UNIVERSIDADE FEDERAL DE GOIÁS
INSTITUTO DE INFORMÁTICA

BRUNO NOGUEIRA DE OLIVEIRA

Construindo um ambiente de Entrega Contínua

Implementando Entrega Contínua desde a Concepção de um Projeto

Goiânia
2016

UNIVERSIDADE FEDERAL DE GOIÁS

INSTITUTO DE INFORMÁTICA

AUTORIZAÇÃO PARA PUBLICAÇÃO DE TRABALHO DE CONCLUSÃO DE CURSO EM FORMATO ELETRÔNICO

Na qualidade de titular dos direitos de autor, **AUTORIZO** o Instituto de Informática da Universidade Federal de Goiás – UFG a reproduzir, inclusive em outro formato ou mídia e através de armazenamento permanente ou temporário, bem como a publicar na rede mundial de computadores (*Internet*) e na biblioteca virtual da UFG, entendendo-se os termos "reproduzir" e "publicar" conforme definições dos incisos VI e I, respectivamente, do artigo 5º da Lei nº 9610/98 de 10/02/1998, a obra abaixo especificada, sem que me seja devido pagamento a título de direitos autorais, desde que a reprodução e/ou publicação tenham a finalidade exclusiva de uso por quem a consulta, e a título de divulgação da produção acadêmica gerada pela Universidade, a partir desta data.

Título: Construindo um ambiente de Entrega Contínua – Implementando Entrega Contínua desde a Concepção de um Projeto

Autor(a): Bruno Nogueira de Oliveira

Goiânia, 15 de Julho de 2016.

Bruno Nogueira de Oliveira – Autor

Me. Otávio Calaça Xavier – Orientador

BRUNO NOGUEIRA DE OLIVEIRA

Construindo um ambiente de Entrega Contínua

Implementando Entrega Contínua desde a Concepção de um Projeto

Trabalho de Conclusão apresentado à Coordenação do Curso de Bacharelado em Sistemas de Informação do Instituto de Informática da Universidade Federal de Goiás, como requisito parcial para obtenção do título de Bacharel em Bacharelado em Sistemas de Informação.

Área de concentração: Métodos Ágeis.

Orientador: Prof. Me. Otávio Calaça Xavier

Goiânia
2016

BRUNO NOGUEIRA DE OLIVEIRA

Construindo um ambiente de Entrega Contínua

Implementando Entrega Contínua desde a Concepção de um Projeto

Trabalho de Conclusão apresentado à Coordenação do Curso de Bacharelado em Sistemas de Informação do Instituto de Informática da Universidade Federal de Goiás como requisito parcial para obtenção do título de Bacharel em Bacharelado em Sistemas de Informação, aprovada em 15 de Julho de 2016, pela Banca Examinadora constituída pelos professores:

Prof. Me. Otávio Calaça Xavier
Instituto de Informática – UFG
Presidente da Banca

Prof. Me. Walison Cavalcanti Moreira
Instituto de Informática – UFG

Prof. Me. Francisco Calaça Xavier
Faculdade de Tecnologia – SENAC-GO

Bruno Nogueira de Oliveira

Graduando em Sistemas de Informação pela Universidade Federal de Goiás e desenvolvedor de Sistemas Web Java. Durante a graduação foi monitor da disciplina de Programação Orientada à Objetos no departamento de Informática. Propós e apresentou trabalhos no Festival Latinoamericano de Instalação de Software Livre em 2015, com o tema "Utilizando o KeePass como gerenciador de senhas"e no Fórum Goiâno de Software Livre em 2015, com o tema "Olha como é fácil começar com Integração Contínua".

Dedico esse trabalho a meu avô Domingos Lustosa Nogueira e a meu amigo Maykon Menezes Carneiro que partiram dessa vida em março de 2016.

Agradecimentos

Agradeço aos meus pais, Aldemiro Araujo de Oliveira e Eliana Nogueira de Oliveira. Tudo foi graças a eles. Ao meu irmão Ademir Nogueira de Oliveira, meu primeiro grande amigo. À minha namorada Dayanne Vieira Conceição pelo carinho e apoio. Agradeço aos amigos, em especial ao Daniel Melo que foi parte ativa nesse trabalho, ajudando com o desenvolvimento do sistema proposto, à Ana Letícia Herculano pela amizade indescritível, simpatía e fidelidade, à Jéssica Millene por todas as conversas divertidas e conselhos; aos amigos da DaRua Squad (Nathan Modesto de Jesus, Blenys Santiago, Wallynson Martins, Jeziel Arruda Marciel, Giovanni Alvarenga) por toda camaradagem e apoio que sempre me prestaram. Ao Prof^{o} Mestre Otávio Calaça Xávier por ter aceitado me orientar e ter prestado apoio incondicional nesse trabalho. Agradeço também à Oobj Tecnologia da Informação, em especial algumas pessoas: Danilo Guimarães Justino Lemes, meu amigo de longa data que acreditou no meu potencial e me indicou para trabalhar; a Gustavo Santana Leite que sempre foi um líder prestativo e sempre me ajudou a evoluir profissionalmente, ao Prof^{o} Mestre Otávio Calaça Xavier, cujo trabalho inspirou essa monografia e à todos os amigos que fiz nessa empresa que de uma maneira direta ou indireta contribuíram para esse momento, em especial ao Rhuan Karlus Silva que ajudou diretamente nesse trabalho. Por fim gostaria de agradecer a todos aqueles que fazem parte da minha vida que não pude citar aqui. Se o fizesse, esse trabalho teria mais agradecimentos do que conteúdo.

Resumo

de Oliveira, Bruno Nogueira. **Construindo um ambiente de Entrega Contínua**. Goiânia, 2016. 91p. Relatório de Graduação. Instituto de Informática, Universidade Federal de Goiás.

É possível iniciar o desenvolvimento de um Software entregando-o continuamente desde a primeira submissão de código fonte? Muitos softwares ainda são desenvolvidos e mantidos segundo os modelos tradicionais da Engenharia de Software. Despende-se muito esforço em documentação e há isolamento das fases de entendimento da solução, desenvolvimento, testes e entrega. Assim, o cliente recebe o Software somente ao final do processo de desenvolvimento. Isso pode gerar alguns problemas graves, como: softwares que, quando implantados, não atendem as espectativas de funcionalidade esperadas pelo cliente; softwares que não foram desenvolvidos pensando na infraestrutura do cliente, causando problemas de escalabilidade e manutenção. Ou seja, o cliente espera muito tempo para poder usar o Software e quando este é implantado, tem muitos problemas e não atende as necessidades do usuário. Entregando continuamente desde do início do projeto é possível disponibilizar constantemente o software para o cliente, em pequenos incrementos, colhendo *feedbacks*, entendendo as espectativas e necessidades do cliente, seu ambiente de operação e seu negócio. Assim o cliente receberá rapidamente o valor agregado pela operação do Software, deixando-o mais satisfeito. O software é desenvolvido testado e homologado, sendo fácil escalá-lo e mantê-lo. Os processos de provisionamento de infraestrutura, instalação e atualização ficam simplificados pois são automatizados e simples de serem executados. A entrega de Software funcional, segundo os princípios ágeis, será efetivamente atendida.

Palavras–chave

Entrega Contínua, Métodos Ágeis, desenvolvimento de sistemas, qualidade

Abstract

de Oliveira, Bruno Nogueira. **<Work title>**. Goiânia, 2016. 91p. Relatório de Graduação. Instituto de Informática, Universidade Federal de Goiás.

Can you start the development of a software delivering it continuously from the first submission of source code? Many software is still developed and maintained according to the traditional models of software engineering. So much effort is spent on documentation and there is isolation of the understanding solution phases, development, testing and delivery. Thus, the client receives only the end of the software development process. This can lead to some serious problems, such as software that when deployed, do not meet the expectations of functionality expected by the customer; software that have not been developed thinking of the customer's infrastructure, causing scalability and maintenance problems. That is, the client waits too long to be able to use the software and when it is deployed, has many problems and does not meet user needs. Handing continuously since the beginning of the project can constantly provide the software for the customer, in small increments, harvesting feedback, understanding the expectations and customer requirements, its operating environment and your business. So the customer quickly receive the added value for the operation of the software, making it more satisfied. The software is developed tested and approved, and easy to scale it and keep it. Infrastructure provisioning processes, installation and updates are simplified because they are automated and simple to run. The delivery of functional software, according to the agile principles, will be effectively met.

Keywords

Continuous Delivery, Agile methods, System development, quality

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Códigos de Programas

CAPÍTULO 1

Introdução

Ao redor do mundo, vários softwares são desenvolvidos e mantidos. Várias abordagens são utilizadas para criar um projeto ou manter um software já em produção. Esses métodos, em geral, focam no gerenciamento de requisitos e na codificação do produto. Desde a divulgação do Manifesto Ágil [14], o foco passou a ser a adaptação à mudança e o atendimento rápido à necessidade do cliente.

Assim, métodos como Entrega Contínua surgem e passam a ser adotados no mercado. A Entrega Contínua é uma prática de desenvolvimento de software onde é possível colocar o software em produção a qualquer momento[8], através de um processo automático, confiável, repetível e testável. Neles, cada modificação feita nos artefatos do projeto passam por um processo de versionamento, construção, testes, análises de conformidade e qualidade de código, disponibilização e implantação em ambientes automaticamente provisionados. Essas etapas contam com pouca intervenção humana. O foco do processo é a automatização de todas as tarefas que se repetem, de forma que o produto seja entregue rapidamente para o cliente.

A abordagem da Entrega Contínua, em geral, é aplicada para projetos que já estão em fase de manutenção. Ou seja, o método passa a ser aplicado em Software que já está em produção. Nos artigos utilizados como referência nesse trabalho, é comum o seguinte cenário: a empresa mantém o software sob um método ágil, porém grande parte do processo de implantação e/ou atualização é feita manualmente. Mesmo sob uma perspectiva ágil, os ciclos de desenvolvimento levam várias semanas e o processo de atualização é complicado e custoso. Um bom exemplo é o cenário da Empresa Rally Software, citada no artigo "*Continuous Delivery? Easy! Just Change Everything (well, maybe it is not that easy)* [18]". O mesmo cenário é apresentado no artigo de Lianping Chen, "*Towards Architecting for Continuous Delivery* [6]". Nos relatos dos autores desses artigos, o impacto dessa mudança de abordagem para Integração e Entrega Contínua é positivo. O qualidade do produto melhora. A equipe de desenvolvimento trabalha menos reativamente e mais proativamente. Os problemas, uma vez solucionados, não voltam a ocorrer. E o principal: o cliente tem suas necessidades atendidas mais rápido e com maior conformidade com suas expectativas.

O presente trabalho levanta as seguintes questões: é possível pensar em Entrega Contínua antes mesmo da primeira linha de código ser escrita? Qual as vantagens e desvantagens de iniciar um projeto de Software pensando em como ele será entregue no ambiente do cliente? É viável o esforço de criar e manter o processo de Entrega Contínua antes mesmo de começar o desenvolvimento do projeto? Em qual grau de maturidade deve ser construído um ambiente de Entrega Contínua para um projeto que está em sua fase de concepção?

Construir softwares aptos a serem entregues continuamente modifica drasticamente a forma como os software foram construídos durante anos. Adaptar projetos já existentes aos modelos que a Entrega Contínua propõe é um trabalho caro e arriscado. O custo e o impacto de tal mudança afeta profundamente a cultura e a dinâmica das organizações, conforme visto nas referência anteriores. Assim, nesse trabalho, é exposto a perspectiva de criar e configurar o ambiente para Entrega antes de iniciar a escrita do código que irá implementar o Software. O objetivo é criar uma estrutura que pode ser copiada e aplicada facilmente para projetos que vão ser iniciados, de forma que o trabalho de estabelecer um ambiente para Entrega Contínua não seja tão complicado. Esse trabalho também demonstra que essa estrutura é aplicável e funcional, permitindo facilmente sua evolução e extensão.

As questões acima serão respondidas através de um estudo teórico sobre os principais conceitos à respeito da Entrega Contínua. No Capítulo 2 é explicado em detalhes o que é a Entrega Contínua e os motivos para aplicá-la em um projeto de Software. No Capítulo 3 é apresentada a Gerência de Configuração e sua importância fundamental para estabelecer um processo de Entrega Contínua. O Capítulo 4 fala sobre a Integração Contínua e quais características possui um ambiente de Integração Contínua. Integração Contínua é o processo de integrar continuamente o trabalho de toda equipe de desenvolvimento de um projeto. O Capítulo 5 apresenta a Pipeline de Entrega. Esse é o assunto central da Entrega Contínua. Esse Capítulo trata das características de uma Pipeline de Entrega e quais as características para estabelecer uma Pipeline de Entrega em um projeto com Entrega Contínua. O Capítulo 6 fala sobre a importância dos testes automatizados na Entrega Contínua, quais os níveis de testes e a prática de escrever testes antes de implementação. A parte teórica finaliza no Capítulo 7, onde é apresentado o conceito da Implantação Contínua em produção, quais são suas características e quais desafíos ela apresenta frente à Entrega Contínua de Software. Também é apresentado um experimento prático no Capítulo 8, onde um Software conceitual foi desenvolvido sob o método de desenvolvimento de Entrega Contínua.

1.1 Trabalhos correlatos

Esse trabalho é baseado, primariamente, no livro de David Farley e Jaz Humble, *Continuous Delivery, Reliable Software Releases Through Build, Test and Deployment Automation*[13]. Através desse livro são relacionados trabalhos de outros autores, como P. Durval, S. Matyas e A. Glover, *Continuous Integration, Improving Software Quality and reducing risk*[19], Steve Neely e Steve Stolt, *Continuous Delivery? Easy! Just Change Everything (well, maybe it is not that easy)*[18] e Lianping Chen, *Towards Architecting for Continuous Delivery* [6], buscando encontrar as caracteristicas elementares para construir projetos rapidamente entregáveis. Usando esses trabalhos como base, é apresentada uma proposta onde um ambiente básico de Entrega Contínua é construído antes do início do desenvolvimento do projeto e após sua construção o desenvolvimento é iniciado. Os resultados demonstram quais as principais vantagens e desvantagens no desenvolvimento de uma aplicação utilizando os princípios da Entrega Contínua desde sua concepção.

No trabalho de Linping Chen [6] são descritas as principais caracteristicas da Entrega Contínua, o porque a aplicação deve ser arquitetada para Entrega Contínua e quais as implicações ao se arquitetar para Entrega Contínua. Essas caracteristicas buscam nortear quais são as prioridades de uma equipe que deseja entregar continuamente seu projeto.

O trabalho de Steve Neely e Steve Stolt descreve a experiência de migrar de um processo ágil utilizando o Scrum para a Entrega Contínua na empresa Rally Software[18]. No trabalho é relatado quais foram as ações tomadas nesse cenário e qual o impacto disso para o negócio. Nesse trabalho é mostrado principalmente o impacto no negócio da empresa, não só no desenvolvimento de Software. Ao implantar a Entrega Contínua o negócio como um todo é afetado positivamente, desde a área de negócios, que passa a ter mais confiança no produto, melhorando estratégias de vendas, marketing e tomada de decisões até a operação. A operação é diretamente impactada pois a incidência de problemas tende a diminuir e os problemas são resolvidos e entregues mais rápido. O trabalho também apresenta algumas práticas possíveis para quem deseja migrar seus projetos para Entrega Contínua.

Outro trabalho relacionado é o "Modelo de Maturidade da Entrega Contínua"de Andreas Rehn, Tobias Palmborg e Patrik Boström[2]. Nesse trabalho os autores definem cinco níveis de maturidade (base, iniciante, intermediário, avançado e expert) para cinco categorias essenciais de uma organização (Cultura organização, design arquitetura, construção entrega, teste verificação, informação reportes). Eles fazem um mapeamento entre as principais caracteristicas que devem ser implementadas para se alcançar um nível de maturidade em uma categoria essencial da organização. Através desse mapeamento é possível identificar quais os próximos passos a serem adotados para melhorar a prática da

Entrega Contínua.

David Farley e Jez Humble também apresentam um modelo de maturidade no seu livro[13]. No modelo de maturidade de Farley e Humble também existem cinco níveis de maturidade (Regressivo, Repetível, Consistente, Gerenciado Quantitativamente e Em Otimização) em relação a cinco práticas da Entrega Contínua (Ambientes *Deployment*, Gerenciamento de releases *Compliance*, Testes, Gerenciamento de Dados e Gerenciamento de Configurações). O modelo de Farley e Humble é mais voltado às caracteristicas essenciais da Entrega Contínua enquanto o modelo de Rehn, Palmborg e Patrik é voltado para a organização como um todo.

Esses trabalhos mostram quais são os impactos que ocorrem no software ao se aplicar a Entrega Contínua e como esses softwares devem ser construídos para viabilizar a adoção dessa prática.

CAPÍTULO 2

Entrega Contínua

Desenvolver Softwares com qualidade e que entregue valor ao cliente final é o grande desafio que profissionais que atuam com Engenharia de Software enfrentam diariamente. Vários projetos de Software são desenvolvidos e mantidos ao redor do mundo. Processos de desenvolvimento são aperfeiçoados. Novos processos surgem. Todos eles com foco na entrega de um produto que atenda ou supere as expectativas do cliente final. Com o advento dos princípios divulgados pelo Manifesto Ágil[14], que propõe que a satisfação do cliente é o foco principal do desenvolvimento e o único parâmetro de sucesso importante, conceitos como Entrega Contínua ganharam espaço no mercado.

O que é Entrega Contínua? Segundo Martin Folwer[8]

> É a disciplina de desenvolvimento de Software onde você constrói de forma que o Software possa ser colocado em produção a qualquer momento.

Steve Neely e Steve Stolt[18] dizem que Entrega Contínua é:

> [...]A habilidade de entregar o Software sempre que desejarmos. Isso pode ser semanalmente ou diariamente[...]; Isso poderia dizer que cada *check-in*[1] vai direto para produção. A frequência não é o nosso fator de decisão. É a habilidade de fazer isso que nos interessa.

Outra citação interessante é de Akond Ashfaque Ur Rahman, Eric Helms, Laurie Williams e Chris Parnin que dizem que[1]:

> Entrega Contínua é o processo de engenharia de Software que foca na entrega rápida das mudanças do software para o usuário final, e nesse processo de engenharia de Software incremental as mudanças são automaticamente testadas e frequentemente implantadas nos ambientes de produção;

Nota-se que Entrega Contínua é uma prática de desenvolvimento de Software voltada à entrega de novas versões do Software de maneira rápida e eficiente. O foco da

[1] Por *check-in*, entende-se o processo de submeter uma modificação aos artefatos do projeto no Sistema de Controle de versão, abordado no Capítulo 3.

Entrega Contínua é, sobretudo, na automatização de todo o processo de versionamento, construção e implantação. A qualquer momento deve ser possível entregar a versão mais recente do Software em qualquer ambiente onde ele seja executado. O processo de entrega, desde a verificação de conformidade dos artefatos desenvolvidos, até a instalação no ambiente de produção deve ocorrer de maneira eficiente, repetível e confiável.

Martin Fowler diz que estamos realizando Entrega Contínua quando[8]:

- o software pode ser implantado em qualquer fase do ciclo de vida;
- a equipe prioriza a entrega do software ante o desenvolvimento de novas funcionalidades;
- qualquer pessoa tem um *feedback* rápido e automatizado sobre as modificações que ocorreram no Software
- podemos implantar o software, em qualquer versão, apenas através do "clicar de um botão"em qualquer ambiente que seja demandado.

Ainda, segundo Martin Fowler, para alcançar a Entrega Contínua é preciso integrar o trabalho de toda equipe continuamente. Assim, artefatos executáveis são construídos e testados a fim de detectar problemas antecipadamente. Em seguida, esses executáveis são colocados em um ambiente semelhante ao ambiente de produção (ambiente de homologação) para verificar como o software vai se comportar nesse ambiente [8]. Isso pode ser feito através de uma *pipeline* de entrega, discutida no Capítulo 5. Uma pipeline de entrega são divisões feitas no ciclo de construção e entrega, onde as primeiras etapas executam mais rápido dando *feedbacks* mais simples e rápidos, enquanto as etapas posteriores são mais lentas, porém entregam *feedbacks* mais completos.[12]

O *feedback* é importante para a Entrega Contínua. É através dele que podemos avaliar a qualidade do Software desenvolvido. Cada mudança deve disparar um *feedback*. Ele deve ser entregue o mais rápido possível para todos os interessados. A equipe de desenvolvimento deve agir sobre ele, assim que o recebe[13]. Por exemplo, se uma construção ou entrega falha, o time deve agir e resolver o problema rapidamente, para ser possível continuar entregando o software funcional.

Dentre os benefícios de se entregar continuamente o Software, Martin Fowler destaca[8]:

- **Redução dos riscos:** os riscos são mitigados pois serão entregues pequenas mudanças no Software. Isso torna simples o processo de reverter modificações que causaram problemas. É diferente de implementar várias mudanças e tentar entregá-las todas ao mesmo tempo. Se um erro ocorre, é difícil encontrar qual mudança causou o problema. Se as mudanças são implantadas em pequenas porções, uma modificação defeituosa pode ser rapidamente encontrada e corrigida.

