítulo apresenta os resultados obtidos.

5 ESTUDO DE CASO, RESULTADOS E DISCUSSÃO

Esse capítulo apresenta o Estudo de Caso utilizado para se validar a proposta, os resultados obtidos com base na comparação de Sprint de desenvolvimento que utilizaram a arquitetura contra outros que não a utilizaram e a discussão e justificativas desses resultados.

5.1 ESTUDO DE CASO

O sistema que foi utilizado para a prova de conceito é um sistema real de uma empresa cliente da empresa que emprega o autor. É um sistema legado para Mainframe (alta plataforma) utilizado para o gerenciamento de parâmetros de cartão de crédito, que permitia a qualquer usuário do sistema alterar parâmetros sem realizar nenhum tipo de validação ou confirmação, ocasionando alterações equivocadas, mesmo em ambiente produtivo. Por ser um sistema crítico que envolve a manipulação de grandes valores monetários (já que os parâmetros indicam as taxas que serão cobradas nas transações dos cartões de crédito), uma alteração inadequada pode significar uma perda considerável para o cliente. Sendo assim, a necessidade do cliente era que fosse desenvolvido um novo sistema, em baixa plataforma, que permitisse a dupla checagem dos parâmetros alterados, ou seja, um usuário altera e outro usuário valida a alteração dos parâmetros antes de transferi-los para o Mainframe.

O desenvolvimento desse novo sistema Web integrado ao sistema Mainframe, utilizou a metodologia Scrum, sendo dividido em pequenas partes entregáveis. Ao todo foram planejados 12 Sprint separados de acordo com as funcionalidades e complexidades da parte do sistema do Mainframe ao qual as novas funcionalidades deveriam ser integradas.

O sistema web criado era um sistema simples que realizava a inserção de um novo parâmetro ou a atualização de um parâmetro existente no sistema Mainframe, a partir de layouts extraídos das telas do Mainframe. A complexidade se encontrava na integração com o Mainframe que para algumas funcionalidades haviam regras de negócio específicas que impossibilitavam o sistema web realizasse sua tarefa, sendo necessário criar uma customização no fluxo para atender a regra de negócio. Inicialmente, o projeto não contemplava testes automáticos.

Para o primeiro Sprint foram escolhidas duas telas do Mainframe que continham os parâmetros das anuidades dos cartões de credito Visa e Mastercard, sendo que o sistema web atualizava as taxas existentes ou criava uma taxa para um novo banco emissor.

As demais estórias do usuário consistiam em alterar um parâmetro Visa, alterar um parâmetro Mastercard, criar uma nova taxa para um novo banco emissor Visa e criar uma nova taxa para um novo banco emissor Mastercard. O módulo contemplando essas estórias foi testado manualmente pelos desenvolvedores e logo em seguida pelos usuários.

5.2 VALIDAÇÃO

Para validar a arquitetura e entender o ganho da equipe ao utiliza-la, os testes executados pela ferramenta GruntJS, seguindo as práticas estudadas na fase de análise, foram monitorados ao longo das execuções. Um relatório é gerado a cada execução de testes contendo informações sobre o tempo gasto no teste e se o teste foi executado com sucesso ou não. Os resultados obtidos são comparados com dados de um ou mais sistemas, de igual complexidade, que não utilizaram a arquitetura com testes automáticos.

Com o resultado apresentado pelo relatório além do tempo despendido para o teste é possível identificar se houve uma detecção antecipada de falhas pelos desenvolvedores. Além disso, é possível identificar quais testes demoraram mais para serem executados no ambiente integrado, fazer um diagnóstico do motivo dessa demora e quais testes se mostraram mais críticos.

5.2.1 PRIMEIRA VALIDAÇÃO

Nesse ponto do desenvolvimento, não houve complexidade em validar o modelo de sistema proposto, uma vez que a alteração era composta de telas simples onde um usuário alterava um parâmetro, outro usuário confirmava a alteração e, após a confirmação, o sistema simulava um robô conectando ao mainframe e alterando o parâmetro na tela informada. Nesse primeiro Sprint não houve defeitos.

Uma vez validado, novas telas foram adicionadas aos Sprint seguintes no sistema mainframe, sendo: 2 telas alocadas no Sprint 2; 6 telas no Sprint 3; 5 telas no Sprint 4. Por essas telas apresentarem uma complexidade maior e regras de negócio que não foram informadas aos desenvolvedores, os testes preliminares apresentavam falsos positivos, uma vez que o desenvolvedor alterava um parâmetro (por exemplo, a taxa do Visa Platinum de 5% para 10%) e o retorno era sucesso no sistema web, mas o usuário reportava como falha, pois a alteração não havia se concretizado no Mainframe. Assim, dos 8 casos de testes, todos foram reportados com falha.

Para este caso, foi necessário que o robô simulador, além de acionar a tecla “Enter”, acionasse também a tecla “F3” para confirmar a alteração. Com base nessas informações, foram criadas customizações no robô simulador para essas telas específicas.

No Sprint 3, os desenvolvedores adiantaram os problemas encontrados anteriormente e resolveram fazer um teste preliminar diretamente no mainframe para verificar quais telas teriam fluxos alternativos e quais poderiam seguir o fluxo padrão. Com isso, foram identificados que 4 telas do Sprint 3 teriam fluxo alternativo. Porém, perdeu-se muito tempo nesse teste manual e a entrega do Sprint 3 afetou os Sprints 1 e 2 que já estavam corretos.

No Sprint 4, o desenvolvedor estava sobrecarregado com as correções pendentes, além da verificação e desenvolvimento do próprio Sprint 4. Por conta disso, mesmo com os testes realizados antes da entrega, não houve uma melhora na qualidade, pois muitas falhas que haviam sido corrigidas retornaram, ocasionando um descontentamento por parte do cliente.