- **Confiança no resultado final:** o progresso do trabalho não será feito primariamente sobre registros das atividades do desenvolvedor. Não basta apenas dizer que o trabalho está pronto. É preciso que a funcionalidade esteja disponível para ser colocada em produção. Assim, uma atividade é dada como concluída quando foi devidamente testada e aprovada.
- ***Feedback* do usuário final:** com o foco no processo de entrega do software no ambiente de produção, é possível que os usuários interajam com o sistema antes da sua entrega final. Assim ele consegue visualizar a evolução do sistema, testar as novas funcionalidades e relatar expectativas ou inconformidades antes que a versão definitiva seja entregue. A Entrega Contínua possibilita isso através da facilidade em provisionar ambientes semelhantes ao de produção para testes. São os ambientes de homologação ou ambientes de qualidade (AQ);

Conhecendo esses princípios basilares sobre Entrega Contínua, será apresentado nesse trabalho as motivações principais para se pensar em entregar o software desde a escrita da primeira linha de código. Será explicitado quais são os conceitos principais da Entrega Contínua e como é possível gerar melhores resultados finais através da aplicação da Entrega Contínua desde o início do projeto.

CAPÍTULO 3

Gerência de Configuração

O desenvolvimento de um projeto de Software acarreta na criação de vários artefatos. Termo de abertura de projeto, cronogramas de desenvolvimento, documentação de requisitos, arquitetura, fluxogramas, diagramas UML, código-fonte, *scripts* de automação diversos, *scripts* de banco de dados, configuração de ambiente, documentação de testes entre tantos outros artefatos. Todos esses artefatos são importantes para a equipe de desenvolvimento e os *stackholders*[1]. Eles são a garantia de que o projeto está sendo executado, o entendimento do projeto está claro, acordado e documentado entre o cliente final, os gestores e a equipe de desenvolvimento. Todavia esses artefatos são mutáveis, principalmente na fase inicial do desenvolvimento.

Requisitos sofrem modificações e evoluem. Novos requisitos aparecem. A arquitetura pode ser modificada para atender novos requisitos e consequentemente o código fonte sofre alterações. Artefatos de testes e suas documentações também são afetados. Essas modificações constantes e muitas vezes imprevisíveis trazem uma série de problemas: em qual versão do Software um requisito foi atendido? Em qual versão do Software surgiu determinado requisito? Quando determinada parte do código sofreu modificação? Quem foi o responsável pela modificação? Se a modificação causou um erro, como reverter para uma versão estável? Qual é a última versão estável? Para responder essas e outras questões que são levantadas durante o processo de desenvolvimento do Software é que existe a Gerência de Configuração e o Controle de Versão.[13]

A Gerência de Configuração, segundo Steve McConnel é [17]:

> a prática de identificar artefatos do projeto e tratar das alterações sistematicamente, de modo que um sistema possa manter sua integridade com o passar do tempo.[...] Ele inclui técnicas para avaliar as alterações propostas, controlar as alterações e manter cópias do sistema em vários momentos no tempo.

Gerencia de Configuração é um assunto bem amplo. Ian Sumerville diz que a Gerência de Configuração possui quatro atividades fundamentais [24]:

[1] *Stackholders* são quaisquer pessoas ou organizações que tenha interesse, são afetadas ou investem no projeto. São os principais interessados no sucesso do projeto

- **Gerenciamento de mudanças**: sistema que mantêm e controla as diversas solicitações de modificação do software. Nesse sistema os custos e impactos de modificação em um Software são mantidos e gerenciados de forma que se consiga, com o passar do tempo, saber quando uma mudança foi solicitda, por quem, quando foi feita e em qual versão foi entregue.
- **Gerenciamento de versões**: sistema que acompanha e mantém os componentes do software e suas várias versões. Envolve manter os artefatos versionados, de forma que seja possível recuperar uma determinada versão de um componente. Também garante que o trabalho de um desenvolvedor não interfere no trabalho de outro.
- **Construção de Sistema**: Sistema que constroi uma determinada versão do software levando em consideração os dados, bibliotecas e dependências de uma determinada versão. É responsabilidade da construção interligar os diversos componentes em suas respectivas versões a fim de criar um software funcional.
- **Gerenciamento de entrega**: gerenciamento que visa preparar o software para entregas externas e acompanhar os ambientes onde o software é executado, bem como as versões presentes nesses ambientes.

Cada uma das atividades do Gerenciamento de Configuração são importantes para a Entrega Contínua, pois cada uma delas influencia em um aspecto relacionado a como o software será modificado, versionado, construído e entregue. Portanto é necessário entender cada uma dessas atividades, o que ela influência no processo de desenvolvimento e como ela se relaciona com a entrega de um Software para o cliente. Nas sessões à seguir, cada uma dessas atividades é explorada detalhadamaente.

3.1 Gerenciamento de Versão

O Gerenciamento de Versão ou Controle de Versão é uma das atividades fundamentais na Entrega Contínua. Sem controlar a versão dos artefatos do projeto não é possível executar nenhuma das demais atividades de uma Pipeline de Entrega, descrita no Capítulo 5. Com o Controle de Versão é possível submeter e acompanhar o desenvolvimento do projeto. Através do Controle de Versão, segundo Jez Humble e David Farley, é possível responder ”sim” a algumas perguntas chaves [13]:

- é possível reproduzir qualquer um dos ambientes, incluíndo versão do Sistema Operacional, configurações de rede, *patches*, aplicações implantadas e suas configurações?
- é possível executar facilmente uma mudança incremental em qualquer artefato individual e implantar isso em qualquer ou todos os ambientes?

- é possível identificar facilmente cada mudança que ocorreu em um ambiente particular e verificar no histórico exatamente qual a mudança, quem fez e quando ocorreu?
- é possível satisfazer todas as regras de conformidade que o código está sujeito?
- é possível que qualquer membro do time obtenha facilmente as informações que ele precisa para fazer as mudanças que ele deseja?

Para atender as questões acima é necessário uma boa estratégia de versionamento dos artefatos do projeto. É sobre essa estratégia que o processo de Entrega Contínua vai executar. São as mudanças no Controle de Versão que vão disparar os gatilhos para Entrega Contínua. É no sistema de gerenciamento do Controle de Versão que será possível identificar as mudanças que ocorrem no projeto, determinar as versões e consequentemente chegar às respostas dos questionamentos anteriores. Para tal é preciso tomar duas decisões:

1. como as modificações serão submetidas no sistema de controle de versão?
2. qual ferramenta de controle de versão deverá ser utilizada?

Para submeter mudanças nos artefatos do projeto de forma que o desenvolvimento do sistema seja coerente e sustentável é preciso definir um fluxo de trabalho. O fluxo de trabalho é explicado na sessão 3.2. Nessa sessão, o importante é saber que tudo que for referente ao projeto precisa ser versionado. Algumas organizações mais maduras no processo de Entrega Contínua versionam as máquinas virtuais com o Sistema Operacional e configurações padrões. Isso torna mais simples o processo de recriar um ambiente. Tudo que for importante para reproduzir uma versão específica do ambiente de entrega contínua deve ser devidamente versionado[13].

Deve ser possível recuperar os artefatos do projeto em suas últimas versões disponíveis, bem como reverter esses artefatos para versões anteriores. Isso é feito através do *Controle de Versão*. Essa técnica é importante por dois motívos: os artefatos que sofrem modificações podem inserir erros ou inconformidades no software. Quando um artefato é modificado e sua versão anterior não é facilmente recuperada, erros graves podem surgir e sua correção pode ser difícil. É preciso ter a capacidade de retornar os artefatos a versões anteriores estáveis. Outra motivação é que os membros da equipe precisam ter seus trabalhos integrados e sem conflitos. Quando não existe um repositório que centralize as versões mais recentes dos artefatos, muitos conflitos podem surgir. Os desenvolvedores fazem retrabalhos, ou seja, escrevem partes do sistema que outro desenvolvedor já havia escrito. Concluí-se que Controle de Versão é um gerenciamento de *codeline* e *baselines* [24]. *Codeline* é entendida como o histórico de evoluções de um determinado artefato. Ou seja, cada vez que um artefato sofre uma modificação, uma nova versão do artefato é

Tabela 3.1: *Operações comuns nos sistemas de Controle de Versão*

Termo	Significado
Checkout	Cria uma cópia local do projeto à partir da última versão disponibilizada no repositório central
Commit	Submete alterações no repositório, criando uma revisão de trabalho
Update	Atualiza a cópia de trabalho local buscando as modificações que foram submetidas ao repositório central que não estão presente localmente
Push	Envia as revisões para o repositório central
Branch	Cria uma cópia do estado do projeto, chamado também de linha de desenvolvimento independente, de forma que as submissões feitas nessa linha de desenvolvimento, não influênciam na outra
Merge	Mescla o trabalho de duas *Branches* apontando os conflitos quando existirem
Revert	Reverte as modificações de uma revisão, retornando os artefatos ao seu estado anterio

incluída na *codeline*. Já uma *baseline* é um conjunto de artefatos versionados que quando colocados em conjunto gera uma versão funcional do sistema.

Através do Sistema de Controle de Versão é possível resolver alguns problemas como:

- armazenar de maneira centralizada os artefatos do projeto;
- recuperar históricos de alteração dos artefatos;
- possibilitar mesclagem do trabalho. Como fazer com que desenvolvedores diferentes façam modificações no mesmo artefato sem que o trabalho de um impacte de maneira negativa o trabalho do outro?
- desenvolver vários projetos independentes;

Em relação às ferramentas, existem no mercado várias ferramentas para controle de versão, algumas gratuítas e outras pagas. Dentre essas ferramentas destacam-se o Git, SVN, TFS, Mercurial, CVS e Perforce[25]. Cada ferramenta tem suas caracteristicas, porém todas possuem basicamente as mesmas funcionalidades. A tabela 3.1 apresenta algumas operações comuns a essas ferramentas:

Uma comparação de desempenho entre essas ferramentas mostra que o Git tem desempenho superior para grande parte das operações, conforme apresentado no trabalho de André Felipe Dias[7]. A figura 3.1 mostra o resultado desse comparativo. Por esse motivo, nesse trabalho será utilizado o Git como ferramenta de Gerenciamento de Versão.

3.2 Gerenciamento de Mudanças

É necessário que tudo que for referente ao projeto seja versionado [13]. A forma como o versionamento dos artefatos ocorrem devem auxiliar a responder a algumas

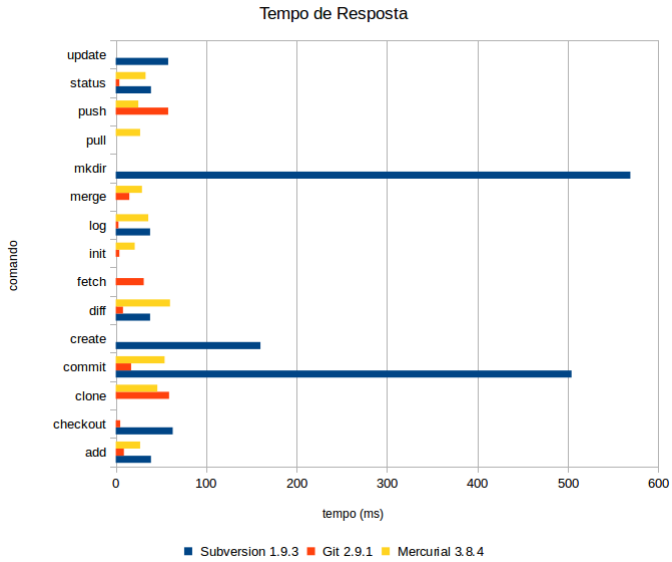

Figura 3.1: *Comparação de desempenho entre Git, SVN e Mercurial, segundo André Felipe Dias[7]*

perguntas, tais como: quais são os requisitos atendidos pela versão X do sistema? Em qual versão o requisito Y foi solicitado? Quando ocorreu a modificação no documento de arquitetura que modificou o arcabouço de codificação do projeto? Em qual versão do sistema o teste automatizado Z foi atendido? Qual a versão do Sistema Operacional consigo executar a versão A do Software? Qual versão do Banco de Dados consigo executar a versão B do sistema? Quais são as configurações de ambiente da versão C do Software?

As respostas às questões anteriores dependem de como as modificações são submetidas pela equipe de desenvolvimento. Por isso, é preciso determinar a forma como os desenvolvedores vão submeter as modificações no Sistema de Controle de Versão. A forma como os artefatos são recuperados, modificados e submetídos devem estar sob um processo claro e centralizado, para que os colaboradores contribuam com o software de maneira saudável. Por isso é necessário desenvolver um fluxo de trabalho. Um fluxo de trabalho consiste num conjunto de passos em um processo e o relacionamento nas transição entre esses passos[5]. Não existe um processo deterministico para um fluxo de trabalho. Cada projeto ou organização pode modelar o fluxo de trabalho de acordo com suas necessidades.

O fluxo de trabalho, apesar de estar fortemente relacionado ao Controle de Versão, fica sob o domínio do **Gerenciamento de mudanças**. Gerenciamento de mudanças é o processo que visa manter o acompanhamento das solicitações de mudanças feitas no Software[24]. Todo software, inevitavelmente sofre mudanças durante seu desenvolvimento e manutenção. Essas mudanças precisam serem devidamente registradas, de forma que seja possível identificar, na solicitação de mudança, quem foi seu solicitante, quem

realizou a mudança, quais artefatos foram modificados e em qual versão tal modificação foi liberada. Todas as solicitações e registros deverão ser mantidas sob o gerenciamento de mundanças. Esse sistema manterá todos as requisições de mudanças feitas, bem como as interações feitas sobre essas requisições de mudanças. O impacto e custo dessas solicitações devem ser registradas de maneira centralizada para que fique visível a todos os *stackholders* do projeto. Deve ser possível recuperar as informações necessárias sempre que preciso através dessa ferramenta. Também deve ser possível saber quais foram os artefatos impactados por essa solicitação de modificação.

Gerenciar mudanças está fortemente ligado à gerenciamento do projeto[24]. Ao gerenciar o processo de mudança no software, assuntos como gerenciamento de pessoas, de escopo do projeto, de tempo, de riscos devem ser levados em consideração. Em alguns cenário são utilizadas ferramentas de Gerenciamento de Projeto para gerenciar mudanças. Um exemplo é o Redmine[2]. Nela é possível diversos processos, fluxos de trabalho, perfis de usuário bem como integrar à ferramenta de Controle de Versão, de forma que as revisões geradas pelos desenvolvedores sejam atreladas às solicitações de mudança registradas. Porém existem no mercados outras ferramentas mais simples. Essas ferramentas são conhecidas como *Issues track system*, ou Sistemas de rastreamento de incidentes. Sistemas de centralização de repositórios baseado no Git, tais como Github[3] e Gitlab[4] oferecem esses serviços junto com o sistema de controle de versão. Essas facilidades ajudam bastante a equipe de projetos a manter centralizado todas as solicitações de melhorias ou reportes de defeitos que aparecem durante o desenvolvimento do projeto. O fluxo de trabalho deve considerar tais ferramentas e ser desenhado de forma que torne coerente o desenvolvimento do Software.

3.3 Construção de sistema

Uma vez que os artefatos do projeto estão devidamente versionados é preciso garantir dois fatores: todas as dependências necessárias para o software funcionar devem estar facilmente acessíveis e o processo de construção do software deve ser garantido.

O primeiro fator é garantido pelo Gerenciamento de Dependência. Entende-se por "dependência"a necessidade que um software possui de uma biblioteca terceira para executar suas funcionalidades[13]. Em geral os softwares dependem de componentes externos. Bibliotecas terceiras que já implementam algumas soluções comuns são reaproveitadas para que a equipe de desenvolvimento não precise reescrever implementações já

[2]Saiba mais em: http://www.redmine.org/
[3]Saiba mais em: https://github.com/
[4]Saiba mais em: https://about.gitlab.com/

feitas. Isso é chamado de **reuso**[26]. Quando um software vai ser construído, essas bibliotecas devem acompanhar a versão compilada do programa para não ocorrer problemas quando as funções disponibilizadas por essas bibliotecas forem chamadas. Essas bibliotecas também, em geral, são versionadas e sofrem atualizações. Tão importante quanto garantir que essa biblioteca terceira esteja disponível para o sistema é garantir que essa dependência esteja com a versão correta. Se a biblioteca, por exemplo, sofre uma atualização e uma função que ela disponibilizava é descontinuada e deixa de existir, é provável que o Software em desenvolvimento apresente falhas de compilação por chamar uma função que não existe na versão mais recente da biblioteca terceirizada. Por isso é preciso gerenciar corretamente as dependências do Software.

Garantida o gerenciamento das dependências externas do projeto é preciso pensar na maneira como o Software é construído. O processo de compilação é que vai garantir a construção definitiva do sistema para ser utilizado. Essa construção pode ser um processo simples, como simplesmente gerar os artefatos binários à partir do código fonte ou pode ser um processo mais complexo. Esse processo geralmente é descrito através de um *script* de construção. Sommerville descreve os passos necessários para construção de um Software sob a perspectiva da **construção de sistemas**[24]. O processo deve contar com as seguintes etapas:

1. Gerar o *script* de construção que descreverá todo o processo de construção do software, inclusive descrevendo as dependências e como gerar documentação;
2. Integrar com sistemas de gerenciamento de versão para que seja possível recuperar as versões requeridas dos componentes que a aplicação depende;
3. Compilar os arquivos de código fonte do projeto para gerar os artefatos binários capaz de serem executados pela máquina
4. Ligar os componentes binários gerados com os artefatos que ele depende, gerando um artefato executável;
5. Executar testes automatizados para verificar se o componente gerado estão em conformidade com o esperado, segundo explicado no Capítulo 6;
6. Gerar relatórios de sucesso ou falha da construção, relatório da execução dos testes e também outros relatórios que podem ser personalizados;
7. Gerar documentação automatica do software, notas de release e/ou página de ajuda do sistema.

Dessa forma é possível padronizar a maneira como o Software é construído, bem como ele recupera suas dependências, em que ordem os processos são executados e quais são os processos básicos de cada etapa. Isso é importante para a execução da Integração Contínua, explicada no Capítulo 4. Usando esse script de construção torna-se possível gerar artefatos executáveis do Software de maneira padronizada, disponibilizando esses artefatos produzidos para os demais processos da Entrega Contínua.

Algumas ferramentas auxiliam o gerenciamento das dependências bem como o processo de construção do software. Essas ferramentas são muito úteis para equipes de desenvolvimento, pois diminui a complexidade de construir softwares. Essas ferramentas automatizam esses processos, que costumam gerar graves problemas de integração em equipes que não tem um processo bem definido de como suas dependências são recuperadas e a construção do seus sistemas são executados. Exemplos de ferramentas que implementam o processo de gerenciamento de dependência e/ou o processo de construção são Ant, Maven e Grandle[5]. Existem outras disponíveis no mercado que atendem seu público dando suporte a plataformas ou linguagens de programação. Cabe à equipe definir a que melhor se encaixa no cenário de desenvolvimento proposto. Para esse trabalho será utilizado o Maven.

3.4 Gerenciamento de Entrega

Por fim é necessário conhecer como o software será entregue, bem como onde ele irá executar. Uma vez que o Software é construído e a Entrega Contínua garante que o Software é passível de executar em produção é preciso definir onde e como entregar o Software. Essa etapa é o **Gerenciamento de Entrega**.

O software desenvolvido pode ter diferentes formas de execução. Ele pode ser entregue em um ambiente centralizado, onde os clientes acessam seus recursos de maneira centralizada. O software pode ser entregue em vários ambientes diferentes, com todos os ambientes compartilhando recursos e executando os processos paralelamente. O software pode ter sido desenvolvido para ser executado no ambiente do cliente, criando várias instâncias independente de execução. Há várias maneiras diferentes de entregar um Software. Na Entrega Contínua, o processo de entrega do software em produção é importante. A Entrega Contínua é a capacidade de entregar rapidamente mudanças no software em ambiente de produção em qualquer ambiente demandado. Portanto nota-se que é importante conhecer onde o Software está executando, bem como é possível acessar esse ambiente para entregar o Software e suas modificações.

Gerenciar entrega envolve conhecimento em infra-estrutura e automação. Os cenários de entrega podem ser bem simples, como o cenário do Software produzido nesse trabalho, onde é necessário apenas fazer a entrega em um container Web em um ambiente específico. Porém essa entrega pode se tornar bastante complexa, como no caso de softwares *clusterizados* ou softwares distribuídos. Cabe à equipe de desenvolvimento,

[5]Veja a apresentação dessas ferramentas em: https://technologyconversations.com/2014/06/18/build-tools/

bem como a equipe de T.I. analisar o cenário de execução do Software e garantir as configurações e ferramentas necessárias para entregar o Software.

Esse assunto não será aprofundado nesse trabalho. A proposta desse trabalho é garantir que é possível, sim, pensar em Entrega Contínua desde o princípio do desenvolvimento do Software. Como o software está em fase de concepção e desenvolvimento, dificilmente ele será colocado em produção até que uma boa gama de requisitos sejam atendidos. O processo de entregue pode amadurecer conforme o Software é desenvolvido e o cenário de execução de produção é conhecido. O experimento proposto nesse trabalho é bastante simples, logo o conhecimento necessário para entregar em produção não será tão complexo. Esse processo será apresentado em detalhes no capítulo 8, bem como as ferramentas utilizadas. Outro motivo é que a gama de ferramentas e possibilidades de Entregar o software é muito grande, logo não compensa abordar alguns cenários ignorando todos os demais que são possíveis. Gerenciar entrega é um assunto que pode ser abordado em um trabalho futuro.

CAPÍTULO 4

Integração Contínua

Uma vez que está definido como os artefatos do projeto serão versionados e compartilhado entre os interessados, o desenvolvimento do projeto deve ser iniciado. Isso implica na escrita do código fonte. Código deve ser compartilhado com toda equipe, compilado, testado e verificado. Também deve estar de acordo com normas e padrões estabelecidos pela equipe para termos um software compreensível e manutenível. Assim o próximo passo é definir um ambiente de Integração Contínua.

Para estabelecer um ambiente de Entrega Contínua em um projeto é importante estabelecer antes a Integração Contínua. Esse processo é o responsável por integrar o trabalho de toda equipe de desenvolvimento, construíndo constantemente o Software que está sendo produzido, encontrando os erros mais óbvios e disponibilizando *feedbacks* rápidos para os interessados no projeto.

Matin Fowler descreve Integração Contínua como:

> ... um pratica de desenvolvimento de Software onde os membros da equipe integram seu trabalho frequentemente, geralmente ao final do dia - levando a múltiplas integrações por dia. Cada integração é verificada por uma construção automatizada (incluindo testes) para detectar problemas de integração o mais rapidamente possível.[11]

Já John Ferguson Smart diz:

> ..., na forma mais simples, envolve uma ferramenta que monitora as mudanças que ocorrem no Sistema de Controle de Versão. Quando uma mudança é detectada, essa ferramenta automaticamente compila e testa sua aplicação. Se alguma coisa errada ocorrer, a ferramenta notifica imediatamente o desenvolvedor para que ele corrija o problema o mais rápido possível.[23]

O conceito de Integração Contínua veio para resolver um problema que não é novo em equipes de desenvolvimento de Software: a Integração do trabalho [19]. Entende-se por Integração o ato de colocar todos os componentes desenvolvidos isoladamente de um projeto em conjunto. Assim o Software deve executar, em sua totalidade, a fução para qual foi desenvolvido. Antes da Integração Contínua, a fase de integração em um projeto de Software ocorria em um momento isolado e era bastante problemática [19]. Durante

essa fase as equipes de desenvolvimento eram alocadas para integrar os componentes de Software que foram desenvolvidos durante um ciclo de algumas semanas. Como os componentes eram desenvolvidos de maneira isolada uns dos outros, na fase de integração eram encontrados vários problemas e defeitos, muitas vezes exigindo refatorações e reversões de implementações já prontas a dias. Isso resulta em retrabalhos e incertezas quanto ao funcionamento do Software em ambiente de Produção. Com a Integração Continua a fase de integração deixa de existir como uma etapa isolada de um processo de desenvolvimento e passa a ser uma atividade transversal, sendo executada constantemente durante o desenvolvimento dos componentes do Sistema.[19]

4.1 Ciclo de vida da Integração Contínua

Paul M. Duvall com Steve Matyas e Andrew Glover descrevem as fases que são necessárias para se ter Integração Contínua[19]. Essas etapas descrevem o que cada atividade executa, bem como a finalidade dessa atividade. Essas etapas são explicadas à seguir:

4.1.1 Compilar o Codigo Fonte

Consiste em criar os arquivos binários à partir do código fonte escrito pelo desenvolvedor. Essa compilação deve ser **contínua** e **automatizada**. Por "contínua", entende-se que a execução da compilação dos códigos fontes ocorre toda vez que uma mudança for submetida ao Sistema de Controle de Versão do Software. Também deve ser "automatizada", ou seja, essa compilação deve ocorrer sem a necessidade de intervenção humana durante o processo. Automatizar uma tarefa repetitiva como compilar o código fonte trás enormes benefícios para a equipe de desenvolvimento como um todo. Ao automatizar a compilação problemas com falhas humanas que eventualmente podem ocorrer quando o processo é manual são evitados. Ao tornar essa compilação contínua torna-se possível garantir que o Software sempre vai estar apto a ser disponibilizado. Assim, é possível ter um *feedback* rápido quando algo que foi submetido impedir a compilação dos artefatos do Software.[19]

4.1.2 Integrar a Base de Dados

A maioria dos Sistemas de Informação desenvolvidos contam com um Banco de Dados. O objetivo da maioria dos Sistemas de Informação desenvolvidos é a manipulação dos dados de uma organização. A fase de integrar a base de dados consiste no processo de execução dos Scripts de criação do Banco de Dado, gerando as tabelas e suas restrições. Essa fase é importante dentro do ciclo de Integração Contínua [19]. É comum que

equipes de desenvolvimento enxerguem a equipe de Banco de Dados como uma equipe separada, portanto tratem a base de dados como parte externa ao sistema. Essa visão é modificada com a Integração Contínua, pois o Banco de Dados é parte importante do Sistema, principalmente para o cliente final que usará esses dados para tomada de decisões estratégicas. Integrar a base de dados garante que a equipe de desenvolvimento conheça a estrutura do Banco e faça com que o desenvolvimento do Sistema seja orientado a essa estrutura [19].

4.1.3 Executar testes

Não é possível dizer que existe Integração Contínua se não existirem testes automatizados que garantam a Integridade do Sistema [19]. Teste de Software é indiscutivelmente uma disciplina muito ampla e está alinhada com a Qualidade do Software. Não dá para se confiar em um Software que não foi testado.

A fase de testes consiste em garantir que todas as funcionalidades desenvolvidas dentro de um projeto estão atendendo as especificações e os requisitos para que elas foram implementadas [19]. Porém, a execução manual desses testes a cada entrega se torna um trabalho dispendioso e caro. Logo, é necessário que existam automações. Testes automatizados são códigos escritos para testar o Sistema. Testes automatizados são executados mais rapidamente que os testes manuais. Por ser um código, sua execução repetitiva não é um trabalho humanamente dispendioso nem caro uma vez que já esteja implementado. Outra vantagem é que eles não cometem falhas humanamente possíveis, tais como fazer asserções incorretas, esquecer de executar testes, burlar os resultados, entre outras falhas humanas [3].

Uma vez que os testes são automáticos, sua execução ocorre a cada modificação detectada no Sistema de Controle de versão. Dentro do ciclo de integração contínua, a execução contínua dos testes garante que o que foi desenvolvido continua funcionando. O Capítulo 6 desse trabalho é dedicado a testes automatizados.

4.1.4 Inspecionar código fonte

A inspenção do código fonte busca garantir que os desenvolvedores estejam escrevendo o código fonte de acordo com regras previamente estabelecidas. Em um projeto de Software, regras de escritas de códigos e padrões de projetos precisam ser estabelecidos e seu conhecimento deve ser claro para toda equipe para que o projeto seja desenvolvido de forma saudável[19]. Isso garante que, se um desenvolvedor precisar trabalhar em um trecho de código que foi implementado por outro desenvolvedor, ele não terá problemas. Essas regras podem ir, desde definição de padrões para criação de variáveis no código, até a quantidade de linhas que um método pode ter, ignorando as

linhas de comentários. Essa inspeção visa garantir que o código escrito tenha qualidade suficiente para ser manutenível.[19]. Existem no mercado ferramentas especializadas em executar inspenção de código fonte automaticamente. Nesse trabalho usaremos o Sonar Qube, apresentado no Apêndice A.9. Nele é possível determinar padrões e regras de escrita de código e então ele verifica o código fonte e aponta as inconformidades que possam existir.

4.1.5 Disponibilizar o Software

Todo o ciclo de Integração Contínua existe por um motivo: disponibilizar softwares funcionais em um tempo realista e praticável [19]. O cliente deseja o Software funcional o mais brevemente possível. Seus processos de negócio dependem disso. Assim, não é desejável que o processo de implantação e atualização sejam um trabalho despendioso e caro. Com a entrega contínua, deve ser possível entregar software funcional em qualquer lugar a qualquer momento.

Porém entregar software funcional requer algumas boas práticas. A primeira delas é fazer um bom sistema de rotulagem dos arquivos de código fonte no Sistema de Controle de Versão. Os arquivos fontes do projeto precisam ser rotulados de acordo com suas versões. Assim é possível garantir qual é a versão de um arquivo em uma determinada versão do ambiente.

Também é preciso que se tenha um ambiente limpo para disponibilização de uma versão do software. Um ambiente limpo garantirá que o funcionamento do sistema irá ocorrer sem **suposições**. Suposição é a falsa sensação de que o ambiente de produção sempre terá os arquivos e/ou configurações necessárias para o software funcionar como esperado. Criar um ambiente limpo garantirá que o Software irá sempre funcionar, pois ele obrigará que o software possua todas as dependências que ele necessita. Logo é necessário que a criação de um ambiente limpo seja automatizada no processo de integração contínua, pois a criação manual desse ambiente pode ser custosa e dispendiosa quando executada manualmente.

Com um ambiente limpo criado, os arquivos fontes, já devidamente rotulados, são compilados, testados e inspecionados. Fechando esse ciclo, uma *release*[1] é gerada. Tal como os arquivos fontes, essa *release* deve ser rotulada. Assim será possível saber o momento em que aquele artefato foi gerado e quais as versões dos fontes estão presente nessa *release*.

[1] Entende-se por *release* o artefato binário criado à partir da compilação de uma determinada versão dos arquivos fontes.

Por fim devemos executar todos os testes existentes no projeto. Isso garantirá que essa *release* é aceitável e passível de implantação em ambiente de produção. Logo o artefato binário gerado pode ser disponibilizado para implantação.

4.1.6 Gerar documentação e *Feedback*

Um ambiente de Integração Contínua existe por um motivo: garantir que todos os componentes desenvolvidos são integráveis e que os problemas de integração serão detectados o quanto antes[19]. Sempre que uma modificação ocorrer no Sistema de Controle de Versão, deve ser possível colocar todos os componentes do Software em conjunto, à fim de disponibilizar a funcionalidade para qual o Sistema foi idealizado. Cada uma das fases anteriores visa garantir uma característica de qualidade do Software que está sendo desenvolvido. Porém, erros podem acontecer durante alguma (ou várias) das fases anteriores. Por exemplo, adicionar uma chamada de função de uma dependência que não foi importada. Quebrar um teste unitário ao modificar uma implementação e não executar uma construção local antes do commit. Ou mesmo criar um método que não esteja alinhado com as politicas da organização. Assim sendo, a ferramenta de Integração Contínua deve ter a capacidade de dar um feedback rápido sempre que ocorrer uma falha em alguma das fases anteriores. Também deve ser possível identificar em qual das etapas ocorreu a falha, pois a prioridade de uma equipe que trabalha com Integração Contínua é manter o software sempre estável. Se erros ocorrem, sua correção deve ser a prioridade máxima da equipe de desenvolvimento.[19]

Uma vez que a construção esteja estável é desejável que a ferramenta de Integração Continua gere a documentação do Software de maneira automatizada. Algumas ferramentas, por exemplo, geram páginas web à partir dos comentários do código (exemplo Javadoc[2]). Essa é uma maneira de induzir os desenvolvedores a sempre documentarem suas classes e métodos.[19]

4.2 Pipeline da Integração Continua

A Figura 4.1 mostra um modelo de pipeline de Integração Contínua. Esse modelo é implantado nesse trabalho. Nesse modelo existe três ambientes, a saber:

- **Ambiente de desenvolvimento:** ambiente onde a equipe de desenvolvimento trabalha. Os desenvolvedores criam e modificam artefatos de Software nesse ambiente. Quando suas modificações estão prontas, eles commitam e enviam os artefatos para o repositório central de Controle de Versão, conforme explicado no Capítulo 3.

[2] http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html

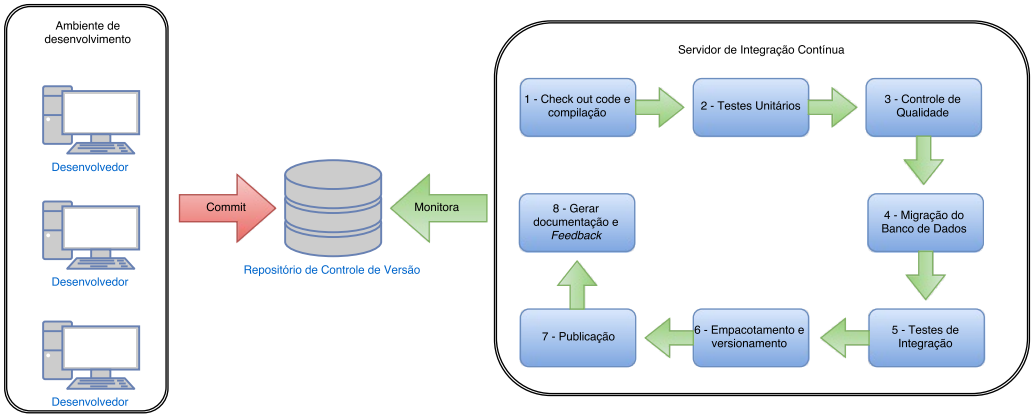

Figura 4.1: *Pipeline e ambiente típico de Integração Continua*

- **Repositorio de Controle de Versão:** local onde está o repositório centralizado onde os artefatos de software são gerenciados e versionados.
- **Ambiente de Integração Contínua:** ambiente onde o processo de Integração Contínua ocorre.

Assim que uma mudança é detectada no repositório de Controle de Versão, inicia-se o processo de Integração Contínua. Os passos executados são:

1. ***Check out code* e compilação**: o ambiente de Integração Contínua realiza um *clone* projeto do Repositório de Controle de Versão. Esse *clone* busca o estados mais recente dos artefatos no controle de versão. Em seguida ocorre a compilação do código fonte para executar os binários executáveis da aplicação;
2. **Testes unitários**: os binários gerados na fase anterior, são submetidos aos testes de unicidade. Esses testes são executados rapidamente e garatem que não existe regressões óbvias no Software. Também garante que as novas modificações não causaram impactos negativos na construção do projeto;
3. **Controle de Qualidade:** o código que passou pelos testes de unicidade é submetido à inspeção para garantir a qualidade do código gerado;
4. **Migração do Banco de Dados:** em seguida é executado o processo de migração do banco de dados, atualizando o *schema* da base para garantir a integridade do sistema;

5. **Testes de Integração:** após a migração da base de dados, são executados os testes de Integração. Esses testes são mais demorados, pois vão testar a integração da aplicação com os sistemas externos. Também pode testar a integração entre os componentes da aplicação.
6. **Empacotamento e versionamento:** uma vez executado os testes de integração e obtido sucesso é possível afirmar que a aplicação possui alguma confiabilidade. Nesse caso o código é empacotado, gerando os módulos binários que serão utilizados para construir o projeto como um todo. Esse módulos binário gerado recebem uma identificação que o versiona.
7. **Publicação:** o módulo binário versionado é publicado no repositório de gerenciamento de dependência para que seja acessível aos demais projetos e membros da equipe.
8. **Gerar documentação e *feedback*:** a documentação do projeto é gerada e disponibilizada. Assim é fácil encontrar qual o estado da documentação do software em uma determinada versão de um artefato. O *feedback*, apesar de aparecer no final do ciclo, é um processo que pode ser executado em qualquer fase para informar aos interessados o sucesso ou falha de alguma das fases da pipeline da Integração Contínua.

A execução das etapas descrita acima constituem o ciclo de vida da Integração Contínua. O conceito de Integração Contínua está fortemente ligado ao conceito de automação do processo. Tanto Martin Fowler, quanto John Fergunson Smart e Paul M. Duvall dizem que é necessário uma ferramenta que monitora as mudanças no Sistema de Controle de Versão e executa as fases do ciclo de vida da Integração Contínua. Várias ferramentas que executam o processo automatizado da Integração estão disponíveis no mercado, com destaque para o JenkinsA.3, Hudson[3] e o Travis[4]. Esse trabalho utilizou o Jenkins.

[3]Saiba mais em: http://hudson-ci.org/
[4]Saiba mais em: https://travis-ci.org/

CAPÍTULO 5

Pipeline de Entrega

Entregar software para os *stackholders* é o desafio da Entrega Contínua. Não somente entregar. É necessário entregar continuamente, de maneira confiável e repetível. É preciso conhecer quais são os passos necessários para executar a entrega. Quais são as atividades realizadas entre a concepção do Software até o momento da sua disponibilização em produção? A resposta dessa pergunta está na Pipeline de Entrega.

Uma Pipeline de Entrega é uma maneira de lidar com o processo de Entrega Contínua através da divisão do processo de entrega em várias etapas. As etapas iniciais são realizadas mais rapidamente e em geral tendem a encontrar a maioria dos problemas óbvios. Isso gera *feedbacks* mais rápidos. As etapas mais avançadas do processo tendem a serem mais lentas e dão *feedbacks* através de sondagens mais complexas. Assim a Pipeline de Entrega é o assunto central da Entrega Contínua.[12].

Jez Humble e David Farley simplificam o conceito descrevendo-o da seguinte forma [13]:

> Em um nível abstrato, a pipeline de entrega é uma manifestação automatizada do processo de pegar o Software do Controle de Versão e disponibilizar para os usuários.

Apesar da descrição simplificada, o processo é complexo. O cenário mais comum em equipes de desenvolvimento de Software que não implementam Entrega Contínua é um processo de desenvolvimento que passa por várias etapas distintas e isoladas. Steve Neely e Steve Stolt descrevem esse cenário em seu artigo [18]. Esse é o cenário da empresa Rally antes de adotarem a Entrega Contínua. Nela, o software era entregue em ciclos de oito semanas, nas quais sete semanas eram dedicadas ao desenvolvimento e uma semana era dedicada à integração e testes. Durante essa última semana toda a equipe era direcionada ao trabalho de implantar o Software em um ambiente de qualidade (entenda "ambiente de qualidade"como "ambiente de homologação") e testar manualmente todas as partes do sistema. Os defeitos encontrados eram reportados ao desenvolvimento para correção imediata. Steve Neely e Steve Stolt descrevem essa semana como cansativa e cara para a companhia. Após essa semana de testes, o processo de entrega era iniciado. Na manhã de sábado era agendada uma parada no ambiente de produção para execução

da migração dos dados. O software testado durante a última semana era transmitidos manualmente através da rede para o servidor de produção e o processo de atualização era iniciado. Várias etapas eram realizadas manualmente, vários scripts eram executados e se algo desse errado a implantação não ocorria. A equipe de desenvolvimento era acionada para corrigir os erros encontrados no ambiente de produção.

É possível observar no cenário descrito que a fase de elaboração das atividades e elicitação do que será entregue durante aquele ciclo de desenvolvimento ocorre no início do processo. Quando as necessidades são identificadas, inicia-se o processo de desenvolvimento, que perdura por algumas semanas. Ao fim da etapa do desenvolvimento inicia-se a fase de integração e testes. Se tudo der certo a entrega é executada. Mesmo sob um perspectiva de desenvolvimento iterativo, essas etapas ainda são parecidas com um modelo cascata, no qual uma fase só começa com a finalização de outra. Se problemas são identificados na semana de testes, erros complexos, muitas vezes não percebidos durante semanas de escrita de código devem ser resolvidos rapidamente. Isso nem sempre é tão simples.

Ainda assim é possível observar que apesar do processo longo e complexo, existe ainda um processo, com etapas bastante definidas e que são executadas de maneira repetível (mesmo que com intervalos de semanas). É possível trabalhar esse processo para ser mais rápido, agregar valor aos *stackholders* e resolver os problemas complexos de entrega. Para isso é desenvolvido uma pipeline de entrega.

Uma pipeline de entrega visa executar todo o processo de entrega a cada modificação realizada no controle de versão do projeto[13]. Cada mudança detectada no controle de versão dispara o processo completo da Entrega Contínua. A Figura 5.1 mostra uma visão geral do processo de entrega;

O primeiro estágio do processo automatizado é chamado de "Estágio de Commit".O servidor de Integração Continua executa a compilação, testes unitários, atualização das dependências, análises estática do código e disponibilização dos artefatos produzidos no repositório de artefatos. Essa fase deve ser rápida, uma vez que a compilação e testes de unidade executam rapidamente e não tem influências do ambiente externo. Qualquer problema identificado nessa fase passa a ser prioridade para a equipe de desenvolvimento.

Concluída essa fase inicia-se a segunda etapa do processo, chamada "Estágio de Aceitação". O software produzido é implantantado em um ambiente e ocorre a verificação se o programa iniciou corretamente. Os testes de integração são executados, juntamente com migração dos dados e testes de aceitação automatizados que existirem. Essa etapa é mais lenta que o estágio de commit por ser preciso providenciar o ambiente de teste, configurá-lo e implantar o software produzido pela etapa de commit.