Diante deste cenário, surgiu a necessidade de se automatizar os testes para agilizar o processo de desenvolvimento e melhorar a qualidade das entregas. Houve uma resistência ao processo por parte dos gerentes da empresa, que não conseguiam entender as vantagens que a arquitetura de teste de integração contínua trazia. Acreditavam que o tempo gasto no desenvolvimento do teste poderia ser melhor aproveitado corrigindo os defeitos ou na construção dos outros Sprints.

Assim, como piloto da arquitetura de testes proposta, foi construído os testes unitários em Jasmine, utilizados para validar cada novo layout que era disponibilizado pela equipe de desenvolvimento do sistema Mainframe. O desenvolvedor disparava a execução dos testes manualmente e assim era possível identificar problemas no formato do layout recebido, agilizando a correção dos mesmos pela equipe.

Com essa nova dinâmica de testes unitários e a melhora na velocidade de identificação de problemas de layout e correção pela equipe Mainframe, os gerentes do projeto começaram a perceber as vantagens da automação de testes no desenvolvimento de sistemas. Assim, foi permitida a criação dos primeiros testes de integração. Os testes de integração, inicialmente, testavam a criação de uma manutenção de parâmetro no layout fornecido.

Com esses testes, foi possível identificar mais rapidamente quando o novo layout inserido não se encaixava no fluxo principal do sistema, sendo necessário uma customização no sistema web para então esse novo layout funcionar corretamente. A Tabela 2 apresenta o comparativo do *Status Report* do cliente antes dos testes automáticos e após a execução com testes automáticos.

Pelos resultados apresentados, observa-se que, para os mesmos casos de testes houve um aumento de entrega correta da ordem de 50% passando de 24 para 36 casos de testes que resultaram em sucesso, enquanto que os casos de testes com algum defeito reduziram de 18 para 6. Esse resultado validou a eficiência da arquitetura de testes e trouxe visibilidade ao projeto e prestígio à equipe dentro da empresa. A melhoria observada só foi possível com a aplicação da arquitetura de integração continua, pois, as falhas foram identificadas e corrigidas previamente.

A Figura 12 apresenta o ciclo de vida do desenvolvimento com a IC. Os desenvolvedores realizam o *commit* do código (1), o detector está monitorando os arquivos (2) e identifica as alterações iniciando a execução dos testes automaticamente (3). Após a execução dos testes é gerado um relatório com as informações de sucesso ou falha dos testes (4). No caso de falhas o teste é parado alertando o desenvolvedor (5), onde o desenvolvedor corrige e realiza o *commit* da correção (6) iniciando o processo novamente (7). No caso de sucesso o fluxo termina exibindo o relatório de sucesso e de cobertura (versão expandida) com todos os testes ok (4).

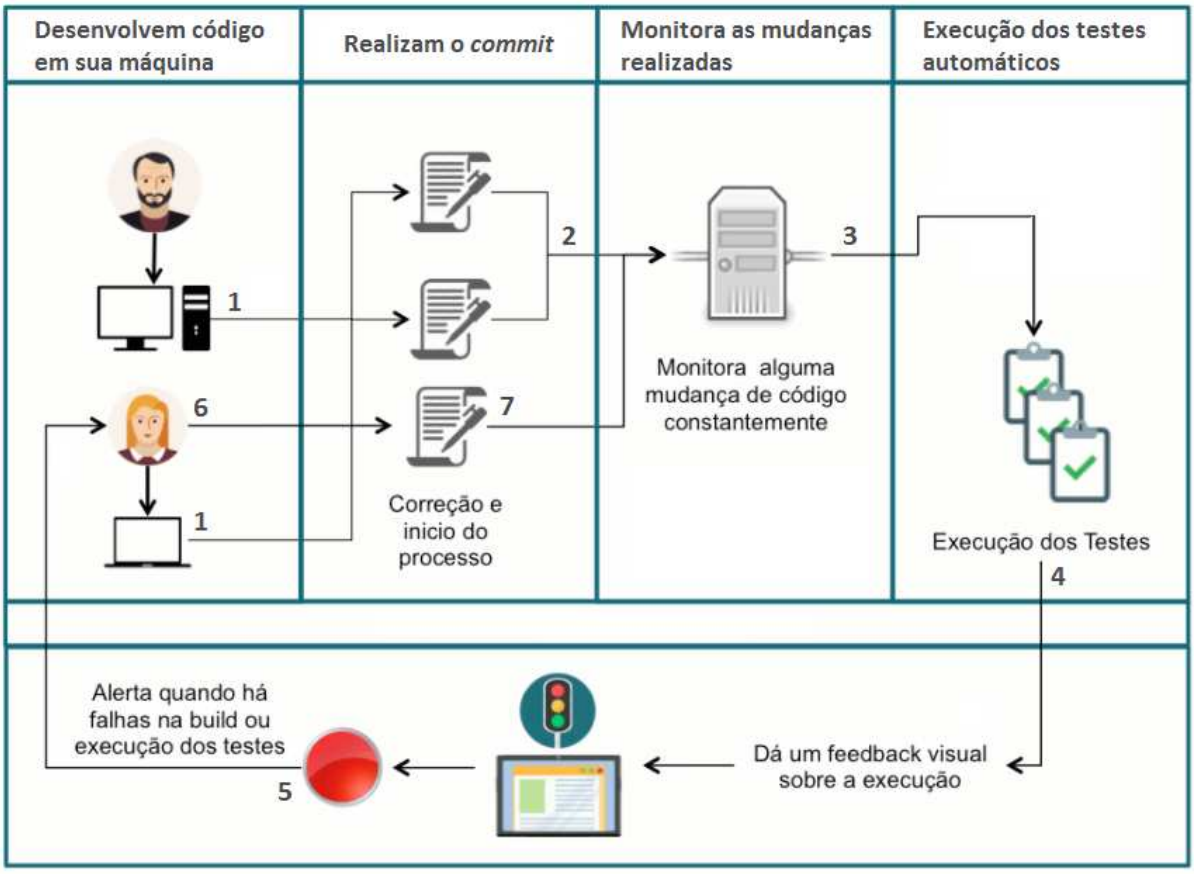

Figura 12: Ciclo de vida com a IC

Tabela 2: Comparativo do *Status Report*

Módulo 6	Antes	Depois
Casos de Testes OK	**24**	**36**
Casos de Testes NOK	**18**	**06**
Casos de Testes NA	**09**	**09**

OK – Sucesso / NOK – Não Sucesso / NA – Não Se Aplica

Em seguida, foram desenvolvidos os testes de integração que realizam a inserção de um novo parâmetro no sistema Mainframe para o layout de manutenção, sendo que o fluxo de inserção não segue o mesmo fluxo de manutenção. Para cada layout novo ou atualizado do Mainframe, os testes de regressão eram executados manualmente pelo desenvolvedor. A Tabela 3 mostra a continuidade dos testes nos demais módulos.

Quando a execução dos casos de Teste foi feita manualmente (Modulo 5), dos 36 casos de Teste que se poderia aplicar a arquitetura, apenas 16 resultaram sucesso, representando aproximadamente 44% de sucesso. Em todos os demais Módulos em que foi feita a execução de forma automática utilizando-se a arquitetura proposta, a taxa de sucesso ficou acima de 88% com a única exceção para os testes do Modulo 12 com 66% de sucesso, sendo mesmo assim 20% acima do resultado obtido nos testes manuais. Esse resultado, somente foi possível com a aplicação da arquitetura proposta utilizando a IC.

Tabela 3: Resultado das entregas dos Módulos

	Módulo 5	Módulo 8	Módulo 9	Módulo 10	Módulo 11	Módulo 12
Casos de Testes OK	**16**	**32**	**40**	**61**	**31**	**08**
Casos de Testes NOK	**20**	**04**	**02**	**07**	**02**	**04**
Casos de Testes NA	**11**	**00**	**00**	**13**	**00**	**00**

OK – Sucesso / NOK – Não Sucesso / NA – Não Se Aplica

Além disso, houve um significativo ganho no tempo despendido na execução dos casos de teste que demandavam em média cinco minutos para a execução de cada caso de teste, passando a despender dois minutos quando executados pelo GruntJS, uma redução de 60% no tempo de

execução. Apesar desse resultado bastante positivo, notou-se que quatro testes específicos, com muitas regras de negócios despenderam mais de uma hora para serem executados pelo GruntJS. Embora ainda assim o tempo era bem menor que o despendido pelos testes manuais, que era de cinco horas, os desenvolvedores optaram por retirar esses testes da fila do GruntJS seguindo a priorização dos casos de testes pelo menor tempo x maior cobertura (SJF - *Shortest Job First*).

A arquitetura prevê essa abordagem de priorização [49] de acordo com as definições do projeto, porém essa priorização não é feita de forma automática. Sendo assim, esses testes específicos eram executados manualmente pelos desenvolvedores a cada dois meses já que eram testes de fluxos muito específicos e que tinham baixa prioridade para o negócio do cliente.

Com todos os testes construídos, o próximo passo seria executar o processo automaticamente quando houvesse alguma alteração no sistema feita pelos desenvolvedores. Porém, devido ao tempo desse projeto e algumas particularidades desse sistema especifico onde o teste de layout era executado antes mesmo de fazer evoluções no sistema, não foi possível completar fluxo de automatização antes do fim do projeto.

Com a arquitetura de testes aplicada, a entrega dos módulos teve em média de duas a quatro falhas durante a fase de aceitação do cliente, sendo que na maioria dos casos, essas falhas eram referentes a regras de negócio que os desenvolvedores não tinham conhecimento. Assim, com o ambiente estável, o autor deste projeto foi alocado em outro projeto que ainda não utilizava a arquitetura proposta.

O projeto piloto da arquitetura começou a apresentar uma grande quantidade de falhas reportado pelo cliente. Foi constatado que o desenvolvedor que assumiu a continuação do projeto não executava a rotina de testes, por não acreditar na arquitetura. Assim muitas falhas que poderiam ter sido identificadas antes da entrega do módulo não foram detectadas, comprometendo o nome da empresa perante o cliente.

Como medida de segurança o autor voltou ao projeto piloto e reforçou a utilização do processo de testes e com essa visibilidade foi possível expandir a arquitetura de testes para outros projetos na empresa.

5.2.2 SEGUNDA VALIDAÇÃO

O segundo projeto que utilizou a arquitetura, era um sistema de extrato online de estabelecimentos comerciais que gostariam de verificar o extrato das vendas realizadas por período. Quando o autor foi inserido ao projeto, o mesmo já se encontrava com um grande número de falhas decorrentes dos diversos *commits* realizados pelos desenvolvedores que,

muitas vezes, perdiam algumas correções já realizadas, passando a percepção de falta de qualidade na entrega do sistema ao cliente.

Para aplicar a arquitetura proposta seria necessário que os casos de teste para o sistema estivessem desenvolvidos. Isso não se verificou, uma vez que os testes eram feitos de maneira *ad hoc* até então. Novamente, a necessidade de mudança nos processos gerou descontentamento inicial entre os desenvolvedores da nova equipe, uma vez que seria necessário esforço adicional para o desenvolvimento dos casos de teste. Depois da conscientização dos benefícios que traria a aplicação da arquitetura, a equipe se engajou para efetuar as mudanças.

Foi realizada uma força tarefa para criar os testes necessários e o autor acrescentou um *plug-in* "*karma-coverage*", que verificava a cobertura de testes, ou seja, quanto do código criado estaria sendo coberto pelos testes, incrementando a arquitetura proposta. Na Figura 13 temos a nova configuração do plug-in no arquivo "*karma-config*" que foi apresentado na seção 3 do capitulo 4. Na linha 24 adicionamos a referência ao plug-in utilizado e na linha 15 definimos o formato de saída do relatório gerado sobre a cobertura.