Em seguida três etapas podem serem executadas. Uma das etapas é o "Estágio de Capacidade". A implantação automatizada, juntamente com os testes de integração são

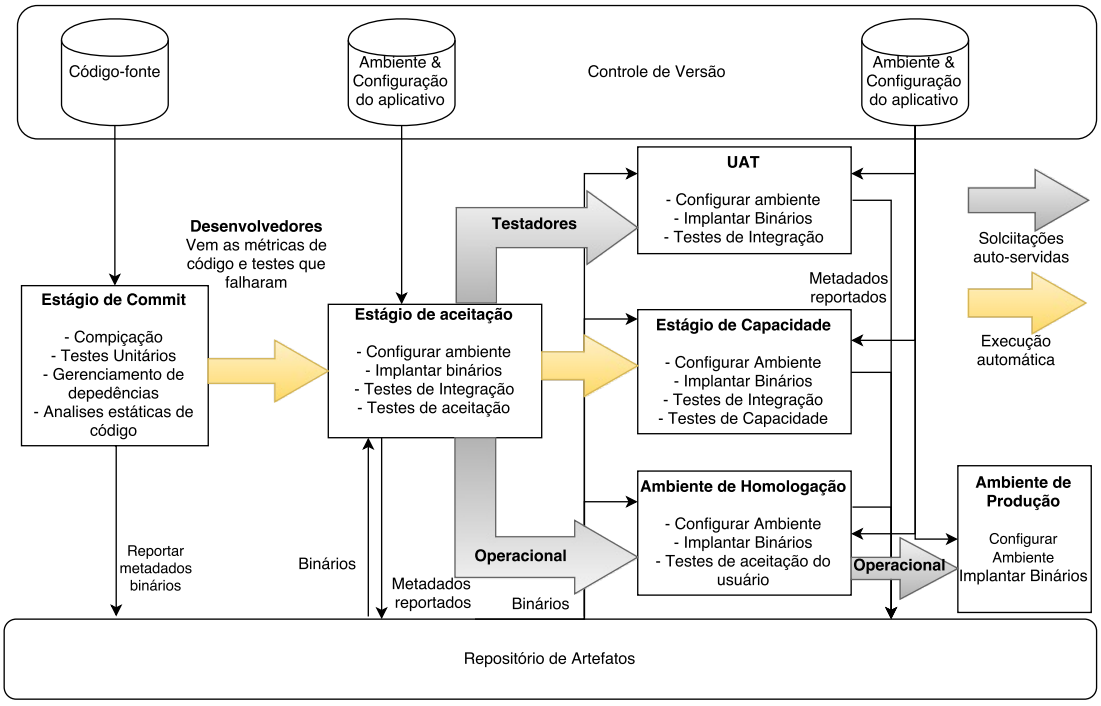

Figura 5.1: *Visão geral de uma pipeline de Entrega Contínua, segundo Humble e Farley[13]*

executados novamente adicionando os testes de capacidade. Esses são testes que verificam se o projeto conseguirá responder a demanda de produção, através de testes automatizado de carga e capacidade.

As equipes de testes e operações devem conseguir criar ambientes para testes de uma forma simples. Essa facilidade significa que, usando as ferramentas de entrega contínua, essas colaboradores devem conseguir providenciar um ambiente completo em uma determinada versão do Sistema, através de um simples clique. Devem existir funcionalidades no Sistema de Entrega Contínua onde essas equipes consigam um ambiente completo, semelhante ao de produção, através de uma requisição simples, como clicar em um botão. Dois ambientes poderão ser gerados. O ambiente "UAT"(*User Acceptance Tests*), ou ambiente de testes de aceitação que é produzido para que a equipe de testes consiga executar seus testes no sistema. Também deve ser possível que o operacional consiga construir um "Ambiente de Homologação". Esse ambiente é mais próximo do ambiente de produção. Ele existe para que os usuários finais consigam testar as novas versões do Sistema.

Uma vez que a aplicação é testada e aprovada é preciso executar sua entrega em ambiente de produção. Esse é um momento crítico para o processo, pois todo o trabalho visa manter o ambiente de produção sempre estável e funcional. As modificações desenvolvidas não pode impactar negativamente o ambiente. Por isso existem as **Estratégias de Entrega**. As estratégias de entrega descrevem padrões de como implantar em produção a versão do Software testada e aprovada.

Algumas estratégias são descritas à seguir:

- **Mudança Paralela:**[22] é um padrão também conhecido como **expansão e contração**. O objetivo desse padrão é implementar mudanças nas interfaces de uma forma segura, dividindo a execução das mudanças em três etapas: expansão, migração e contração. Na fase de expansão, novas funcionalidades são criadas e elas convivem juntamente com as funcionalidades já existentes. Com as novas funcionalidades criadas, os fluxos que usavam as funcionalidades anteriores são gradativamente modificadas para as novas, de forma que as coisas continuem funcionando. Mesmo se uma entrega for solicitada no meio do processo, nenhuma funcionalidade deixa de funcionar, nova ou antiga. Quando a equipe sentir que a maior parte das funcionalidades foram migradas, então faz-se o processo de contração, onde as funcionalidades antigas são eliminadas e as quebras de compatibilidade que porventura ainda existam são facilmente corrigidas.
- **Migração *Blue-Green*:**[21] no padrão de entrega *blue-green*, ou, azul-verde, o risco das implantações em produção é mitigado através do redirecionamento dos usuários para o servidor definido pela equipe de desenvolvimento, conforme é demandado. Nesse padrão, o ambiente de produção é chamado de **ambiente azul**. Esse ambiente é o que possui a versão do Software que está sendo utilizado em produção. Um ambiente é iniciado paralelamente ao ambiente de produção, com as novas modificações desenvolvidas pelas equipes. Esse é o **ambiente verde**. Esse ambiente é testado pela equipe de teste e os *stackholders* do projeto. Quando é dado um ok sobre o ambiente verde, o processo de entrega consiste em redirecionar os usuários para o ambiente verde, tornando esse, o ambiente de produção, enquanto o ambiente azul é mantido por segurança. Se algum problema não previsto acontecer no ambiente verde, o fluxo é redirecionado para o ambiente anterior. Caso tudo ocorra bem, o ambiente azul é eliminado.
- **Implantação Canário:**[20] no padrão implantação canário, o risco da disponibilização de uma nova versão do software é mitigado através do redirecionamento gradatívo dos usuários para um novo ambiente. Nesse processo, um ambiente com a nova versão do software é iniciado e um grupo seleto de usuários é redirecionado para esse novo ambiente. Esse ambiente é monitorado e conforme a confiança vai aumentando, gradativamente mais usuários são migrados para esse ambiente. Quando todos os usuários já estiverem consumindo recursos do novo ambiente, o ambiente anterior é eliminado. Se problemas ocorrerem, os clientes voltam para o ambiente antigo e o novo ambiente recebe as manutenções para funcionar efetivamente.
- ***Feature Toggle*:**[9] no padrão *Feature Toggle*, a entrega de funcionalidades ocorre de maneira mais fina e seletiva. Uma funcionalidade é desenvolvida e alguns

fluxo de controle determinam quando essa funcionalidade vai ser ativada ou não. Assim, uma nova versão de software pode conter uma determinada funcionalidade solicitada, porém ela estará oculta e só será acionada mediante alguma configuração específica.

- **Servidor Fénix:**[10] esse padrão é voltado ao provisionamento de ambientes. Nesse padrão, cada vez que uma modificação for solicitada em um ambiente, essa modificação é feita de maneira versionada e o servidor é eliminado, criando outro servidor novo onde essa modificação é executada automaticamente. Assim, as modificações são devidamente rastreadas e os ambientes podem ser provisionaddos a qualquer momento, com qualquer versão possível.

Uma vez definida a estratégia de entrega é importante observar que cada uma das etapas da Pipeline de Entrega deve proporcionar *feedbacks* para os *stackholders*. A Figura 5.2 mostra um exemplo de como deve ser esse processo.

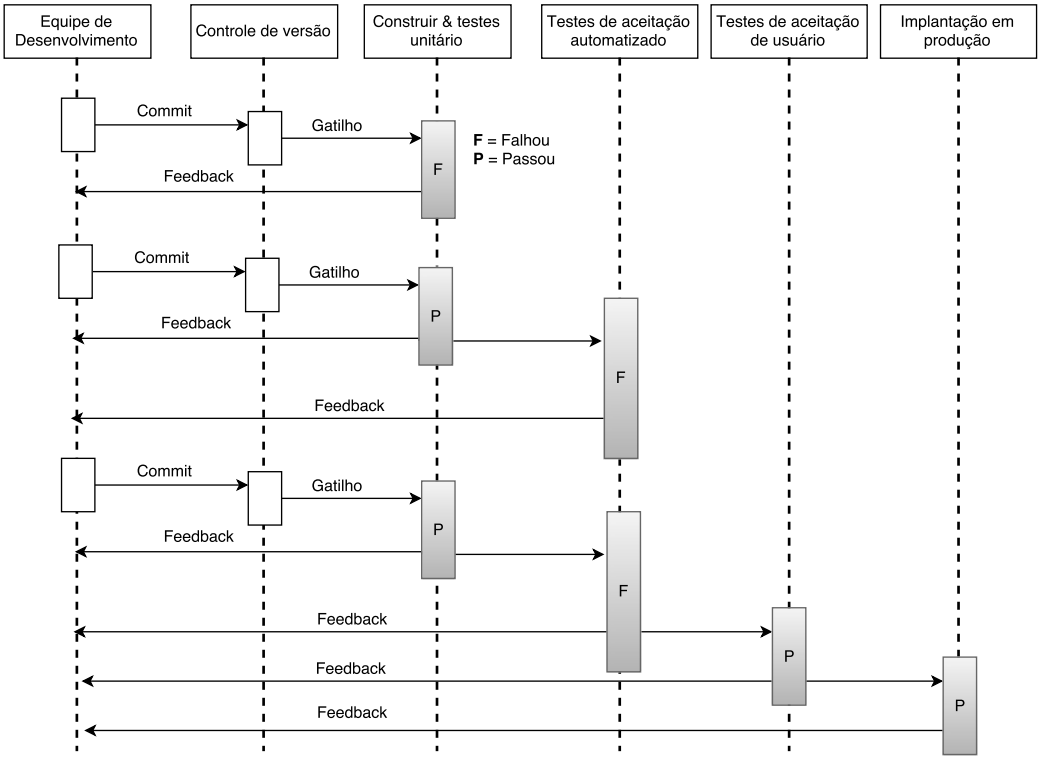

Figura 5.2: *Processo de Feedback sobre a execução de cada etapa do processos, segundo Humble e Farley[13]*

Cada fase deve gerar um *feedback* tanto sobre o sucesso como para a falha. Isso mantém os *stackholders* cientes do estado do projeto a cada estágio, desde o desenvolvimento, até a entrega. *Feedbacks* mais rápidos trazem mais confiança no produto. Porém, conforme dito anteriormente, a confiabilidade no produto final será inversamente proporcional ao tempo de entrega do *feedback* de cada etapa da pipeline. Etapas iniciais tem repostas mais rápida, porém a confiança será menor. Etapas finais são mais confiáveis,

porém seu tempo de execução será maior. A Figura 5.3 demonstra essa proporcionalidade.

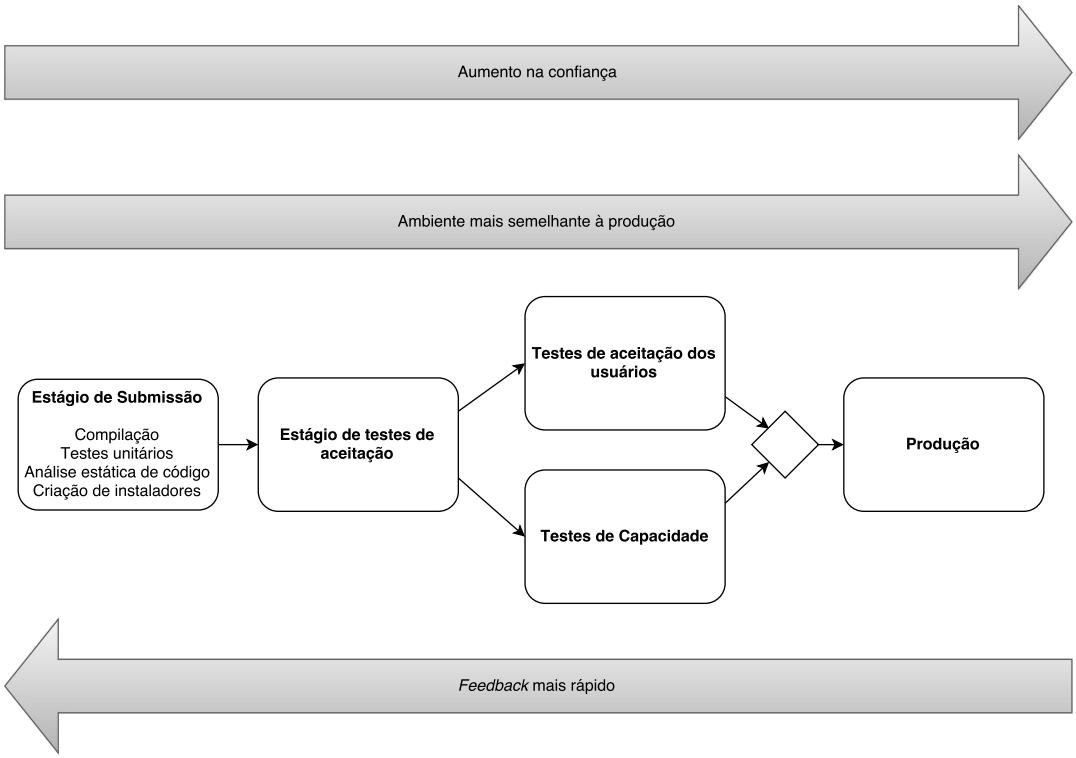

Figura 5.3: *Relação entre confiabilidade e tempo de execução de cada etapa da Entrega Contínua, segundo Humble e Farley[13]*

Assim, definir uma pipeline de entrega possibilita compreender, ainda no início do projeto como será a entrega para o cliente final. Através dessa abordagem a equipe de desenvolvimento estará ciente de como será o ambiente do cliente e o cenário onde o Software vai executar. Consequentemente é possível decidir, nos estágios iniciais do projeto, formas de resolver problemas que, em outras abordagens, só seriam descobertos quando a aplicação fosse para produção.

5.1 A cultura DevOps

Entrega Contínua é uma prática de desenvolvimento que trabalha com métodos ágeis. Quando se fala em métodos ágeis, significa o rompimento da ídeia de isolamento entre conceitos da Engenharia de Software, como Gerenciamento de Requisitos, Análise do Projeto, Desenvolvimento e Testes. Ao invés desses conceitos trabalharem em etapas individuais e isoladas, elas aparecem na mesma etapa.

Porém, romper os isolamentos entre as práticas comuns da Engenharia de Software não é o suficiente para extrair todo o valor de um método ágil. É preciso ir além e

romper a barreira entre as equipes, principalmente as barreiras que existem entre o desenvolvimento e a operação. Assim, surge a prática de **DevOps**. DevOps é um conjunto de práticas que destina-se a diminuir o tempo entre a realização de uma alteração no sistema e a disponibilização dessa mudança em produção, assegurando alta qualidade[15].

DevOps é uma mistura dos termos *Development and Operations*, traduzindo para *Desenvolvimento e Operação*[16]. Uma caracteristica muito forte dessa prática é a colaboração que existe entre o desenvolvimento e a operação. DevOps não é apenas a aplicação de ferramentas, ou uma equipe existente na organização, e sim uma cultura organizacional que engloba todo o time[27].

Jaz Humble e David Farley defendem que a cultura DevOps tem dois objetivos[13]:

1. Reforçar a colaboração entre as atividades de desenvolvimento e operação;
2. Usar princípios ágeis como a automatização para gerenciar o provisionamento de ambientes e suas configurações.

Essa prática visa reforçar atitudes positivas nas equipes, principalmente o compartilhamento da responsabilidade. Quando um problema acontece, a solução é responsabilidade de todos, não somente da equipe de desenvolvimento ou operação[27]. Quando a responsabilidade é compartilhada, a equipe de desenvolvimento se preocupa em desenvolver a aplicação com foco na manutenção e operação, facilitando o trabalho das equipes de operação. Já a equipe de operação consegue compartilhar facilmente a visão de negócio com os desenvolvedores, auxiliando-os no entendimento do problema e colaborando com a construção das soluções.

Para que essa colaboração aconteça de maneira mais eficiente, é preciso dar autonomia para as equipes. Todos devem se sentir confiantes em propor mudanças e tomar decisões. Assim, o processo de tomada de decisões se torna mais simples. Isso envolve tornar os times confiantes, mudar a maneira como os riscos são gerenciados e cria um ambiente livre do medo de falhas[27].

Quando a prática de DevOps é aplicada, a qualidade do produto é construída dentro do processo de desenvolvimento e reverbera por toda organização. Automatizando processos complexos, e valorando os *feedbacks* é possível trazer todo o valor que a cultura DevOps propõe.

DevOps é um assunto interessante para se aplicar em organizações que desenvolvem e mantêm softwares, contudo não é o foco principal desse trabalho. É importante reinterar que a prática existe e está fortemente ligada aos processos que envolve Entrega Contínua.

CAPÍTULO 6

Estratégia de Testes

O foco da Entrega Contínua é a automatização de todo o processo de desenvolvimento e entrega, conforme visto em Pipelines de Entrega no Capítulo 5. É importante que o processo de Entrega seja pensado desde o início do projeto para garantir que o Software produzido está de acordo com o ambiente em que será executado no cliente. Mas só entregar não é suficiente. É preciso que o Software execute corretamente suas funcionalidades, atendendo os requisitos funcionais e não funcionais elicitados. Também é preciso que esse software atenda as expectativas do usuário. Um software não entregue não agrega valor, assim como um Software que não atende aos objetivos para o qual ele foi concebido. Por isso o software em desenvolvimento deve passar por um processo de verificação e validação[24]

Para garantir o atendimento dos requisitos é que existe a fase de testes. É na fase de teste que a aplicação será posta à prova a fim de verificar e validar se a aplicação atende as necessidades que motivaram sua criação. É através dos testes que é possível identificar a qualidade e confiabilidade do Software entregue. Por isso é preciso pensar em como testar o software.

Nos modelos de desenvolvimentos tradicionais a fase de testes ocorre após a fase de codificação. Somente após a equipe de desenvolvimento definir que um software está pronto, ou pelo menos um conjunto de funcionalidades negociadas em um determinado ciclo, é que os testes serão executados. A equipe de testes implanta o Sistema em um ambiente semelhante ao de produção e executa todas as funcionalidades do Software a fim de encontrar erros ou inconformidades no atendimento dos requisitos. Esse cenário é semelhante ao descrito no artigo Steve Neely e Steve Stolt [18]. Essa abordagem é conhecida como testes manuais.

Testes manuais são caros. Se uma pessoa precisa testar todas as funcionalidades de um software manualmente, dependendo do tamanho desse software, o resultado dos testes provavelmente não serão entregues em um tempo viável. Outro problema é que recursos humanos são caros e uma equipe de testes consome recursos valiosos em um projeto. Testes manuais são propensos falhas. Testar é um trabalho repetítivel e as pessoas tendem a perder o foco quando executam tarefas repetitíveis e que requerem muita atenção

[3] [24].

Assim é necessário se pensar em uma outra abordagem para os testes. Uma maneira eficiente de testar é automatizando o processo de teste. Testes em geral são situações repetitíveis que tem um processo bem definido, a saber[24]:

- **Preparação:** nessa fase o software é configurado com as entradas que o sistema deverá processar;
- **Execução:** as entradas são submetidas ao sistema à fim de que o software faça as operações que ele foi programado para fazer mediante aquela entrada;
- **Validação:** após a execução, o estado da aplicação é validado a fim de identificar se o resultado alcançado pelo processamento está de acordo com o que foi projetado.

Pelos motivos apresentados acima é viável que os testes sejam também automatizados. Testes automatizados tem diversas vantagens, como [4]:

- A execução de um teste automatizado é mais rápida que um teste manual.
- Uma vez automatizado, um teste pode ser reexecutado repetidas vezes a um custo muito baixo.
- Testes automatizados aumentam a produtividade. O desenvolvedor gasta menos tempo testando e mais tempo desenvolvendo.
- Os erros são encontrados rapidamente.
- O índice de regressões diminui. Uma vez escrito um teste automatizado, se o problema tratado voltar a ocorrer o teste falhará, forçando o desenvolvedor a tratar o problema.
- Estimula refatorações. É possível avaliar o impacto de uma refatoração, pois os testes vão garantir que as funcionalidades continuam funcionando após a refatoração.
- Diminui o tempo gasto com depuração[1].

Existem muitas outras vantagens em automatizar os testes de um Software. As apresentadas aqui são as mais relevantes.

Execução de teste automatizado é parte fundamental na Pipeline de Entrega Contínua. Assim, os testes devem ser uma atividade transversal presente em todas as fases do projeto e devem ser executados continuamente, desde o início [13]. Pensando em testes como uma atividade transversal à todas as etapas do processo, o software gerado nascerá auto-testado e com uma cobertura de qualidade maior do que projetos onde os testes são feitos depois.

Contudo os testes manuais ainda são necessários. Alguns testes dificilmente são passíveis de automatização, por exemplo, testes de usabilidade de interface ou mesmo

[1] Depuração é técnica que tem por objetivo localizar e corrigir defeitos manifestados em um software

testes de interação entre Software e o ambiente corporativo onde ele está sendo executado. É difícil testar automaticamente se um software causará algum efeito colateral indesejado no macro-ambiente[2] onde foi implantado. Assim, os testes automatizados devem existir para verificar se o software executa corretamente as operações propostas enquanto os testes manuais devem existir para verificar se o software atende as expectativas do usuário [24].

Existem, basicamente, três tipos de testes automatizados. Eles serão descritos nas próximas sessões. Em seguida será apresentado uma estratégia de desenvolvimento orientada à testes.

6.1 Testes de Unicidade

O primeiro tipo de teste são os **testes de unicidade**. Testes de unicidade são testes executados nos menores artefatos de código. Em projetos Orientados à Objetos, por exemplo, esses artefatos são as classes. Eles existem para testar a funcionalidade dos objetos e suas operações individuais [24].