Assim, a cada *commit* realizado, antes de executar o teste automaticamente, era gerado um relatório informando a cobertura dos testes. Com esse novo relatório era possível identificar quais partes do código não estavam cobertas com testes.

Assim os desenvolvedores criavam os testes faltantes e, à medida que realizam o *commit* do teste, o sistema executava o teste de regressão automaticamente seguindo a arquitetura que foi apresentada na seção 1 do capitulo 4 e gerava um novo relatório com o status de cobertura obtida.

```
1  module.exports = function(config) {
2    config.set({
3      // Linguagem assertiva que será utilizada
4      frameworks: ["jasmine"],
5
6      // Lista de arquuivos de testes
7      files: [
8        "app/js/*.js",
9        "app/js/**/**/*.js",
10       "test/js/*.js",
11       "test/js/**/*.js"
12     ],
13
14     // Formato do arquivo report de cobertura
15     coverageReporter: {
16       dir: "reports/coverage/",
17       reporters: [
18         { type: "html", subdir: "report-html" },
19         { type: "cobertura", subdir: ".", file: "cobertura-coverage.xml" }
20       ]
21     },
22
23     // Plug-ins utilizados
24     plugins : [
25       "karma-coverage",
26     ],
27
28     // Porta web
29     port: 9876,
30
31     // Nivel de log utilizado
32     logLevel: config.LOG_INFO,
33
34     // Continuous Integration mode
35     singleRun: true
36   });
37 };
```

Figura 13: Exemplo do arquivo *karma.conf.js*

A Figura 14 apresenta o teste de integração do módulo de vendas. A linha 11 carrega as bibliotecas de terceiros necessárias para o sistema. Das linhas 15 a 40 são inseridos *mocks* das funções das bibliotecas carregadas. São funções que não fazem parte do escopo do segundo projeto, uma vez que são bibliotecas de terceiros já garantidas pelo seu fornecedor. Na linha 44 a 48 criamos um *mock* para a função *$window* do navegador, pois alguns métodos das classes utilizam essa função para realizar o "*pageBack*" no sistema. Nas linhas 51 a 60 foi inserido um *mock* da biblioteca de gerar relatório *.csv* e *.pdf*, assim será possível testar a função sem efetivamente criar um arquivo. Nas linhas 63 e 64 são carregados os contextos do módulo vendas e seu dependente, o módulo de comunicação (que faz a comunicação interna do sistema com a base de dados Mainframe). A linha 70 e 71 criam o contexto do *controller* que será testado e seus serviços. A linha 76 a 84 realiza os testes reais nas funções do *controller* especificado sendo, neste caso, *controller-venda-data*.

```
/**
 * @ngdoc object
 * @name controller venda data
 *
 * @description
 * Roteiro de teste unitário.<br/>
 */
describe("CRE - ctrl-venda-data.", function () {

    beforeEach(function () {
        angular.module("arq-spa-base.navegacao", []);
    });

    // Criar mocks dos providers da arquitetura
    beforeEach(module(function ($provide) {
        $provide.provider("sfNavegador", function () {
            this.adicionarFluxoNavegacao = jasmine.createSpy("adicionarFluxoNavegacao");
            this.criarFluxoNavegacao = function () {
                return {
                    adicionarEstado: function () { return this; },
                    definirEstadoInicial: function () { return this; }
                }
            };
            this.$get = function () {
                return {
                    navegar: jasmine.createSpy("navegar"),
                    iniciarFluxo: jasmine.createSpy("iniciarFluxo"),
                    encadeamento: function() { return {}; },
                    voltar: jasmine.createSpy("voltar"),
                };
            };
        });
        $provide.service("sfContexto", function () {
            this.obterValorContextoTrabalho = function(){ return {};};
            this.obterValorContextoSessao = function() { return "123132"; };
            this.definirValorContextoTrabalho = jasmine.createSpy("definirValorContextoTrabalho");
        });
        $provide.service("sfMemorizador", function () {
            this.obter = function() { return {}; };
        });
    }));

    // Mock do provider window
    beforeEach(module(function ($provide) {
        $provide.service("$window", function() {
            return { ga: function() {} };
        });
    }));

    // Mock do serviceExport
    beforeEach(module(function ($provide) {
        $provide.service("serviceExport", function () {
            this.relatorio_ajustes_data = function() { return { save: function() {} } };
            this.relatorio_extrato_consolidado = function() { return { save: function() {} } };
            this.relatorio_lancamentos_futuro_data = function() { return { save: function() {} } };
            this.relatorio_lancamentos_realizado_data = function() { return { save: function() {} } };
            this.relatorio_venda_data = function() { return { save: function() {} } };
            this.exportCSVNew = function() { return null; };
        });
    }));

    // Carrega o módulo no contexto
    beforeEach(module("vendas"));
    beforeEach(module("comunicacao"));

    // Pega o controller
    var $controller;
    var executeServiceMock;
    beforeEach(inject(function(_$controller_, _executeServiceMock_){
        $controller = _$controller_;
        executeServiceMock = _executeServiceMock_;
    }));

    // Executa os testes
    it("fluxo padrão", function () {
        var ctrl = $controller("ctrl-venda-data", { executeService: executeServiceMock });
        ctrl.filtroPeriodo();
        ctrl.navegar({});
        ctrl.visualizarMais();
        ctrl.dataInicial = new Date();
        ctrl.dataFinal = new Date();
        ctrl.exportarPDF();
        ctrl.exportarCSVClick();
        ctrl.goBack();
    });

});
```

Figura 14: Teste de integração do módulo de vendas

Considerando o prazo do projeto, foi definido que para cada classe de código, um mínimo de 70% do código deveria estar coberto com testes. Com a execução dos casos de testes de regressão de forma automática, verificamos que o caso de teste apresentado na Figura 13, garantiu 86.89% de cobertura de código para o *controller-venda-data* conforme apresentado na Figura 15.

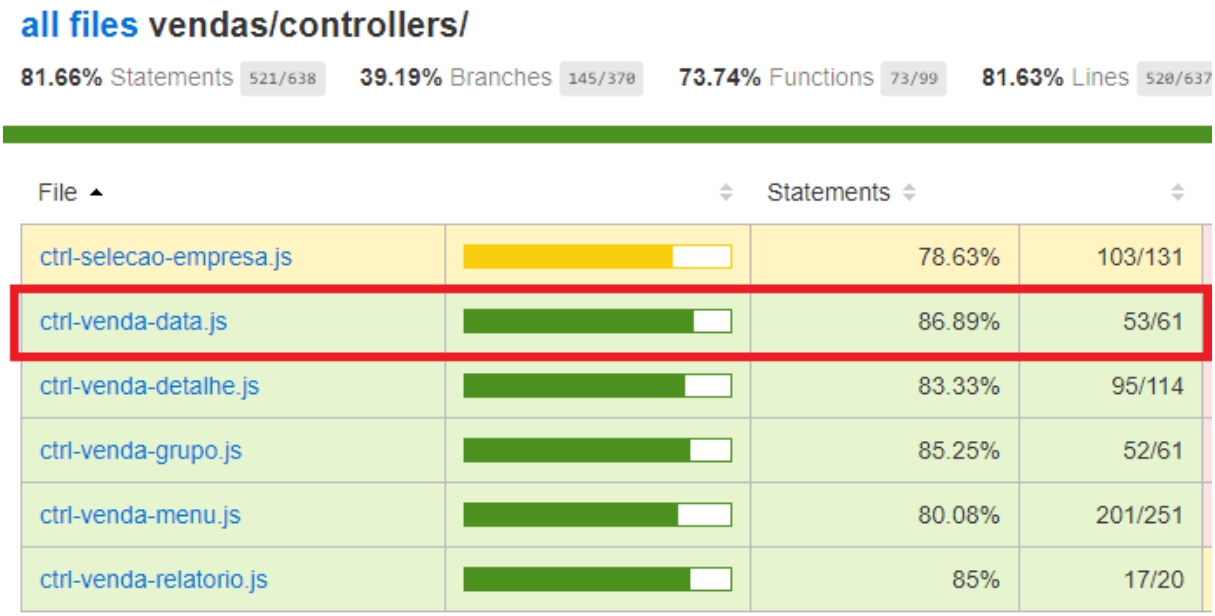

Figura 15: Porcentagem da cobertura de testes da classe venda-data

Com os testes de regressão sendo executados automaticamente, o segundo projeto apresentou uma grande melhoria na detecção de falhas. Foram registradas em média de 3 a 5 falhas nos testes feitos pelo cliente, que eram referentes a regras de negócio mal definidas, contra uma média de 15 falhas apresentadas no mesmo período (por semana) antes da implementação dos testes, sendo que esses eram, em sua maioria, falhas recorrentes.

Depois dos resultados positivos, os desenvolvedores concordaram e entenderam os benefícios da prática apresentada, onde ao realizar o *commit*, o sistema executava os testes automaticamente e detectava falhas nos módulos devido as alterações. Além das falhas detectadas, também eram detectados os testes que não haviam sido atualizados, decorrente das alterações nos módulos.

Foi comprovado que a velocidade de correção das falhas aumentou uma vez que a identificação do defeito no momento do *commit*, auxiliava a correção das falhas, uma vez que o desenvolvimento do código que tinha sido integrado ainda era recente.

Embora tenha havido uma melhoria significativa dos processos de teste na empresa ainda é necessário um certo amadurecimento relacionado a arquitetura. Mesmo depois de acata-la, os desenvolvedores ainda cometem enganos na sua utilização e, por exemplo, esquecem de corrigir ou atualizar os testes. Nesse caso, para cada manutenção no código é feito, em média, dois *commits*. O primeiro para enviar a alteração do código ao versionador, quando o teste falha devido a alteração necessária nos códigos fontes dos testes, e o segundo *commit* com a atualização desses testes.

Para auxiliar os desenvolvedores para a adoção da arquitetura, o autor escreveu um guia de boas práticas que pode ser encontrado em [68].

5.3 CONSIDERAÇÕES FINAIS SOBRE O CAPÍTULO

Neste capítulo foram apresentados dois projetos que utilizaram a arquitetura de testes em um ambiente de integração contínua, que foi sugerida neste trabalho. A aplicação da arquitetura foi utilizada como prova de conceito da proposta.

Os resultados obtidos com a utilização da arquitetura e sua validação para o ambiente de IC para os sistemas submetidos aos testes, reduziram significativamente a quantidade de defeitos na entrega ao cliente.

O próximo capítulo apresenta nossa conclusão e sugestões para trabalhos futuros.

6 CONCLUSÃO E TRABALHOS FUTUROS

Considerando os desafios para manter a construção de seus sistemas com número de falhas reduzido, os desenvolvedores de empresas de produtos de *software* buscam ferramentas para automatizar a execução do teste. Este trabalho teve como objetivo oferecer uma alternativa que contribua para a melhoria dos sistemas desenvolvidos em ambientes de integração contínua, visando a qualidade do *software*. Foram descritos os conceitos de automação de testes utilizados para alcançar o objetivo proposto. Ainda foram apresentadas ferramentas de integração contínua e as vantagens de sua utilização.

O objetivo principal deste trabalho foi apresentar uma arquitetura de integração capaz de executar automaticamente vários tipos de testes. Esta demanda surgiu na empresa onde o autor trabalha e foi confirmada após pesquisa na literatura e no estudo de ferramentas existentes.