Os testes de unicidade testam pequenas porções de códigos e a execução de suas funções de maneira isolada. Isso garante que elas entregam os resultados esperados, mediante sua execução. Testes de unicidade não são testes que executam a funcionalidade integral do sistema, e sim a funcionalidade das pequenas partes individuais que compõe o todo. São fáceis de escrever e rápidos para executar. O *feedback* entregue pela execução dos testes de unicidade é mais rápido. Por testar partes pequenas do sistema, ele, em geral, cria uma cobertura maior das funcionalidades do sistema e evita regressões óbvias. Cobertura e regressão são explicadas na Seção de desenvolvimento orientado à testes 6.4.

Uma caracteristica importante desse tipo de teste é que ele precisa ser executado sem ser influênciado por fatores externos. Se a aplicação consome algum recurso de outra aplicação ou módulo, é preciso garantir que esses serviços externos vão responder conforme esperado. Para isso existem os ***Simuladores***. *Simuladores* são objetos criados pelo programador que fornece as mesmas interfaces que os sistemas externos porém simulam respostas desses sistemas, conforme a necessidade do programador[3]. Essa técnica é importante para os testes automatizdos, pois, às vezes não é possível garantir como um serviço externo irá responder uma requisição conforme o teste espera. Esses sistemas podem ter modificado os modelos de suas respostas ou mesmo estarem indisponíveis. Os

[2]No contexto apresentado neste trabalho, "macro-ambiente"é um ambiente amplo que envolve outros fatores externos além do software. Por exemplo: é possível testar automaticamente o seguinte cenário: se a execucação funcionalidade X executar 60% mais rápido será possível alavancar o percentual de vendas em 8% conforme definido na estratégia da organização? Esse tipo de cenário dificilmente será testado pela equipe de desenvolvimento de Software, visto que o mercado é imprevisível e outros fatores podem influênciar nas vendas de uma organização. Não faz sentido tentar automatizar esse tipo de teste.

testes ficam frágeis se dependerem diretamente da disponibilidade desses sistemas. Por isso, quem escreve o teste deve criar simuladores e os configurar para reagir conforme sua necessidade. Assim o programador tem um controle maior no fluxo da aplicação, pois ele pode configurar o objeto para responder conforme ele precisar.

Dentro de uma Pipeline de Entrega Contínua, os testes unitários são executados na fase de Integração Contínua, logo após a compilação do código. Por serem rápidos de executar e entregarem um *feedback* rápido, sua execução automatizada nas primeiras fases do ciclo de entrega garante uma boa confiabilidade nos artefatos produzidos.

6.2 Testes de Integração

Outro tipo de teste automatizado são os **testes de integração**. É muito raro desenvolver aplicações que executam totalmente isoladas de todo o resto do ecossistema de uma corporação. Aplicações precisam interagir com outros sistemas, trocar informações, solicitar serviços ou disponibilizar serviços. Essas aplicações devem fornecer interfaces com as quais devem ser possível interagir. Logo é preciso testar a funcionalidade dessas interfaces [24].

Tão importante quanto testar a funcionalidade interna da aplicação, é testar as interações com sistemas externos. Testes de integração existem para isso: testar o comportamento da aplicação com sistemas externos. Também testam as interfaces disponibilizadas pelos componentes desenvolvidos a fim de verificar se elas respondem conforme foram especificados. Podem testar, também, a integração entre componentes desenvolvidos internamente pela equipe.[4]

Por exemplo: um sistema de informação que armazena registros de vendas e consulta esses registros para gerar relatórios. Esse sistema precisa armazenar, buscar, atualizar e excluir informações durante sua execução. Cada interação faz uma conexão com uma base de dados. A troca de informações com o Banco de Dados, em geral, ocorre através da linguagem SQL[3]. A dúvida é: os SQLs presentes na aplicação são funcionais? Como testar se os SQLs escritos pelo programador, ou mesmo os SQLs gerados automaticamente por algum *framework* ORM[4] serão efetivos quando executados? A resposta para essa pergunta é: testando a integração com o Banco de Dados. Logo os testes de integração configuram, nesse caso, uma conexão com o banco de dados e submete os SQL presentes na aplicação, para verificar se eles estão corretamente escritos

[3]SQL é um acrônimo de *Structured Query Lenguage*. Essa é a linguagem utilizada nos bancos de dados para realizar operações como buscas, inserções, atualizações ou deleções. Outros procedimentos podem ser escritos com essa linguagem

[4]ORM: *Object Relational Model*, ou Modelo Objeto Relacional. É uma técnica que encapsula toda complexidade em transformar modelos Orientados à Objetos em Modelos Relacionais.

e se os resultados gerados por esses SQLs são os que a aplicação precisa para continuar sua execução.

Outro possível cenário de testes de integração é quando uma funcionalidade necessita da integração entre vários módulos para realizar sua operação corretamente. Quando vários módulos precisam ser integrados para atender uma funcionalidade, testes de integração devem ser escritos para verificar se essa integração irá gerar os resultados esperados.

Justamente por essa caracteristica de precisar se comunicar efetivamente com outros módulos ou aplicações externas, esses testes são mais lentos, se comparados com os testes de unicidade. Todavia testes de integração trazem a vantagem de garantir que a aplicação desenvolvida interage bem com outros sistemas e faz uso correto das interfaces disponibilizadas. Isso aumenta a confiança na aplicação aprovada nessa suíte de testes. Em uma Pipeline de Entrega Contínua, esses testes podem ser executados na fase de Estágio de Aceitação.

6.3 Testes de Sistema

Uma vez que já foi testado as unidades individuais do sistema e também sua integração entre com os módulos e os sistemas externos, faz sentido que seja testado o sistema em sua totalidade. Para isso existem os **testes de sistema**

Nessa categoria de testes automatizados, o sistema é implantado em um ambiente semelhante ao de produção e seus serviços são disponibilizados externamente para serem chamados. Por exemplo: um sistema que disponibiliza uma comunicação REST[5] externamente para processamento de informações. Esse sistema é implantado e os testes automatizados chamam as funcionalidades através de URIs[6] REST e os resultados gerados são validados.

Esses testes são os mais caros para serem executados, porque:

- primeiro: o sistema precisa estar em um ambiente semelhante ao de produção. Isso gera um custo adicional para providenciar a infraestrutura para esse ambiente.
- segundo: esse ambiente precisa estar pré-configurado. Executando em um ambiente de testes, é necessário, por exemplo, pré-carregar uma massa de dados para que os bancos de dados consigam responder as chamadas. Pode ser preciso criar simuladores complexo para responder como sistemas externos que podem não disponibilizar serviços semelhantes ao de produção para testes.

[5]REST é um modelo de comunicação entre sistemas que usa como base o protocolo HTTP e suas caracteristicas

[6]URI é um modelo de representação que indica a disponibilização de um determinado serviço. Uma URI é uma URL basicamente

- tercerio: esses testes são os mais dificeis de se escrever. Um desenvolvedor precisa desenvolver esse testes através usando as camadas mais externas do Software. Ele precisa desenvolver os testes sabendo exatamente como a aplicação deve se comportar como um todo. A preparação dos dados e as validações podem se tornar complexas conforme o tamanho do teste. Esses testes podem testar requisitos não-funcionais também. Os testes automatizados de requisitos não funcionais requerem bastante conhecimento sobre o Software e das ferramentas que auxiliam esse tipo de teste.

Esses testes são os mais lentos para serem executados. É preciso providenciar ambiente, implantar aplicação, carregar massa de dados e simular chamadas externas para então iniciar a execução desses testes. Em contra-partida esses testes automatizados são os mais próximos do ambiente de produção. A confiança gerada pelo resultado desses testes é muito maior que os testes de unicidade e integração. Outra caracteristica importante é que esses testes podem validar requisitos não funcionais através de testes de capacidade do serviço, tempo de resposta, caracteristicas de segurança, validação de acessos, entre outros. O valor e a confiança gerada pelos resultados desses testes são importantes. No ciclo de Entrega Contínua, esses testes são os últimos a serem executados, principalmente pela sua necessidade de um ambiente mais próximo ao ambiente de produção.

6.4 *Desenvolvimento orientado a testes*

Testes automatizados ganharam uma importância significativa nas equipes de desenvolvimento atualmente. Tornar a fase de testes uma etapa transversal ao processo de desenvolvimento trouxe resultados expressívos para as equipes.

O método de **desenvolvimento orientado à testes** surgiu para levar à etapa de desenvolvimento a prática de escrever testes automatizados. Esse método é popularmente conhecido como ***TDD***, um acrônimo da expressão em inglês *Test-Driven Development*. Nessa prática de desenvolvimento, o desenvolvedor inicia a implementação criando um código de teste que verifica se a execução da funcionalidade a ser implementada retornará o resultado esperado. Somente após escrever o teste, o desenvolvedor implementa o código que executa aquela funcionalidade[3].

O processo de codificação segue os seguintes passos[24]:

1. O desenvolvedor identifica um incremento de funcionalidade. Esse incremento deve ser pequeno e possível de ser escrito em poucas linhas de código.
2. O desenvolvedor escreve um código de teste que verifica o resultado esperado daquele incremento. Esse código de teste vai garantir que a funcionalidade foi atendida quando executada.

3. O desenvolvedor executa o código de teste, junto com todos os outros testes existentes a fim de verificar se os testes anteriores estão sendo executados com sucesso e o novo caso de teste está falhando. É esperado que o novo caso de teste falhe, pois até esse momento o desenvolvedor não implementou o incremento indentificado no primeiro item.
4. O desenvolvedor implementa código de incremento identificado de forma que ele passe no teste. O desenvolvedor deve implementar o código mais simples possível para atender o caso de teste[3]. Essa abordagem serve para que o desenvolvedor não crie códigos complexos. Também pode ser possível que o desenvolvedor refatore o código já existente a fim de atender ao novo incremento.
5. Após implementado o novo incremento o desenvolvedor deve executar novamente todos os testes. Todos os testes devem passar. Se todos os testes forem executados com sucesso, o desenvolvedor volta a identificar uma nova funcionalidade ou incremento no código. Caso algum teste falhe, cabe ao desenvolvedor refatorar o código para atender todos os casos de teste.

A prática de **TDD** é bastante utilizada para escrita de testes unitários e de integração vistos nas sessões 6.1 e 6.2 respectivamente. Ao utilizar a abordagem de **testar primeiro**, vários benefícios são alcançados[24], tais como:

- Programadores conseguem **compreender** melhor o sistema que estão desenvolvendo. Ao escrever testes para pequenos incrementos, o desenvolvedor entende claramento o que aquela funcionalidade deve fazer e como ela deve se comportar.
- O código produzido terá uma **cobertura de código** maior. Um código é dito "coberto"quando existe um teste automatizado que verifica sua execução. Como o sistema é desenvolvido em pequenas interações, cada uma sendo implementada somente após a criação de um teste automatizado que a verifica, o código final consequentemente será mais coberto por testes automatizados que garantem a funcionalidade.
- A incidência de **regressão** diminui. Uma vez escrito um teste automatizado, ele pode ser executado sempre que preciso, rapidamente e a um baixo custo. Esses códigos de teste servem como **testes de regressão**. Testes de regressão são um conjunto de testes que uma vez executados para verificar o software, devem ser reexecutados toda vez que o software sofrer modificações a fim de identificar se novos bugs não foram inseridos no sistema.
- Menor grau de **depuração**. Cada teste escrito cobre um pequeno incremento do software. Um teste que falha estará associado a um pequeno incremento de código. Não deve ser difícil identificar onde os erros estão quando um teste automatizado acusa falha. Sommerville diz em seu livro [24] que em software

desenvolvidos orientados à testes quase nunca é necessário utilizar sistemas de depuração automáticos.

- O código de teste serve como **documentação** do sistema. O código de teste descreve como o sistema deve se comportar e o que aquela determinada funcionalidade deve fazer. Através do código de teste é possível entender o software e usar esse código como documentação para definir o que o sistema deve ou não fazer.
- Melhora a **qualidade** do código produzido. No processo de escrita de código orientado à testes o desenvolvedor deve implementar da maneira mais simples possível o código para atender determinado caso de teste. Assim, o desenvolvedor tende a escrever códigos mais enxutos e fáceis de manter.
- Estimula as **refatorações**. O desenvolvedor é encorajado a refatorar o código e se sente seguro ao fazê-lo, uma vez que os testes automatizados vão validar se a refatoração feita não surtiu efeitos negativos no sistema. Um código refatorado que já está coberto pode ser submetido a execução automatizada dos testes para garantir que novos bugs não surgiram.

Essa prática tem sido adotada com sucesso no mercado, principalmente em empresas de desenvolvimento de pequenos e médio porte [24]. Desenvolvedores que a praticam se mostram bastante satisfeitos.

Para a Entrega Contínua essa abordagem é fortemente aconselhada. Testes automatizados servirão como fonte de *feedbacks*. Através da aplicação dessa prática, juntamente com o conhecimento dos diversos níveis de testes, será possível incluir na Pipeline de Entrega a execução automatizada dessa suíte que garantirá a qualidade do Software. Através de uma estratégia bem definida de testes os desenvolvedores serão encorajados a escrever testes durante o desenvolvimento. A probabilidade de que os *stackholders* recebam um produto que atenda suas necessidades e seus propósitos, com código final mais limpo, compreensível e manutenível será maior.

CAPÍTULO 7

Implantação Contínua

Finalmente cabe esclarecer a diferença entre os termos "Entrega Contínua"e "Implantação Contínua". É comum a confusão que existe entre os termos. Afinal, qual a diferença? Jez Humble e David Farley dizem que Implantação Contínua é o ato de implantar o Software diretamente em produção após cada mudança no Sistema de Controle de Versão. Já a Entrega Contínua é a capacidade de disponibilizar em produção um Software funcional quando se desejar, porém esse processo não é automático. Ele precisa de intervenção manual para ocorrer.[13]. A Figura 7.1 mostra a direfença principal entre Entrega Contínua e Implantação Contínua.

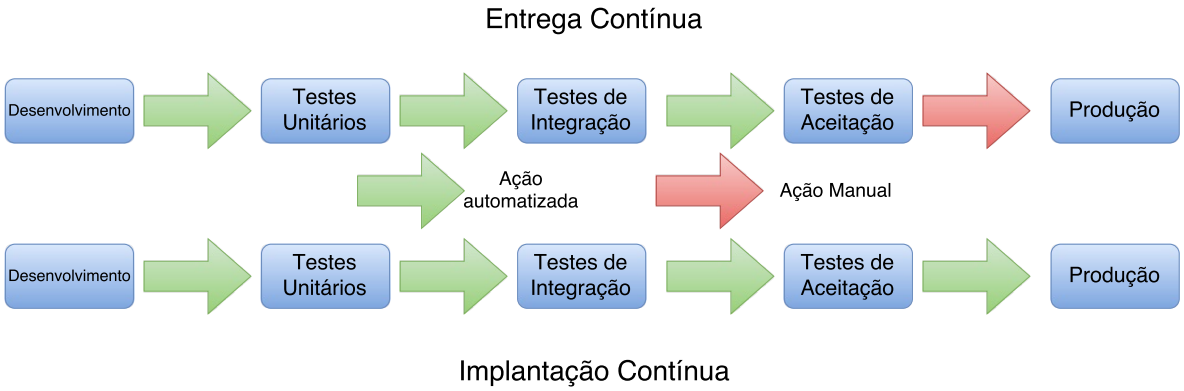

Figura 7.1: *Diferença entre o processo de Entrega Contínua e Implantação Contínua.*

Enquanto na Entrega Contínua, o processo de implantar a nova versão do Software precisa de uma intervenção manual, na Implantação Contínua essa implantação é automática e ocorre diretamente no ambiente alvo. Jez Humble e David Farley tratam em seu livro esse "ambiente alvo"como o ambiente de produção. Ou seja, a Implantação Contínua pode ser considerada o próximo passo após a Entrega Contínua.

Para que essa abordagem funcione é necessário que exista testes automatizados em toda a aplicação: testes unitários, testes de integração, testes de componentes, testes de aceitação (funcionais e não funcionais) e testes de sistema. Esses testes devem cobrir

o máximo possível da aplicação de forma que seja confiável disponibilizar em produção um software que tenha passado integralmente em todos os testes.

Isso faz com que a implantação contínua tenha alguns benefícios. O risco, por exemplo, é reduzido. A Entrega Contínua pressupõe que as mudanças realizadas no software sejam feitas gradativamente. Cada mudança deve afetar apenas uma pequena porção do código. Assim, cada mudança passa por todo o ciclo de integração, testes e implantação, sendo finalmente implantada em produção somente se ela for aprovada em todos os estágios de aceitação. Uma modificação simples pode ser facilmente revertida. O retorno de versão é menos problemático. Assim os riscos de modificações causarem problemas significativos são reduzidos.

Outra vantagem da Implantação Contínua é que somos forçados a fazer a coisa certa. Não será possível ter Implantação Contínua sem que todo o processo de Integração, Implantação e Entrega seja automatizado. Sem um conjunto de testes automatizados confiáveis e compreensíveis, sem testes de sistemas e de aceitação em um ambiente similar ao de produção, sem que o processo de implantação seja totalmente automatizado sem nenhuma interferência humana, não pode-se dizer que se tem um ambiente de Implantação Contínua confiável. Sem um processo completo de Integração e Testes, não há como garantir a confiabilidade de um software para ser implantado diretamente em produção[13].

O processo de implantação Contínua torna possível o que vem sendo discutido por anos sobre processo de entrega de Software: automatizar todo o processo, desde a compilação, testes, disponibilização, validações e implantação para que ele seja confiável e repetível. Implantação Contínua é a conclusão lógica para essa discussão. Ela muda o paradigma de como o software é entregue. Jaz Humble e David Farley diz que não há boas razões para não querer entregar o software a cada modificação[13].

Porém nem sempre é desejável que o softwares vá diretamente para produção. Algumas companhias possuem restrições de conformidade e exigem autorizações para disponibilizar em produção uma nova versão de um produto [13]. Logo, implantação contínua não é uma prática para qualquer software que esteja sendo desenvolvido. Essa prática afeta a cultura de uma organização e da equipe de desenvolvimento. É preciso muita maturidade e conhecimento para se implantar um ambiente de Implantação Contínua.

Apesar de ser uma evolução importante na maneira como o Software é entregue, esse trabalho não irá aprofundar na discução da Implantação Contínua. Contudo, esse método de entrega será utilizado no provisionamento de ambientes de qualidade. No estudo de caso, apresentado à seguir, a disponibilização do Software no ambiente de homologação será feita a cada modificaão detectada no sistema de Controle de Versão. O impacto significativo da Implantação Contínua ocorre quando se tenta implantar

continuamente em produção.

CAPÍTULO 8

Estudo de Caso

Nos capituloas anteriores foram apresentados os assuntos principais que envolvem a Entrega Contínua. A abordagem desse trabalho foca em pensar na Entrega Contínua desde o início do projeto, antes mesmo da primeira submissão de código fonte ao sistema de versionamento. O assunto Entrega Contínua é amplo. Construir um ambiente completo, com todas os detalhes que envolve tal assunto, pode ser um trabalho despendioso e caro. A questão levantada nesse trabalho é a seguinte: compensa pensar em Entrega Contínua em um projeto ainda em suas primeiras etapas de construção? Nesse estudo de caso será feito um experimento levando em conta essa abordagem.

Esse estudo de caso consiste em duas etapas: a primeira etapa foca na configuração da infraestrutura necessária para executar a entrega contínua e a tomada de decisões de como a aplicação será entregue. A segunda etapa consiste no desenvolvimento de um Software fictício sobre essa infraestrutura configurada para apoiar o desenvolvimento. Após essas duas etapas serão relatados os resultados alcançados no desenvolvimento da aplicação.

8.1 Proposta de Projeto

Nessa Seção será abordada uma proposta de Software para ser construído sobre a perspectiva da Entrega Contínua. Como o foco do trabalho não é o Software em si e sim os processos que envolvem sua construção e entrega, o sistema abordado será para uma situação fictícia. O Software em sí será bastante simples. O objetivo é constatar que, uma vez submetido um código fonte, o software será compilado, testado, analisado, construído e entregue automaticamente em ambientes de teste e, somente quando requisitado, em produção.

8.1.1 Software a ser desenvolvido

Abaixo é descrita a necessidade de um Software:

A empresa "XYZ Informática"presta serviços de manutenção de computadores. Alguns dos serviços prestados são formatação de computadores, instalação de programas e drivers, escaneamento e varreduras de vírus, instalação de dispositivos periféricos como leitoras de CDs e DVDs, construção e configuração de pequenas redes e suporte técnico a impressoras e scanners. Os clientes são pessoas individuais e empresas de pequeno porte. Recentemente a empresa decidiu informatizar seu cadastro de clientes. O objetivo é construir planos de atendimento, de forma que seja possível criar planos personalizados de prestação de serviços e cobranças de acordo com o perfil do cliente.

Por exemplo: clientes com computadores pessoais para uso doméstico serão atendidos em um plano onde o pagamento ocorre de acordo com a demanda. Assim, o cliente só paga quando requisita o serviço. Esses clientes são atendidos somente em horários comerciais, de segunda à sexta-feira, das 8 às 18 horas. O tempo médio para atendimento desses clientes é de 24 horas. Um serviço solicitado tem até 24 horas para ser realizado.

Já clientes empresariais precisam de um suporte personalizado com atendimento 24 por 7 (todos os dias da semana em qualquer horário). O tempo de resposta é de 4 horas. Esses clientes estão dispostos a pagar planos mensais desde que tenham suporte personalizado e atendimento rápido.