Com base nesses estudos, observou-se que as ferramentas existentes para realizar a integração automática e os testes de regressão nesse ambiente de integração contínua são poucas e não se enquadram aos testes de sistemas modernos. Além disso, há uma falta de definição sobre os ambientes integrados.

Dessa forma, foi proposta uma solução para testes em um ambiente de integração contínua que utilizou a ferramenta GruntJS, escolhida para gerenciar o ambiente utilizado para construir e testar os produtos de *software*. O ambiente ainda integra a ferramenta GIT para controlar o versionamento do produto em teste e o NodeJS para identificar as mudanças e ativar o processo de teste com a ajuda do GruntJS.

A validação da arquitetura proposta foi feita utilizando-se um experimento real. Os produtos de *software* em teste são sistemas Javascript que foram submetidos a um conjunto de testes básicos (de unidade, de integração e de regressão) escritos em Jasmine. Os resultados mostram uma diminuição significativa de falhas identificadas em tempo de entrega, quando comparamos com cópias dos mesmos sistemas que não foram submetidos ao processo de IC. Foi constatado que essa melhoria ocorreu porque a maioria das falhas foram identificadas e corrigidas previamente, ao longo do processo de IC. Os resultados sugerem que a solução traz fortes indícios de melhoria na interação com o cliente e na qualidade dos sistemas desenvolvidos em ambientes de IC, tendo reduzido significativamente o número de falhas na entrega do produto. A avaliação da empresa também foi positiva e em decorrência decidiu adotar esse procedimento para os próximos projetos.

As contribuições deste trabalho incluem, além da arquitetura, a implementação de um ambiente de integração contínua capaz de realizar, automaticamente, uma bateria de testes em

sistemas que utilizam tecnologias mais recentes. Essa contribuição é genérica o suficiente para permitir que tanto a arquitetura quanto o ambiente de IC possam ser utilizados para outros sistemas ou empresas, aproximando as áreas da tecnologia, computação e de negócio da empresa.

A arquitetura foi útil por padronizar a forma de construir a IC e por nivelar todos o conhecimento de todos desenvolvedores que utilizaram a IC. Porém, uma das limitações é a sobrecarga inicial em um desenvolvedor para a montagem inicial do ambiente, onde não foi encontrado um meio de paralelizar as atividades de instalações e configurações. Outra limitação encontrada no processo foi doutrinar os desenvolvedores para realizarem o *commit* com frequência evitando assim conflito de código.

Convencer os gerentes de projeto a adotarem a arquitetura é uma tarefa árdua e nem sempre rápida de se realizar, além disso, montar a arquitetura para projetos pequenos também pode incorrer custos adicionais com treinamento dos desenvolvedores, mas mesmo diante desses fatores negativos, a abordagem ainda se faz eficiente e traz ganhos a médio e longo prazo, principalmente na cultura da equipe já que o TDD é opcional, mas fortemente recomendável.

Uma vez configurado corretamente, o conhecimento foi difundido e a automatização ajudou nos processos internos da empresa, da elaboração do escopo, da estimativa das atividades e do processo de reportar as falhas encontradas, contribuído muito em velocidade. Embora inicialmente aplicar a arquitetura tenha sido custoso, tanto por convencer os gerentes quanto por doutrinar os desenvolvedores, os ganhos foram significativos na qualidade dos sistemas desenvolvidos, melhorando a relação empresa-clientes.

Para trabalhos futuros pretende-se aplicar a solução utilizando-se casos de teste mais complexos como por exemplo testes funcionais e testes de usabilidade, para um grande conjunto de sistemas que devem ser submetidos ao ambiente de IC. Outra vertente de evolução deste trabalho é a junção do GruntJS com a prática Pipeline da ferramenta Jenkins, sendo uma solução sugerida na literatura recente a ser investigada em trabalhos futuros.

REFERÊNCIAS

[1] Portal do *Software* Público Brasileiro - Integração Contínua. Website. Disponível em http://www.softwarepublico.gov.br/5cqualibr/xowiki/integracaoContinua. Último acesso em 12/09/2015.

[2] Gartner - Integração de Aplicativos e Serviços na Web. Abril 2004, Rio de Janeiro.

[3] Fernandes S. L. P. Maturidade da Integração Contínua. Website. Disponível em https://blog.octo.com/pt-br/maturidade-da-integracao-continua. Último acesso em 15/09/2017.

[4] Fowler, M. "The Benefits of Continuous Integration". Website. Disponível em http://martinfowler.com/articles/originalContinuousIntegration.html. Último acesso em 12/09/2015.

[5] Rocha F. G. Integração contínua: uma introdução ao assunto. Website. Disponível em http://www.devmedia.com.br/integracao-continua-uma-introducao-ao-assunto/28002#ixzz3GQBu1OLf. Último acesso em 12/09/2015.

[6] Oliveira, P. A. C. de; Nelson, M. A. V. Integração de ferramentas para minimizar erros provenientes de efeitos colaterais inseridos durante a manutenção. PUC Minas, 2007.

[7] Agile Data. Introduction to Test Driven Development (TDD). Website. Disponível em http://www.agiledata.org/essays/tdd.html. Último acesso em 09/12/2015.

[8] Agile Alliance. Behavior Driven Development (BDD). Website. Disponível em http://guide.agilealliance.org/guide/bdd.html. Último acesso em 09/12/2015.

[9] Humble, J. Continuous Delivery. Website. Disponível em http://continuousdelivery.com/. Último acesso em 26/09/2015.

[10] Forrester Consulting. *Continuous Delivery: A Maturity Assessment Model*. Website. Disponível em http://info.thoughtworks.com/rs/thoughtworks2/images/Continuous%20Delivery%20_%20A%20Maturity%20Assessment%20ModelFINAL.pdf. Último acesso em 26/09/2015.

[11] Myers, G. J.; Sandler, C.; Badgett, T. *The art of software testing*. 3.ed. John Wiley & Sons, 2011.

[12] Pádua Filho, P.; W. de. Engenharia de software: fundamentos, métodos e padrões. 3.ed. Rio de Janeiro: LTC, 2009.

[13] Barbosa, E.; Maldonado, J.C.; Vincenzi, A.M.R.; Delamaro, M.E; Souza, S.R.S. e Jino, M. Introdução ao Teste de Software. XIV Simpósio Brasileiro de Engenharia de Software, 2000.

[14] Rocha, A. R. C., Maldonado, J. C., Weber, K. C. Qualidade de software – Teoria e prática. São Paulo: Prentice Hall, 2001.

[15] Pressman, R. S. Engenharia de Software: Uma abordagem Profissional. 7.ed. Porto Alegre: Bookman, 2011.

[16] Meszaros G. "xUnit Test Patterns – Refactoring Test Code". Pearson Education Inc., Boston. ISBN-10: 0131495054. Disponível em http://martinfowler.com/books/meszaros.html. Último acesso 16/04/2018.

[17] Larman C. Utilizando UML E Padrões, 3. eds., Porto Alegre. Editora Bookman, 2007. Livro referente ao modelo domínio do sistema para aplicação dos testes.

[18] Pezzè, M.; Young, M. Teste e Análise de Software: processos, princípios e técnicas. Porto Alegre. Bookman, 2008, 512p.

[19] Palermo J. Guidelines for Test-Driven Development. Disponível em: https://msdn.microsoft.com/en-us/library/aa730844%28v=vs.80%29.aspx. Último acesso em 15/04/2018.

[20] SOMMERVILLE, Ian. Engenharia de software. 6.ed. São Paulo: Person, 2010.

[21] Algar Tech. Guideline: Guia de Estratégias de Integração. Website. Disponível em http://minds.synos.com.br/minds/synosup/guidances/guidelines/guia_integracao_4FAA991.html. Último acesso em 02/10/2016.

[22] Fowler M. Eradicating Non-Determinism in Tests. Disponível em http://martinfowler.com/articles/nonDeterminism.html. Último acesso em 15/04/2018.

[23] Rothermel, G.; Elbaum, S.; Malishevsky, A.; Kallakuri, P.; Qiu, X. On Test Suite Composition and Cost-Effective Regression Testing, 2003

[24] Gouveia, C. C.; Oliveira, J. C.; Filho, R. Q., A way of Improving Test Automation Cost-Effectiveness. CAST 2006, Indianápolis, EUA.

[25] BRUNELI, M. V. Q. A utilização de uma metodologia de teste no processo da melhoria da qualidade do software. Tese, Campinas, 2006.

[26] ROCHA, A. R. C., MALDONADO, J. C., WEBER, K. C. Qualidade de software – Teoria e prática. São Paulo: Prentice Hall, 2001.

[27] IEEE IEEE 829. Website. Disponível em https://artedotestedesoftware.wordpress.com/tag/ieee-829. Último acesso em 02/10/2016.

[28] IEEE. *IEEE Standard Glossary of Software Engineering Terminology*. *Standard* 610.12, IEEE Press, 1990.

[29] Lu, J.; Yang, Z.; Qian, J. "*Implementation of Continuous Integration and Automated Testing in Software Development of Smart Grid Scheduling Support System*". *In International*

Conference on Power System Technology (POWERCON 2014). Chengdu, China, outubro 2014.

[30] Brandtner, M.; Giger, E.; Gall, H. "*SQA-Mashup: A mashup framework for continuous integration*". *In Journal Information and Software Technology*. Outubro 2014.

[31] Fowler, M. "*Practices Of Continuous Integration*". Website. Disponível em http://martinfowler.com/articles/continuousIntegration.html#PracticesOfContinuousIntegration. Último acesso em 28/10/2015.

[32] Extreme Programming. *Extreme Programming: A gentle Introduction*. Website. Disponível em http://www.extremeprogramming.org/. Último acesso em 12/12/2015.

[33] Jenkins User Documentation. Website. Disponível em https://jenkins.io/doc/. Último acesso em 02/11/2015.

[34] Sikuli Script. Website. Disponível em http://www.sikuli.org/. Último acesso em 12/12/2015.

[35] Yüksel, H.; Tüzün, E.; Gelirli, E.; Biyikli, E.; Baykal, B. "*Using continuous integration and automated test techniques for a robust C4ISR system*". *In 24th International Symposium on Computer and Information Sciences* (ISCIS 2009). Guzelyurt, Chipre, setembro 2009.

[36] JetBrains. TeamCity. Website. Disponível em http://www.jetbrains.com/teamcity/. Último acesso em 12/12/2015.

[37] JUnit. Website. Disponível em http://junit.org/. Último acesso em 12/12/2015.

[38] Shahin, M.; Babar, M. and Zhu, L. Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. In *IEEE Access*, vol. 5, pp. 3909-3943, 2017.

[39] Hilton M.; Tunnell T.; Huang K.; Marinov D. and Dig D., "Usage, costs, and benefits of continuous integration in open-source projects," 2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), Singapore, 2016, pp. 426-437.

[40] GitHub. Website. Disponivel em https://github.com/. Último acesso em 06/07/2017.

[41] Grunt *The JavaScript Task Runner*. Website. Disponível em http://gruntjs.com/. Último acesso em 02/11/2015.

[42] Cruise Control. Website. Disponível em http://cruisecontrol.sourceforge.net/. Último acesso em 02/11/2015.

[43] Continuum. Website. Disponível em http://continuum.apache.org/. Último acesso em 02/11/2015.

[44] Scons. Website. Disponível em http://scons.org/. Último acesso em 12/12/2015.