Uma exigência da empresa é que o Software esteja disponível na Web para que os colaboradores possam acessar o cadastro dos clientes em qualquer lugar, a partir de qualquer dispositivo.

Nesse trabalho é executado o início da construção desse Software. Os requisitos do sistema não serão abordados no escopo desse trabalho, mas estarão disponíveis no repositório de Gerência de Configuração. O uso desses artefatos fazem parte do escopo desse trabalho, mas o conteúdo em si não. O foco é o processo de versionamento e entrega desses artefatos.

8.2 Construção e configuração do ambiente de Entrega Contínua

Nessa Seção serão discutidas as etapas desenvolvidas para providenciar a infraestrutura necessária para o processo de Entrega Contínua. A Entrega Contínua será planejada e elaborada para um projeto em fase de concepção. Com isso, o trabalho parte do princípio que a fase de negociação do projeto está concluída. Já foi acordado entre o cliente e o responsável pelo desenvolvimento do software a execução do projeto. O software já foi aprovado e está nas fases iniciais de construção.

8.2.1 Estabelecer um repositorio central para Gerência de Configuração

Conforme dito no Capítulo 3, a gerência de configuração é atividade fundamental na Entrega Contínua. Assim o primeiro passo é estabelecer um repositório de artefatos centralizados para realizar a Gerência de Configuração do projeto. Essa etapa é pensada antes mesmo do levantamento de requisitos. Como requisitos são documentados e aprovados, são artefatos que precisam ser versionados.

Existem várias soluções no mercado que oferecem parcialmente ou completamente os serviços de Gerência de Configuração. Nesse experimento será adotado o **Gitlab** em sua versão *SaaS*[1]. O Gitlab fornece grande parte dos recursos necessários para Gerência de Configuração. Mais detalhes sobre a ferramenta podem ser visto no Apêndice A.2.

A versão *SaaS* foi escolhida pois:

- não será necessário criar uma máquina dentro da infraestrutura para instalar o Gitlab;
- é possível criar ilimitados repositórios públicos e privados gratuitamente;
- é possível fazer Gerenciamento de Mudanças através da ferramenta de *Issue*;
- implementa um excelente controle de acesso, onde é possível determinar qual o nível de acesso que cada pessoa tem em cada projeto;
- oferece ferramentas de mesclagens e recuperações de versões de maneira simplificada;

A escolha do Gitlab deve-se também ao fato de ser um repositório centralizado para projetos versionados pelo Git. O Git é um sistema de versionamento de código distribuído, *Open-Source*, muito robusto e flexível a vários fluxos de trabalho. Mais detalhes sobre o Git no Apêndice A.1.

Nesse experimento foi criado um *Grupo* chamado *ContinuousDeliveryTCC*[2]. Um grupo no Gitlab permite agrupar vários repositórios de um projeto em comum. Outra vantagem de um grupo é que as permissões de acesso de um usuário no grupo, vão refletir em todos os repositórios. Nesse grupo foram criados os repositórios:

- **InfraestruturaCD:**(https://gitlab.com/ContinuousDeliveryTCC/InfraestruturaCD) Nesse repositório estão os *scripts* utilizados para criar e configurar as máquinas

[1] *Saas* é um acrônimo para *Software as a Service*, traduzido como "Software como um Serviço". Nessa modalidade de Software a empresa disponibiliza o serviço na Internet, ficando a cargo dessa empresa toda a infraestrutura de hardware, software, segurança e disponibilidade. O cliente somente se preocupa em utilizar o serviço.

[2] Acesse o grupo no seguinte endereço: https://gitlab.com/groups/ContinuousDeliveryTCC

que vão orquestrar a Entrega Contínua. Os *scripts* criam as máquinas, configuram a rede, instalam os aplicativos e configuram a inicialização dos serviços. Essa infraestrutura é explicada na Seção 8.3

- **SourceCode:**(https://gitlab.com/ContinuousDeliveryTCC/Source-code) Esse é o repositório responsável pelo versionamento do código fonte do software. Esse é o principal repositório do processo de Entrega Contínua, uma vez que os principais gatilhos do processo de entrega vão ocorrer quando os artefatos desse repositório forem criados ou modificados.
- **Documentacoes:**(https://gitlab.com/ContinuousDeliveryTCC/Documentacao) Esse repositório mantém os artefatos de documentação do projeto. Nesse repositório são versionados artefatos como documentação de requisitos, diagramas de arquitetura, UML, processos definidos, contratos, etc.
- **ConfiguracoesCD:** (https://gitlab.com/ContinuousDeliveryTCC/ConfiguracoesCD) O objetivo desse repositório é guardar os artefatos que providenciam as configurações necessárias para execução do Software e os ambientes de Entrega Contínua. Por exemplo: o arquivo **settings.xml** do Maven (detalhes sobre o Maven no Apêndice A.4) que configura os repositórios remotos para busca de bibliotecas de terceiros. Também mantém os arquivos que configuram conexão com o banco de dados nos ambientes de produção e homologação. Quaisquer artefatos que sirvam ao propósito de providenciar configurações no Software serão mantidos nesse repositório.

Nesses repositórios a gerência de configuração é aplicada para evolução do projeto.

8.2.2 Fluxo de trabalho

O gatilho da Entrega Contínua é disparado quando o desenvolvedor submete os artefatos criados ou modificados ao repositório central de versionamento, conforme explicado no Capítulo 5 Pipeline de Entrega Contínua. As submissões no repositório devem ser guiadas por um fluxo de trabalho, conforme explicado no Capítulo 3. O fluxo de trabalho definido para o experimento é mostrado na Figura 8.1.

Os passos desse fluxo de trabalho são:

1. O processo inicia com o colaborador **clonando**[3] a *branch master*[4] para sua má-

[3] *Clone* é o comando utilizado no Git para recuperar do repositório central os artefatos de uma linha de desenvolvimento.

[4] *Branch* é uma linha de desenvolvimento independente, onde qualquer modificação feita em um artefato daquela linha, surte efeito somente nela, sem impactar outras linhas de desenvolvimento em um primeiro momento. No Git a linha de desenvolvimento padrão é chamada de *master*

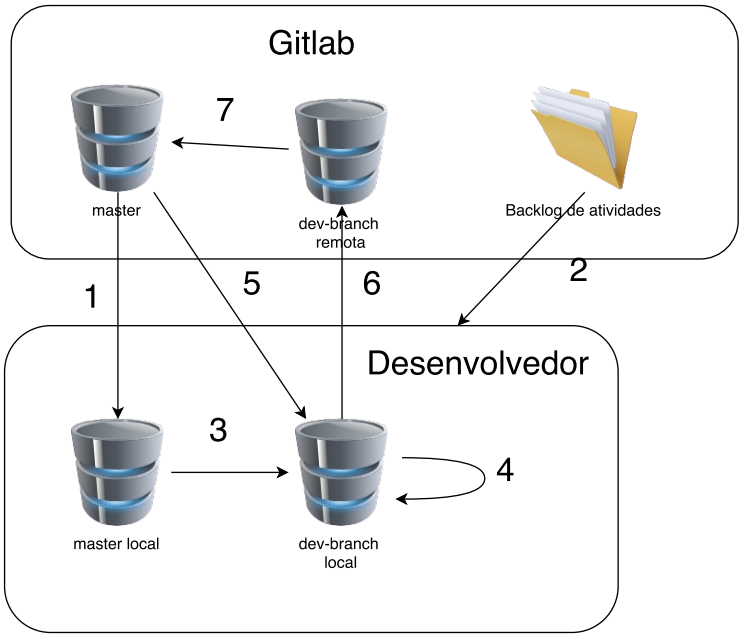

Figura 8.1: *Fluxo de trabalho do experimento*

quina.

2. Uma vez que o colaborador possui as versões mais recentes dos artefatos da linha de desenvolvimento principal em sua máquina, ele deve buscar no *backlog* de atividades[5] uma atividade para realizar no projeto.
3. Para implementar o que foi solicitado na atividade, o desenvolvedor deve criar uma linha de desenvolvimento independente na sua máquina. Essa linha de desenvolvimento será chamada de *dev-branch local*.
4. Nessa linha de desenvolvimento independente o desenvolvedor vai implementar e submeter os artefatos necessários para atender a solicitação da atividade. Esse passo se repete quantas vezes forem necessárias até o desenvolvedor se sentir confiante para submeter o código ao repositório remoto.
5. Quando o desenvolvedor entende que suas modificações estão prontas, antes de submeter o código que ele produziu em sua máquina ao repositório remoto, ele deve verificar se ocorreu modificações na linha de desenvolvimento principal e deve trazê-las para sua linha de desenvolvimento individual, de forma que ele possa resolver os possíveis conflitos que tenham surgido.
6. Resolvido os conflitos, o desenvolvedor então submete sua linha de desenvolvimento individual para o repositório centralizado do controle de versão. Será criado a linha de desenvolvimento *dev-branch remota*.
7. Em seguida esse desenvolvedor solicita um *Merge request*[6] para a *branch mas-*

[5] *Backlog* de atividades são as solicitações de mudança ou registros de novas funcionalidades registradas no Gitlab, na ferramenta de *issues*

[6] *Merge request*, ou requisição de mesclagem é uma funcionalidade disponibilizada pelo Gitlab para

ter. Essa etapa foi elaborada dessa forma, pois no Gitlab os usuários com perfil "desenvolvedor"não tem permissão para commitar diretamente na *branch master*. Somente usuários com o perfil *Owner*[7] ou *master*[8] podem commitar ou aprovar solicitações de *merge* para o fluxo de desenvolvimento príncipal.

Esse é o fluxo de trabalho adotado para o desenvolvimento do Software.

8.3 Infraestrutura de Integração e Entrega Contínua

Estabelecido os repositórios de controle de versão e o fluxo de trabalho, o próximo passo é pensar na Integração Contínua, conforme visto no Capítulo 4. Um ambiente de integração contínua deve ser um ambiente específico para tal [11]. É preciso garantir que o ambiente que irá fazer a integração do projeto continuamente não possua os vícios do ambiente do desenvolvimento. O servidor de Integração Continua também é o orquestrador da Entrega Contínua. Nesse ambiente é centralizado os procedimentos de integração e entrega, como também a geração de *feedback* e gerenciamento de dependências (visto no Capítulo 3). A *pipeline* da Entrega Contínua, explicada no Capítulo 5, será definida nesse ambiente.

8.3.1 Definir Softwares para Integração e Entrega Contínua

Primeiro é preciso definir quais são os softwares que apoiam os processos de integração e entrega. Foi definido que o Jenkins será o orquestrador da Integração e Entrega Contínua. Mais detalhes sobre o Jenkins podem ser vistos no Apêndice A.3. A configuração do Jenkins e seus detalhes técnicos não fazem parte do escopo desse trabalho[9].

Para orquestrar a construção do Software foi escolhido o Maven. Mais detalhes sobre o Maven são encontrados no Apêndice A.4. O Maven será o responsável por manter o *script* de construção do projeto, além de gerenciar as dependências do Software. No Maven são configuradas quais as bibliotecas que o software precisa e quais as versões dessas bibliotecas.

Quando uma dependência é configurada no Maven, no momento da construção do Software, essas dependências são baixadas. Para melhorar o desempenho da busca

integrar o trabalho de um desenvolvedor ao fluxo principal do projeto.

[7] *Owner* é o perfil correspondente a dono do projeto.

[8] *Master* é o perfil para líder de um projeto.

[9] Para facilitar o entendimento de detalhes técnicos, algumas vídeo aulas foram criadas mostrando como instalar e configurar o Jenkins para Integração Contínua. Essas vídeo aulas estão disponibilizadas no Youtube e podem ser acessadas no seguinte link: https://goo.gl/oFhBaJ.

dessas dependência e também para manter as dependências sobre o controle da equipe de desenvolvimento é necessário um repositório de artefatos de dependências. Os módulos produzidos pela equipe, segundo explicado na Seção 4.1.5, serão disponibilizados através desse repositório de dependências. O gerenciador de dependências escolhido foi o Nexus. O Nexus é apresentado no Apêndice A.8.

A linguagem de programação escolhida para desenvolver o Software foi o Java[10]. É preciso testar o código produzido, conforme visto no Capítulo 6 e também na Seção 4.1.3. O Java possui um framework para criação e execução de testes automatizados chamado JUnit. Mais detalhes sobre o JUnit no Apêndice A.5. Será utilizado o JUnit para escrever a suíte de testes da aplicação.

É necessário também manter o versionamento da base de dados da aplicação. A importância desse versionamento foi discutida na Seção 4.1.2. Para gerenciar as migrações da base de dados foi escolhido o framework Flyway, apresentado no Apêndice A.7.

A inspeção do código fonte e sua validação de conformidade, discutidas na Seção 4.1.4, será feita pela ferramenta SonarQube, apresentada no Apêndice A.9. A instalação e configuração de todas as ferramentas utilizadas não fazem parte do escopo desse trabalho. Mas, em sua maioria, as configurações utilizadas nas aplicações são as configurações padrões.

8.3.2 Ambiente de Entrega Contínua

A Entrega Contínua foi determinada de uma maneira bem simples nesse experimento. Como o projeto está em fase de concepção, foi definido que a entrega do Software deveria acontecer em duas máquinas virtuais, apoiadas por uma terceira.

Uma máquina virtual servirá como ambiente de homologação. Nesse ambiente, cada modificação que passar pela *Pipeline* de Entrega será implantada automaticamente. Esse processo é apresentado no Capítulo 7. A implantação contínua pode acontecer nesse ambiente, pois é um ambiente para validação da equipe de testes e usuários finais. O segundo ambiente será o ambiente de produção. Foi definido que esse é o ambiente final de entrega. O software entregue nesse ambiente é o software que será efetivamente utilizado pelo cliente final. A implantação desse ambiente vai ocorrer segundo a estratégia de implantação *Blue-Green*, explicado no Capítulo 5. A terceira máquina é o ambiente de banco de dados da aplicação. Nesse ambiente será disponibilizado o sistema gerenciador de banco de dados e as bases de dados de produção e homologação. Essa abordagem não é a mais aconselhada para projetos reais, mas foi abordada aqui apenas para fins didáticos.

[10]Saiba mais sobre o Java em: https://www.java.com/pt$_BR$/$about$/$whatis_java.jsp$

O ambiente completo é mostrado na Figura 8.2.

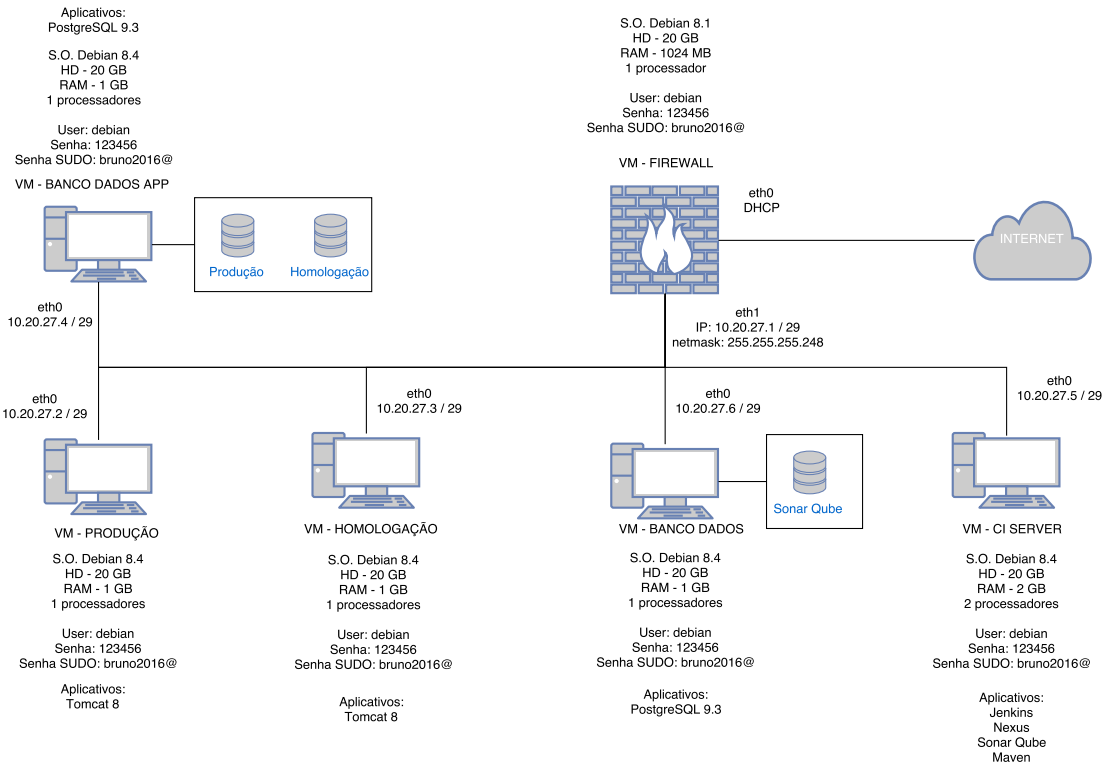

Figura 8.2: *Ambiente completo para Entrega Contínua*

Como o objetivo do trabalho é mostrar que é possível criar um ambiente de Entrega Contínua para um projeto em fase de concepção, foi utilizadas máquinas virtuais em ambiente local para as máquinas que configuram e orquestram a Entrega Contínua. Foi utilizado o gerenciador de máquinas virtuais Oracle Virtual Box[11] na versão 4.3.38. As máquinas virtualizadas utilizam o Sistema Operacional Debian[12] na versão 8.1. Foi construída as seguintes máquinas:

- **VM - FIREWALL** - Essa máquina será a roteadora da rede de Integração e Entrega Contínua. Essa máquina foi escolhida para ser o ponto central de acesso à todos os ambientes internos, tanto para proteção desses ambientes (não permitir acessos indevidos), como para centralizar acesso aos recursos internos do ambiente de Integração e Entrega Contínua.
- **VM - CI SERVER** - Esse é o ambiente que vai orquestrar a Integração e Entrega Contínua. A maior parte das ferramentas apresentadas até o momento estará disponbilizada nessa máquina.
- **VM - BANCO DADOS** - Essa máquina foi disponbilizada apenas para manter a base de dados utilizada pelo SonarQube. O SonarQube faz várias validações e

[11] Saiba mais em: https://www.virtualbox.org/
[12] Disponível em: https://www.debian.org/index.pt.html

executa várias requisições nesse banco e isso gera uma sobrecarga na máquina virtual. Por isso foi disponibilizado um ambiente específico para o banco de dados do SonarQube.

- **VM - HOMOLOGACAO** - Essa máquina será utilizada para testes. Toda modificação que disparar a Pipeline de Entrega concluirá na entrega automatizada do software produzido nesse ambiente.
- **VM - PRODUCAO** - Esse é o ambiente final da Entrega. Trata-se do ambiente de produção, onde o software será entregue para o cliente final.
- **VM - BANCO DADOS APP** - Esse ambiente mantém as bases de dados utilizadas pelo ambiente de produção e homologação.

O procedimento completo para criação e configuração dessas máquinas é descrito no repositório *InfraestruturaCD*[13]. Para que esses ambientes sejam provisionados, basta descompactar os arquivos do repositório que contém a máquina virtual já criada. Será gerada uma *appliance* do Virtual Box. Esse arquivo deve ser importado e então a máquina estará pronta para uso. Em seguida deve-se iniciar a máquina, com a rede configurada para acesso à Internet. Depois é preciso fazer o download do arquivo "orquestrador.sh"referente ao ambiente que se deseja montar. Esse arquivo deve ser executado pelo usuário **root** da máquina e então o procedimento de instalação dos aplicativos e configuração na máquina será feito automaticamente.

Essa abordagem para provisionamento de máquinas de maneira manual, apesar de não ser completamente incorreta, deve ser evitada. Existem serviços que provisionam a máquina de maneira automática. Outros serviços criam contêiners de configuração das máquinas e as disponibilizam para uso. Para não haver um escopo muito amplo, esse trabalho não aborda o uso dessas ferramentas. Essas abordagens são indicadas como fontes para trabalhos futuros.

8.3.3 Configurando os aplicativos da Integração Contínua

Com a infraestrutura montada é preciso configurar os aplicativos que gerenciam a Entrega Contínua. Iniciando pela Integração Contínua o primeiro aplicativo configurado é o Jenkins. A primeira vez que o Jenkins é acessado ele acompanha o usuário no processo de configuração. É configurado o primeiro usuário do Jenkins com perfil administrador e os plugins iniciais que o usuário deseja utilizar. Após esse procedimento o Jenkins estará apto para uso.

Em seguida é preciso realizar as configurações gerais do Jenkins. Essas configurações vão variar de acordo com o projeto que está sendo criado. Será necessário definir

[13] https://gitlab.com/ContinuousDeliveryTCC/InfraestruturaCD

a linguagem de programação utilizada e o seu compilador, quais os frameworks escolhidos, entre outras configurações diversas. Para esse projeto foi configurado, inicialmente, o kit de desenvolvimento Java (JDK) e o Maven para gerenciar o script de construção do projeto.

Para garantir que as configurações feitas estão funcionais, uma boa prática é criar um pequeno projeto para servir de testes das configurações. No cenário desse projeto foi criado o projeto raiz do Maven que vai centralizar as principais configurações para os demais projetos. Ele foi nomeado como **Projeto - ROOT**. Ele é o projeto pai dos demais artefatos que serão criados no decorrer do projeto. Foi criado outro projeto chamado **Hello World**. Esse projeto é um módulo do projeto raiz. Ele possui apenas duas classes: uma que contém o código de teste e a classe de produção que implementa a funcionalidade testada.

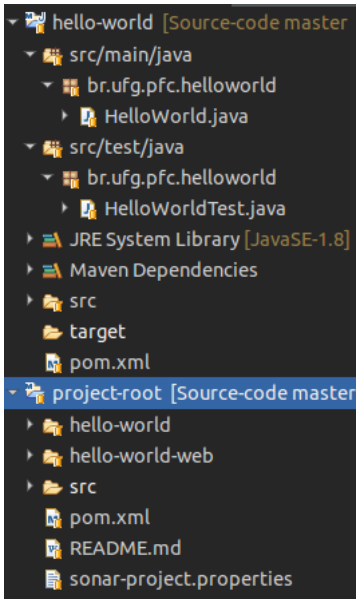

Figura 8.3: *Estrutura de código com Hello World para testes*

Com a conclusão dessa configuração o próximo passo foi submeter esse código fonte para análise de conformidade, conforme discutido na Seção 4.1.4. O Jenkins é capaz de submeter o código a análise de conformidade do Sonar. Para que funcione, primeiro é preciso instalar o *plugin* do Sonar no Jenkins. Com o *plugin* instalado, deve-se configurar os dados de conexão com o Sonar no Jenkins. Após a configuração, no *job* de testes do Jenkins, foi adicionada uma chamada para análise de conformidade do Sonar. Foi necessário criar um arquivo chamado sonar-project.properties no projeto raiz para que a analise ocorresse com sucesso.

Após a configuração com o Sonar, o próximo passo foi publicar os artefatos produzidos no Sonatype Nexus OSS. Para publicar os pacotes produzidos foram utilizados os recursos do Maven. Esse recurso é ativado através da configuração do arquivo *settings.xml*. Nele foi criado um perfil que configura os dados de conexão com o servidor

onde foi instalado o Nexus, conforme apresentado no código 8.5. Foi configurado também o usuário e senha de acesso ao servidor (conforme código 8.2), além do espelho onde os artefatos gerenciados pelo Nexus podem ser baixados para serem utilizados no projeto (conforme código 8.3). Também foi configurado no arquivo *pom.xml* do projeto raiz a propriedade *distribuitionManagement* que configura os os dados de conexão com o repositório central e de *snapshots* do Nexus. Veja o código 8.4 que mostra o trecho que código necessário acrescentar no arquivo. Dessa forma, as dependências do projeto estarão centralizadas em um repositório central dentro da rede do projeto, centralizando os artefatos e os tornando acessíveis para a equipe.

Código 8.1 `profiles do settings.xml`

```
<profiles>
  <profile>
    <id>nexus</id>
    <properties>
      <releaseRepoName>release</releaseRepoName>
      <releaseRepoUrl>
        http://ci.pfc2.si.ufg.br:8081/repository/maven-releases/
      </releaseRepoUrl>
      <snapshotRepoName>snapshot</snapshotRepoName>
      <snapshotRepoUrl>
        http://ci.pfc2.si.ufg.br:8081/repository/maven-snapshots/
      </snapshotRepoUrl>
    </properties>
    <repositories>
      <repository>
        <id>central</id>
        <url>http://central</url>
        <releases>
          <enabled>true</enabled>
        </releases>
        <snapshots>
          <enabled>true</enabled>
        </snapshots>
      </repository>
    </repositories>
    <pluginRepositories>
      <pluginRepository>
        <id>central</id>
        <url>http://central</url>
        <releases>
          <enabled>true</enabled>
        </releases>
        <snapshots>
          <enabled>true</enabled>
        </snapshots>
      </pluginRepository>
    </pluginRepositories>
  </profile>
</profiles>
```

Código 8.2 `servers do settings.xml`

```
<servers>
  <server>
    <id>central</id>
    <password>jenkins123!</password>
    <username>jenkins</username>
  </server>
  <server>
    <id>snapshots</id>
    <password>jenkins123!</password>
    <username>jenkins</username>
  </server>
</servers>
```

Código 8.3 `mirrors do settings.xml`

```
<mirrors>
  <mirror>
    <id>nexus</id>
    <mirrorOf>external:*</mirrorOf>
    <url>http://ci.pfc2.si.ufg.br:8081/repository/maven-public/</url>
  </mirror>
</mirrors>
```

Código 8.4 `distributionManagement do pom.xml da raiz do projeto`

```
<distributionManagement>
        <repository>
                <id>central</id>
                <name>${releaseRepoName}</name>
                <url>${releaseRepoUrl}</url>
        </repository>
        <snapshotRepository>
                <id>snapshots</id>
                <name>${snapshotRepoName}</name>
                <url>${snapshotRepoUrl}</url>
                <uniqueVersion>true</uniqueVersion>
        </snapshotRepository>
</distributionManagement>
```

Esses arquivos estão disponíveis completos em:

- https://gitlab.com/ContinuousDeliveryTCC/Source-code/blob/master/pom.xml
- https://gitlab.com/ContinuousDeliveryTCC/ConfiguracoesCD/blob/master/maven/settings.xml

8.3.4 Configurando a Implantação Contínua no ambiente de Homologação

Para esse experimento, o ambiente de homologação é implantado continuamente. Cada *commit* realizado no projeto gerará o artefato final do Software. O produto final do Software é um arquivo Java Web (WAR) que é implantado no *container web* do Tomcat[14] na versão 8.

Para testar a implantação, foi criado um projeto Web que contém apenas uma página inicial. É o projeto Hello World Web. Com o projeto criado, o Jenkins foi configurado para implantar esse artefato automaticamente no ambiente de homologação. Essa configuração consiste em criar um usuário para o Tomcat que tenha permissão de deploy. Para configurar o usuário foi utilizado o seguinte trecho de código apresentado em 8.5 no arquivo *tomcat-users.xml* da pasta *conf* no diretório de instalação do Tomcat.

Código 8.5 `tomcat-users.xml`

```
1 <?xml version='1.0' encoding='utf-8'?>
2 <tomcat-users>
3   <role rolename="manager-script"/>
4   <user username="deployment" password="deployment123@" roles="manager-script"/>
5 </tomcat-users>
```

Esse usuário é configurado no Jenkins para, sempre que o arquivo WAR for gerado, ele deverá ser lançado no Tomcat. Foi instalado o plugin *Deploy to Websphere container Plugin*[15] no Jenkins. Esse é o plugin responsável por fazer o deploy automático do projeto Java Web no container do Tomcat.

8.3.5 Configurando a Entrega Contínua no ambiente de Produção

A fase final é escolher a estratégia de implantação no ambiente de produção. As estratégias de implantação em produção são discutidas no Capítulo 5. Das várias possibilidades, foi escolhido para esse trabalho a estratégia de entrega *Blue-Green*.

O ambiente de homologação é atualizado constantemente. Cada modificação que ocorre nos artefatos do projeto, gera, no final, o software que é automaticamente

[14] Tomcat um container de servlets escrito em linguagem Java. É utilizado para publicações de aplicações WEB. Conheça mais sobre o Tomcat no link: http://tomcat.apache.org/

[15] Veja detalhes desse plugin em http://wiki.hudson-ci.org/display/HUDSON/Deploy+WebSphere+Plugin

implantado no ambiente de homologação. Um Job no Jenkins é responsável por entregar o projeto em produção. Esse Job é disparado manualmente. Quando esse Job é disparado ele conecta na máquina de **FIREWALL** via SSH[16] e modifica as rotas configuradas na tabela de IPs *iptables*[17]. Resumidamente, o Job modifica as máquinas fazendo com que a máquina de homologação passe a responder como produção e vice-versa.

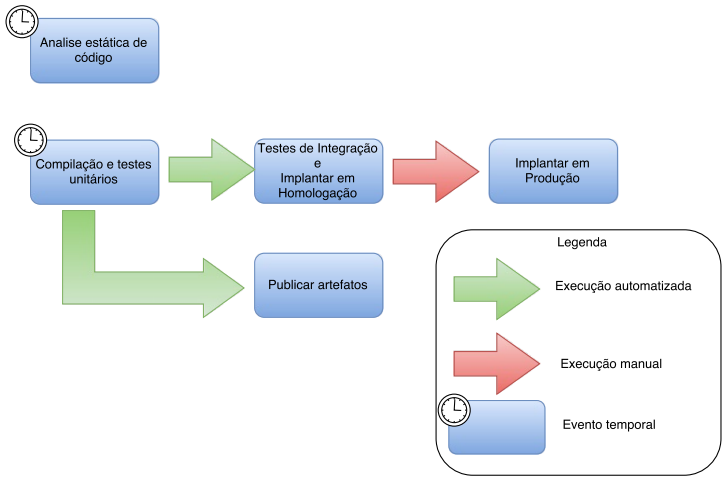

Figura 8.4: *Pipeline de Entrega Final do Trabalho*

As configurações apresentadas nas seções anteriores objetivaram tornar o Jenkins apto a executar todas as fases do ciclo de vida de Entrega. Por fim é necessário definir a *pipeline* de Entrega. A pipeline foi dividida em 5 etapas. A Figura 8.4 apresenta como essas etapas, e a relação entre elas, foram configuradas.

Foram definidas duas etapas temporais: "Compilação e testes unitários"e "Analise estática de código". A etapa de **compilação e testes unitários** é um Job no Jenkins que executa a cada minuto. Esse Job conecta no repositório de código fonte no Gitlab e verifica se ocorreu mudanças na linha principal de desenvolvimento do projeto. Se mudanças forem detectadas, esse Job é disparado. Na sua execução ele compila o código fonte do projeto e executa os testes unitários. Em seguida ele empacota a aplicação e arquiva os artefatos produzidos na área de trabalho do Jenkins. Por fim ele publica o relatório de execução dos testes no Job, possibilitando o acompanhamento da evolução dos testes e suas execuções.

A maneira mais indicada para disparar esse Job é através do evento de submissão de modificação do Gitlab. Cada vez que ocorrer modificações no repositório central, o Gitlab deve ser capaz de informar o evento para o servidor de Integração Contínua. Somente então o servidor de Integração Contínua deveria buscar as modificações e

[16]SSH é um aplicativo e um protocolo de rede que permite a conexão e execução de comandos em computadores remotos.

[17] O *Iptables* é uma ferramenta para criar e administrar regras e assim filtrar pacotes de redes. Veja mais em http://www.netfilter.org/projects/iptables/

executar o seu ciclo de integração e entrega. Essa abordagem é a ideal, porém não foi viável no projeto pois as máquinas virtuais criadas não possuiam um endereço fixo na Internet e estavam constantemente mudando de IP. Portanto, não foi possível configurar esse evento no Gitlab. A abordagem de verificação a cada minuto no Gitlab não é incorreta, contudo, dependendo do tamanho do projeto, essa fase passará a consumir muito tempo e recurso do servidor de Integração Contínua.

Ao final do processo de "Compilação e testes unitários"dois outros Jobs são executados: "Deploy em Homologação"e "Publicar aplicativo". Eles serão explicados posteriormente.

Já a "análise estática de código"é um Job configurado para executar todos os dias à meia noite. Esse Job vai baixar o código fonte do repositório do Gitlab e executar a análise de conformidade do código. No Jenkins foi configurado um plugin que executa a verificação via Sonar de maneira automatizada. Ao final da execução, as métricas do código ficam disponíveis através da interface do portal Web do Sonar. As inconformidades encontradas no código também são disponibilizadas nesse portal. Contudo, não foi implementada nenhuma verificação mais elaborada nessa etapa de execução. A construção do projeto não é impactada caso alguma regra de verificação do Sonar tenha sido violada.

Conforme dito, o Job de compilação e testes unitários chama dois outros Jobs no Jenkins. O Job "Publicar aplicativo"vai pegar os artefatos produzidos pelo Job de compilação e enviar para o Nexus. No Nexus, os artefatos serão disponibilizados para toda a equipe de desenvolvimento. O outro Job chamado é o job "Deploy em Homologação". Esse job é responsável por executar os testes de integração do software e ao final publicar o artefato web produzido. Essa publicação é feita através do *Hot Deploy*[18] do Tomcat.

A última etapa do processo é uma chamada manual ao Job "Implantar em Produção". A Figura 8.5 mostra o estado inicial do ambiente antes da execução do Job. Esse Job é executado manualmente recebendo um parâmetro chamado "VERSAO". Esse Job vai executar todo o procedimento de implantação em produção. Os passos desse job são:

- Baixar o código fonte: nessa etapa, o estado atual dos artefatos da linha de desenvolvimento principal da aplicação é clonado do Gitlab.
- Apontar nova versão: o parâmetro VERSAO recebido pelo usuário que dispara o Job é utilizado para determinar a nova versão que será desenvolvida. Para isso o

[18] *Hot Deploy* é uma funcionalidade que permite que a atualização seja atualizada sem a necessidade de parar o serviço do Container Tomcat, transferir o pacote da aplicação Web e iniciar novamente o Tomcat. Através do *Hot Deploy* a aplicação pode ser atualizada enquanto está em execução

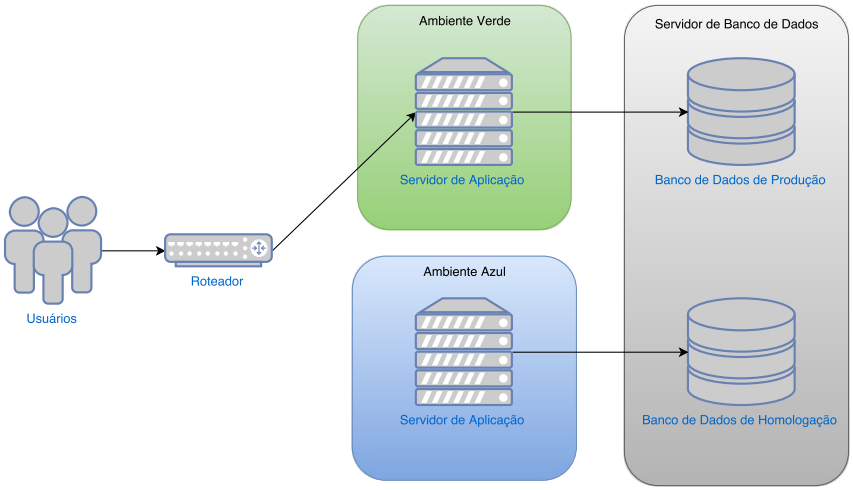

Figura 8.5: *Ambientes antes da Implantação em Produção*

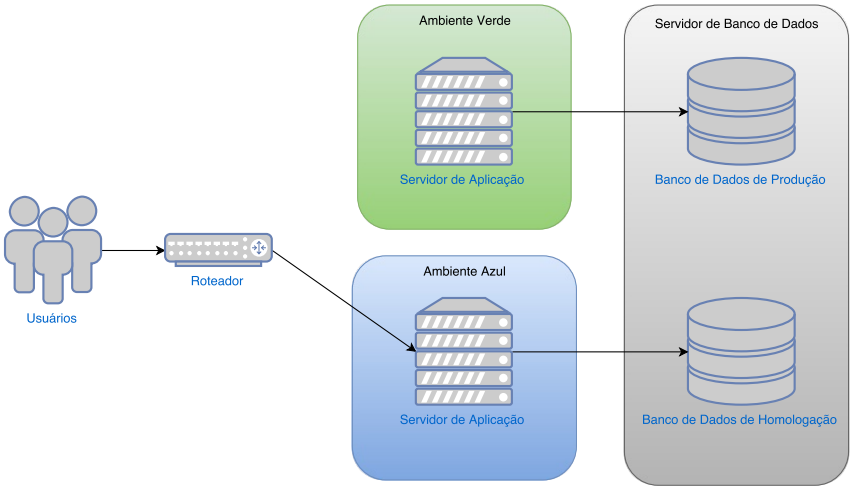

Figura 8.6: *Redirecionando usuários para novo ambiente de Produção*

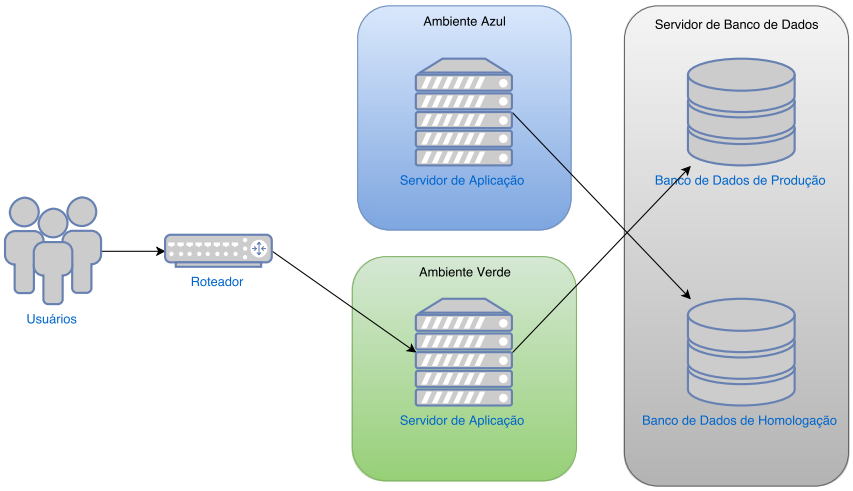

Figura 8.7: *Reapontando bases de dados. Nova configuração do ambiente*

Maven é configurado com um plugin para atualizar a versão dos projetos Maven. Esse plugin seta como nova versão o valor recebido através do parâmetro do Job.

- Atualização do Firewall: em seguida o Jenkins conecta na máquina de Firewall via conexão SSH e modifica a tabela Iptables. A Figura 8.6 resume essa modificação. Essa tabela é inicialmente configurada de forma que as requisições enviadas para a porta 80 do Firewall sejam redirecionadas para a porta 8080 da máquina 10.20.27.2. Tal máquina responde inicialmente como servidor de produção. Já as requisições enviadas para a porta 81 do Firewall, redireciona as requisições para a porta 8080 da máquina 10.20.27.3, a máquina de homologação. A execução dessa etapa da entrega faz uma troca nessas portas. A porta 80 passa a redirecionar para a máquina 10.20.27.3, e a porta 81 passa a redirecionar as requisições para a máquina 10.20.27.2. Isso troca a máquina de produção pela máquina de homologação.
- Reapontamento dos bancos de dados: após a troca de máquinas, é necessário atualizar a configuração dos bancos. As máquinas de produção e homologação apontam para a mesma máquina de banco de dados, porém existem esquemas de bancos de dados diferentes nessa máquina, um para produção e outro para homologação. Para configurar a conexão com o banco de dados da aplicação foi criado um arquivo chamado *db.hikari.properties* que é salvo na pasta de usuário do Tomcat. Esse arquivo mantém os dados de conexão com o banco de dados utilizados pela aplicação naquele ambiente. Quando os ambientes são modificados, é preciso atualizar, nesse arquivo, o apontamento dos esquemas do banco de dados. Assim, o arquivo que está na máquina 10.20.27.3, que aponta para o esquema de homologação, deve passar a apontar para o esquema de produção. O inverso acontece no ambiente 10.20.27.2. O Job então se conecta nas máquinas, para o serviço do Tomcat em ambas, realiza a edição do arquivo de configuração de conexão com o banco e depois inicia os serviços. A Figura 8.7 apresenta a configuração final do ambiente.
- Criação da tag no Gitlab: por fim a versão recebida como parâmetro é utilizada para criar uma tag no Gitlab. Essa tag serve para marcar o estado do código fonte no momento da virada para produção.

8.4 Desenvolvimento da aplicação

Realizadas as configurações do ambiente de Integração e Entrega Contínua de Software, foi iniciado o desenvolvimento da aplicação solicitada. A equipe de desenvolvimento foi formada por dois desenvolvedores, incluíndo o autor. A etapa de desenvolvimento foi realizada durante três dias.

No primeiro dia foi alinhado com os desenvolvedores como funcionava o pro-

cesso de Entrega Contínua, tal como seria os processos de desenvolvimento, explicado na Seção 8.2.2. Depois, foi elaborada a documentação do projeto do Software. Foi definida apenas uma documentação básica do Software, que contém os seguintes artefatos:

- **Diagrama de caso de uso:** esse diagrama possui dois casos de uso: "Cadastrar cliente"e "Pesquisar Cliente";
- **Diagrama de classes de entidades de domínio:** esse diagrama possui as entidades de domínio da aplicação, que consiste basicamente na entidade "Cliente";
- **Diagrama de atividade:** para cada caso de uso foi criado um diagrama de atividade que determina como será o fluxo de execução daquele caso de uso;
- **Diagrama de Componentes:** esse diagrama determina os componentes da aplicação, bem como seu relacionamento para elaboração da arquitetura do Software. Foram criados alguns diagramas de classe, um para cada componente, para determinar as interfaces dos serviços que serão oferecidos por cada camada de serviços do Software.

Essa documentação foi disponibilizada no repositório de documentações no Gitlab.