[45] Xiang, Y.; Jiang, S. “*Research of Weapon Equipment Integrated Test Platform*”. *In Second International Conference on Instrumentation, Measurement, Computer, Communication and Control* (IMCCC 2012). Harbin, China, dezembro 2012.

[46] Holmes, A.; Kellogg, M. “*Automating Functional Tests Using Selenium*”. *In Proceedings of AGILE* 2006 *Conference* (AGILE'06). Minneapolis, Estados Unidos, julho 2006.

[47] Selenium. Website. Disponível em http://www.seleniumhq.org/. Último acesso em 12/12/2015.

[48] Brajnik, G.; Baruzzo, A.; Fabbro, S., "*Model-Based Continuous Integration Testing of Responsiveness of Web Applications*". *In Software Testing, Verification and Validation* (ICST), 2015 IEEE 8th International Conference on, vol., no, pp.1-2, 13-17 April 2015

[49] Elbaum S., Malishevsky A.G., and Rothermel G. Test case prioritization: A family of empirical studies. IEEE Transactions on Software Engineering, 28(2), Feb. 2002.

[50] Amrit C.; Meijberg Y., "Effectiveness of Test Driven Development and Continuous Integration – A Case Study," in IT Professional, vol.PP, no.99, pp.1-1, 2017.

[51] Angelov S., Grefen P. and Greefhorst D., "*A classification of software reference architectures: Analyzing their success and effectiveness*" 2009 Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture, Cambridge, 2009, pp. 141-150.

[52] Nakagawa E. Y., Guessi M., Maldonado J. C., Feitosa D. and Oquendo F., "*Consolidating a Process for the Design, Representation, and Evaluation of Reference Architectures*" 2014 IEEE/IFIP Conference on Software Architecture, Sydney, NSW, 2014, pp. 143-152.

[53] Nakagawa E.Y., Oquendo F. and Becker M., "*RAModel: A Reference Model for Reference Architectures*" 2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture, Helsinki, 2012, pp. 297-301.

[54] Riley C. Two Approaches to Test Automation Architectures. Disponível em https://saucelabs.com/blog/two-approaches-to-test-automation-architectures. Último acesso em 07/06/2018.

[55] Elberzhager F. Architecture-Centric Integration Testing. Disponível em https://blog.iese.fraunhofer.de/architecture-centric-integration-testing/. Último acesso em 07/06/2018.

[56] NodeJS. Website. Disponível em https://nodejs.org/en/. Último acesso em 26/12/2015.

[57] SPA Javascript. Website. Disponível em https://imasters.com.br/desenvolvimento/single-page-applications-e-outras-maravilhas-da-web-moderna/. Último acesso em 22/06/2017

[58] Jasmine. Website. Disponível em https://jasmine.github.io/. Último acesso em 26/12/2015.

[59] Mocha. Website. Disponível em https://mochajs.org/. Último acesso em 25/06/2017.

[60] Jest. Website. Disponível em https://facebook.github.io/jest/. Último acesso em 25/06/2017.

[61] Chai. Website. Disponível em http://chaijs.com/. Último acesso em 25/06/2017.

[62] Unexpected. Website. Disponível em http://unexpected.js.org/. Último acesso em 25/06/2017.

[63] Sinon. Website. Disponível em http://sinonjs.org/. Último acesso em 25/06/2017.

[64] Enzyme. Website. Disponível em http://airbnb.io/enzyme/index.html. Último acesso em 25/06/2017.

[65] Testdouble. Website. Disponível em https://github.com/testdouble/testdouble.js. Último acesso em 25/06/2017.

[66] Ava. Website. Disponível em https://github.com/avajs/ava. Último acesso em 25/06/2017.

[67] Rabelo E. Ecossistema de testes em Javascript. Website. Disponível em https://medium.com/@oieduardorabelo/o-ecossistema-de-testes-em-javascript-em-2017-74a78625aa15. Último acesso em 25/06/2017.

[68] Guia de Boas Práticas. Website. Disponível em https://github.com/nilo-giannecchini/GuiaBoasPraticas. Último acesso em 11/06/18.

[69] Galeote, S. Tipos de testes de software. Website. Disponível em http://www.galeote.com.br/blog/2011/06/tipos-de-testes-de-software. Último acesso em 20/09/2015.

[70] Rios, E; Moreira, T. Teste de Software. Rio de Janeiro, Alta Books, 2003.

[71] Pfleeger S.L. Engenharia de Software: Teoria e Prática. São Paulo: Prentice Hall, 2ª edição, 2004. Capítulo 1.

[72] BlackDuck. Website. Disponível em https://www.openhub.net/p/grunt-js/similar. Último acesso em 12/11/2015.

[73] Livro "Continuous Integration: Improving Software Quality and Reducing Risk". http://www.amazon.com/Continuous-Integration-Improving-Software-Reducing/dp/0321336380

[74] Livro "Refactoring: Improving the Design of Existing Code". http://www.amazon.com/Refactoring-Improving-Design-Existing-Code/dp/0201485672

[75] Fowler M. Padrão de Refatoração "Extract Method". http://refactoring.com/catalog/extractMethod.html.

[76] Fowler M. Padrão de Refatoração "Extract Class". http://www.refactoring.com/catalog/extractClass.html.

[77] Korel, B. e Al-Yami, A. (1998), Automated Regression Test Generation, Department of Computer Science Illinois Institute of Technology e College of Technology at Dammam, Chicago, http://portal.acm.org/citation.cfm?id=271771.271803

[78] Binder R. V. “Testing Objet-Oriented Systems: models, patterns and tools". Addison Wesley, 2000.

[79] Do H., Mirarab S., Tahvildari L., and Rothermel G. The effects of time constraints on test case prioritization: A series of empirical studies. IEEE Transactions on Software Engineering, 36(5), Sept/Oct 2010.

[80] Fowler M. Test Double. Disponível em https://martinfowler.com/bliki/TestDouble.html. Último acesso em 02/10/2016.