No segundo dia ocorreu efetivamente o desenvolvimento dos módulos. Cada desenvolvedor implementou um módulo de acordo com a documentação disponibilizada. Seguindo o fluxo de trabalho definido, cada desenvolvedor criava uma atividade no Gitlab informando qual o módulo irá desenvolver. Então o desenvolvedor codificava esse módulo, seguindo a prática de *Test Driven Development*. Um teste automatizado deveria ser criado utilizando o JUnit como *framework* para escrita de testes automatizados. Quando o teste estava implementado o desenvolvedor codificava a solução. Com a solução implementada, ele rodava todos os testes em sua máquina local. Com todos os teste passando na fase de testes o código produzido era submetido ao Gitlab através de uma *branch*. O outro desenvolvedor revisava o código produzido. Em seguida era feito um *Merge Request* para a *branch master*. Com o *merge* feito, o processo de Integração Contínua e Implantação Contínua no ambiente de testes era executado. Cada vez que a implantação ocorria com sucesso era possível visualizar as modificações produzidas no ambiente de homologação. Durante as submissões diárias somente os Jobs de Integração Contínua e Implantação Contínua eram executados. À meia-noite um Job no Jenkins submetia o trabalho do dia para análise do Sonar a fim de detectar os problemas de codificação encontrados. As figuras 8.8 e 8.9 mostram o resultado da análise do código da aplicação pronto. Quanto a estratégia de migração da base de dados da aplicação, foi usado o Flyway para gerenciar as migrações. Para tal, foi implementado para que a aplicação, ao ser inicializada, chamasse as funcionalidades do FlyWay para conectar no banco de dados e atualizar as versões do esquema.

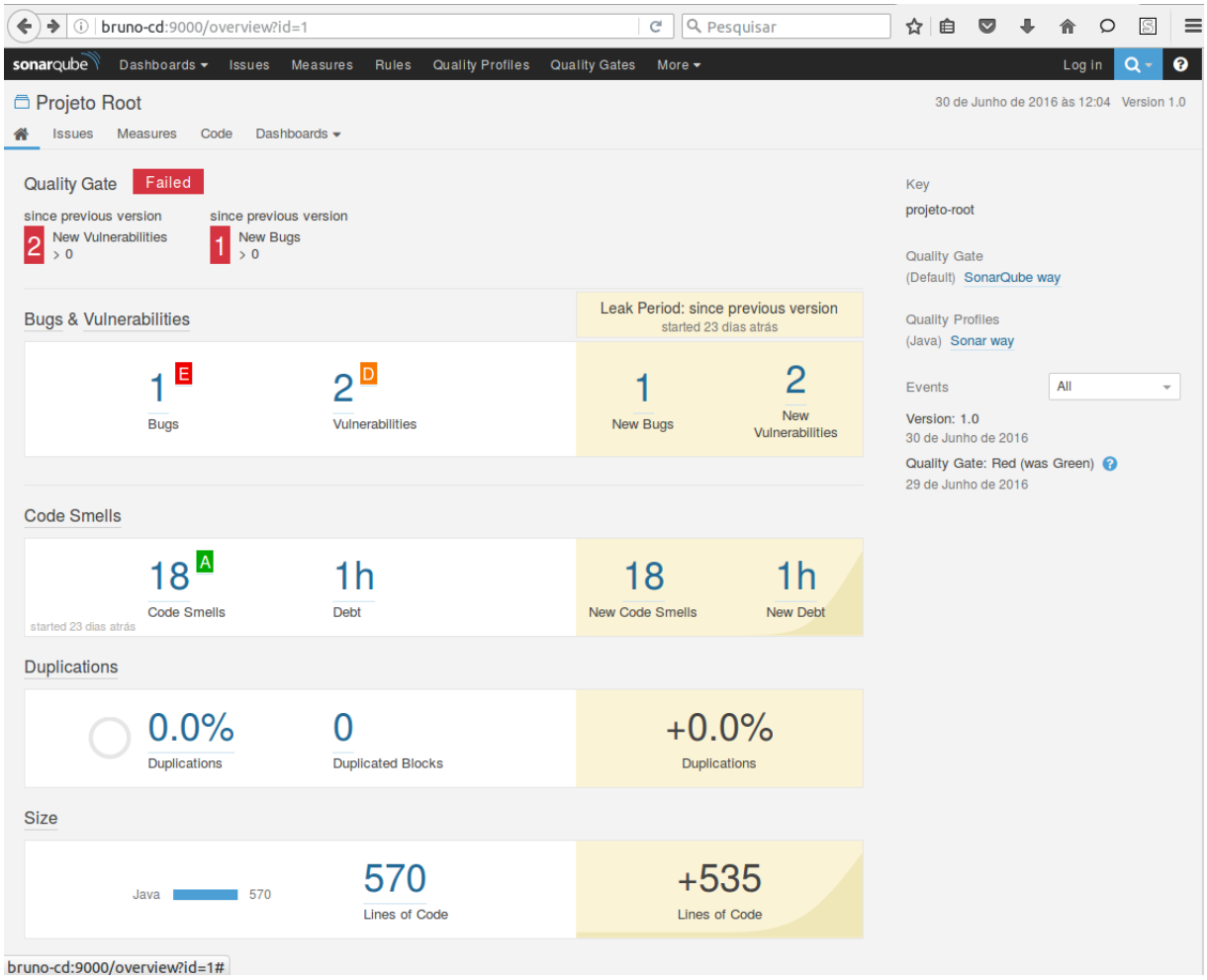

Figura 8.8: *Resultado da análise do Sonar*

No último dia de desenvolvimento os módulos foram concluídos e uma força tarefa foi feita para garantir que todos os módulos se comunicavam corretamente. Os testes foram executados no ambiente de homologação a fim de verificar se as funcionalidades produzidas estavam efetivamente funcionando. Alguns erros foram encontrados. Cada erro encontrado era corrigido e submetido ao repositório de artefatos, tal como definido no processo de integração e implantação em homologação. Ao final do terceiro dia foi definido que o software estava passível de Entrega em produção, então a virada para produção ocorreu chamando o Job "Implantar em Produção".

8.5 Resultados do experimento

Após todo o procedimento de criação e configuração dos ambientes e desenvolvimento da aplicação, alguns resultados foram percebidos. A proposta desse trabalho foi verificar se é viável a construção e configuração de um ambiente para Entrega Contínua desde o início do projeto, antes mesmo da primeira linha de código fonte ser escrita. O

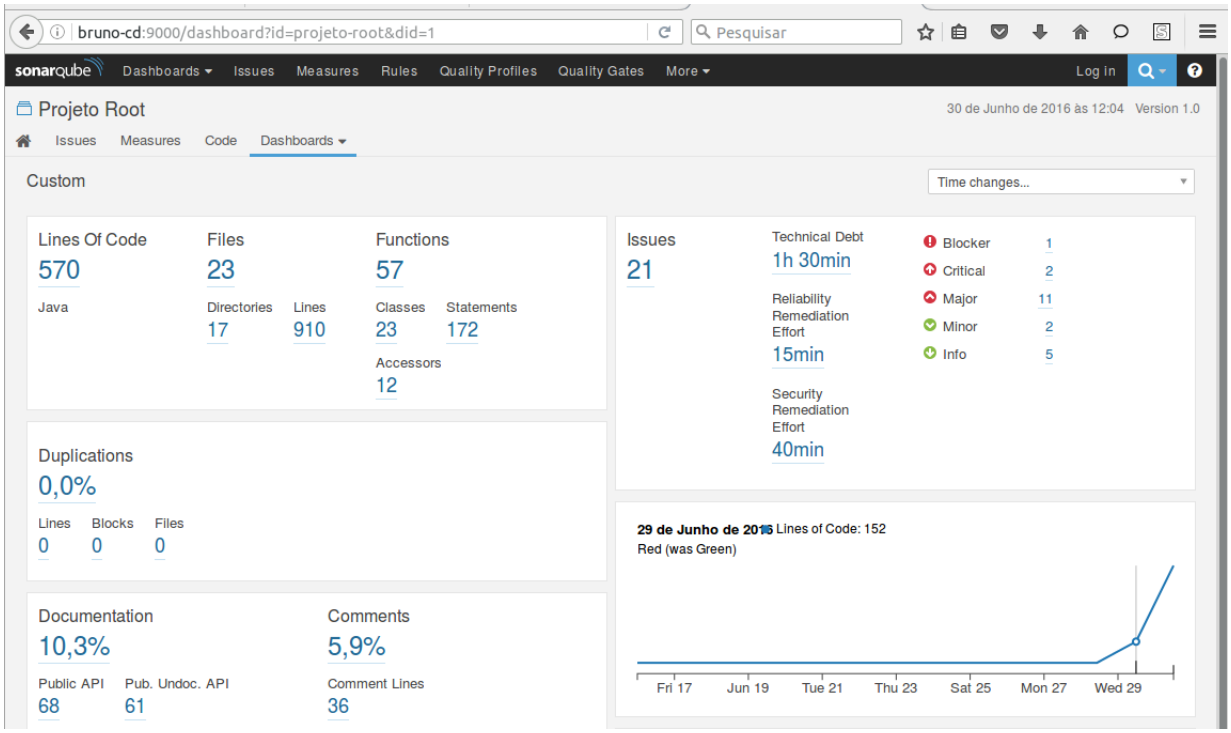

Figura 8.9: *Outro dashboard da análise do Sonar*

experimento demonstrou que é preciso considerar três fatores para essa viabilidade: conhecimento, tempo e recursos.

O primeiro fator é demonstrado através da parte teórica do trabalho. É necessário possuir uma série de conhecimentos para se pensar em Entrega Contínua. Conceitos como Gerência de Configuração, automatização de testes, pipeline de entrega, arquitetura de projeto, frameworks para auxiliar o desenvolvimento, ferramentas de Entrega Contínua, maturidade em processos de negócio e desenvolvimento, além do conhecimento sobre as caracteristicas do ambiente do cliente onde a aplicação será executada, entre outros. Por isso é necessário que, ao propor entregar continuamente um projeto, a equipe possua profissionais capacitados a trabalhar com toda essa estrutura de conhecimentos. Entregar continuamente diz respeito principalmente ao alinhamento das expectativas e conhecimentos entre todos os envolvidos, desde *stackholders*, gerência até a equipe de desenvolvimento.

Já os recursos são influênciados por dois fatores. Um fator é a maturidade da equipe. Uma equipe com profissionais capacitados a trabalhar com entrega contínua é cara. Os profissionais com o conhecimento necessário para essa equipe pode despender boa parte dos recursos financeiros destinados ao projeto. Em contrapartida a contratação de profissionais menos experientes pode consumir recursos com treinamentos, podendo impactar o tempo de entrega do projeto. Outro fator que influência nos recursos é a infraestrutura necessária para os ambientes de Integração e Entrega contínua. Nesse experimento foi necessária uma máquina exclusivamente dedicada ao processo de Entrega Contínua.

Nessa máquina foi virtualizada seis máquinas para comportar os aplicativos e ambientes de entrega. Como o projeto desenvolvido foi pequeno e as aplicações configuradas utilizaram praticamente as configurações padrões, pouco recurso computacional foi consumido. Porém, projetos maiores necessitam de mais recursos computacionais e dependendo da caracteristica do ambiente do cliente, podem ser necessárias muitas máquinas para realizar o processo de entrega. Existe no mercado empresas que trabalham com virtualização de máquinas e disponibilização de infraestrutura. Um exemplo é a Amazon WS[19]. Os custos de criação e manutenção de servidores virtuais são mais baratos do que a manutenção de uma estrutura física própria, porém, ainda assim, recursos financeiros do projeto são consumidos pelo uso desses serviços.

Por fim o tempo do projeto também deve ser levado em consideração. Estabelecer um ambiente de Entrega Contínua requer tempo. Muitas configurações são necessárias. Mesmo com vários aplicativos oferecendo várias facilidades, é necessário algum tempo para configurá-los e deixá-los alinhados com a necessidade do projeto. Como o mercado é dinâmico e a necessidade de responder a desafios rapidamente e com qualidade é constante, dependendo do projeto a ser desenvolvido, o tempo pode ser escaço. O tempo para se contruir o ambiente deve ser levado em consideração. Em contrapartida, à partir do momento que o ambiente de Entrega Contínua estiver estabelecido, o tempo de entrega das funcionalidades e correção de problemas será reduzido. Portanto deve-se pesar o que é mais importante:

- postergar o trabalho da Entrega Contínua para economizar tempo no início do projeto e levar mais tempo para entregar as funcionalidades para o cliente,
- ou investir em Entrega Contínua desde o início do projeto, ainda que isso atrase um pouco o início do desenvolvimento da aplicação, para futuramente esse tempo ser revertido em agilidade na entrega de demandas e solução de problemas.

Essa não é uma pergunta simples de responder e requer um profissional capacitado e com experiência em Entrega Contínua para tomar a decisão de qual caminho tomar.

Com base nesses fatores, foi possível concluir que pensar em Entrega Contínua desde o início do desenvolvimento pode ser interessante para alguns projetos enquanto para outros é aconselhável postegar essa prática para o futuro. As conclusões à respeito desses fatores serão abordadas no próximo capítulo, na Seção 9.2.

[19] Veja mais detalhes em: https://aws.amazon.com/pt/?nc2=h$_lg$

CAPÍTULO 9

Conclusão

9.1 Propostas de trabalhos futuros

Esse trabalho buscou produzir dois artefatos:

1. O conhecimento mínimo necessário para se iniciar a implantação de um ambiente de Entrega Contínua para um projeto que ainda será desenvolvido;
2. Um repositório de artefatos com scripts que viabilizam automaticamente grande parte da infraestrutura necessária para se aplicar a Entrega Contínua;

A Entrega Contínua é um assunto amplo. Nesse trabalho foram abordadas somente as características elementares do método. O foco desse trabalho foi o desenvolvimento de novos softwares e não a manutenção de softwares que já estão em produção. Portanto, mesmo que se estabeleça um ambiente básico para Entrega Contínua antes do desenvolvimento do projeto, o próprio ambiente de Entrega Contínua deve evoluir com o projeto. Enquanto o software está sendo codificado, oportunidades de utilizar novas ferramentas no ambiente de Entrega Contínua, ou mesmo modificações nas configurações já estabelecidas, serão necessárias. Várias propostas de trabalhos futuros podem ser definidas para esta evolução.

Com o que foi abordado nesse trabalho, algumas propostas são latentes:

- **Provisionamento automatizado de máquinas:** abordar as ferramentas e métodos para se criar e provisionar máquinas automaticamente, sem necessidade de intervenção manual. Existe no mercado aplicações que provisionam máquinas virtuais de maneira automatica, inclusive disponibilizando essas máquinas nos principais serviços de disponibilização de infraestrutura do mercado. Soluções como o Docker[1] provisionam máquinas em containers, tornando fácil o trabalho de provisionar novos ambientes para execução da aplicação. Essa abordagem é uma evolução na forma como as máquinas foram criadas e configuradas nesse ambiente;

[1] Veja mais em: https://www.docker.com

- **Uso de recursos nativos de integração contínua nos serviços *Saas* de Gerência de Configuração:** O Gitlab oferece recursos natívos que fazem a Integração Contínua do projeto dentro da sua própria infraestrutura. O uso desse recurso, já integrado no sistema de Controle de Versão, pode diminuir a complexidade de instalar, configurar e manter uma máquina local para execução desses processos. O Github[2] também disponibiliza diversas ferramentas com integração nativa com os projetos mantidos por ele. Isso facilita e diminui a complexidade de um ambiente de Integração e Entrega Contínua.
- **Construir o ambiente de Entrega Contínua segundo um modelo de maturidade:** foi apresentado nos trabalhos correlatos as atividades para se construir um software para Entrega Contínua através dos modelos de maturidade. Uma proposta de trabalho futuro é abordar o modelo de maturidade para Entrega Contínua ao se construir um Software novo.
- **Como aplicar Entrega Contínua para aplicações Mobile?** O software apresentado nesse trabalho é um Software de computadores típico, onde os recursos disponíveis são relevantes e o programa está disponível em um ambiente. Softwares com essa caracteristica não costumam se espalhar indefinidamente. É o cenário oposto às aplicações móveis. Aplicações para dispositivos móveis, como *smartphones* tornaram-se o foco de muitas empresas e desenvolvedores que querem espandir seus serviços. Uma proposta de trabalho futuro é como a Entrega Contínua pode ser aplicada para softwares de dispositivos móveis, onde o alcance da aplicação pode ser imprevisível.

Os artefatos produzidos nesse trabalho foram disponibilizados publicamente no Gitlab, no seguinte endereço: https://gitlab.com/groups/ContinuousDeliveryTCC. O trabalho está aberto à comunidade para ser analisado e melhorado. Todas as contribuições serão bem-vindas.

9.2 Considerações finais

Pensar em Entrega Continua do Software desde o início do projeto é um desafio que requer conhecimento e maturidade. Nesse trabalho foi possível verificar que construir um ambiente para Entrega Contínua é um trabalho que requer tempo, prática, conhecimento e recursos. Existem muitas ferramentas no mercado que auxiliam as diversas etapas da construção e execução da Entrega Contínua, mas, ainda assim, é preciso de maturidade para lhe dar com toda a complexidade e escolher as ferramentas corretas.

[2] Veja mais em: https://github.com/integrations

Quais são os cenários onde pensar em Entrega Contínua desde o princípio é interessante? Em geral são os cenários onde o software já tem clientes dispostos a participar do projeto e a equipe de desenvolvimento possui profissionais com maturidade para desenvolver para Entrega Contínua. Empresas que já estão estabelecidas no mercado e que querem desenvolver novos produtos podem pensar em iniciar seus novos projetos com foco em Entrega Contínua. *Software houses* são exemplos dessas empresas. Por já possuirem uma carteira de clientes e profisssionais capacitados à trabalhar com as melhores práticas da Engenharia de Software, é possível produzir novos projetos com foco em Entrega Contínua. Considerando os trabalhos de Steve Neely e Steve Stolt, *Continuous Delivery? Easy! Just Change Everything (well, maybe it is not that easy)*[18] e Lianping Chen, *Towards Architecting for Continuous Delivery*, é possível percerber que haverá muitos ganhos na adoção dessa prática, sendo o principal deles a proximidade que os usuários do sistema terão com a equipe de desenvolvimento.

Outro cenário positivo é a reconstrução de Softwares legados. Existem softwares críticos que funcionam atualmente, porém foram codificados em linguagens antigas, o que torna difícil encontrar profissionais para dar manutenção, ou mesmo estender as funcionalidades do mesmo. A falta de documentação é outro fator que faz com que a manutenção se torne inviável com o passar do tempo. Algumas empresas optam por adotar a Engenharia Reversa, mas esse é outro processo caro e com resultados incertos. Assim, é comum que as empresas optem por reconstruir a aplicação utilizando tecnologias mais recentes no mercado. Esse é um bom cenário para se pensar em Entrega Contínua antes do início do desenvolvimento. Primeiro, porque o software legado ainda é funcional, ou seja, o processo de negócio não é prejudicado até a construção da nova aplicação. Segundo, porque a empresa já está no mercado e pode conseguir arcar com os custos desse novo projeto usando esse novo método. Por fim, entregando continuamente, será possível encontrar novas perspectivas do funcionamento do Software. Será possível validar, ainda durante o desenvolvimento, se as funcionalidades essenciais presentes no Software legado estarão presentes na nova versão e se as novas funcionalidades requisitadas atenderão as expectativas dos usuários finais.

Já nos projetos nos quais o cliente final ainda não é conhecido, não é possível determinar o ambiente onde o software será executado, ou ainda não é possível tomar todas as decisões de projeto, como a plataforma de disponibilização do serviço, capacidade de uso da aplicação, frameworks, etc., não é interessante se pensar em Entrega Contínua desde o início. Esses cenários são cobertos de incertezas. Algumas decisões tomadas durante a construção do projeto podem ser tão impactantes que pode ser preciso eliminar tudo o que foi produzido até o momento e recomeçar a construção do zero. Entrega Contínua requer maturidade, experiênia, conhecimento do ambiente do cliente e decisões definidas dos padrões e processos utilizados para o desenvolvimento. Ao estabelecer um

ambiente de Entrega Contínua, uma mudança de impacto, como trocar a linguagem de programação, pode prejudicar todo o projeto, sendo necessário reconstruir grande parte da infraestrutura de entrega, ou mesmo toda a infraestrutura. Portanto, não faz sentido despender um esforço para entregar continuamente quando ainda não é possível determinar o futuro do Software. A Entrega Contínua pode ser construída no futuro onde o produto já foi viabilizado no mercado. Muitas empresas fazem isso e não é um problema. Portanto se o futuro da aplicação é incerto e existem muitas incertezas em relação ao produto talvez seja melhor, em um primeiro momento pensar em outras estratégias de desenvolvimento. O mais importante é amadurecer a ideia do produto e garantir a viabilidade de produção do mesmo. Se o projeto tem viabilidade, a prática da Entrega Contínua trará muitos ganhos.

Construir para Entrega Contínua é mudar os paradigmas comuns do desenvolvimento do Software desde antes de começar a codificar o projeto. Essa quebra de paradigma requer profundas mudanças na cultura de desenvolvimento. Também é preciso estar alinhado as práticas de desenvolvimento ágil. Tais mudanças de paradigmas e cultura, apesar de difíceis, tendem somente a trazerem benefícios para o projeto. A prática DevOps costuma dizer que ”se machuca, faça mais vezes”[15]. Através da repetição dos processos difíceis, a prática surge e as oportunidades de automatização são encontradas. Com o tempo esses processos deixam de ser dolorosos e passam a agregar valor ao negócio.

Contudo, há o dever de se enxergar o problema a ser resolvido e conhecer as limitações. É possível sim, pensar em Entrega Contínua desde o início do projeto. Construir aplicações possíveis de serem entregues continuamente é uma forma de se escrever aplicações. Modularizar aplicações em pequenos pacotes coesos e com objetivos específicos, manter as configurações centralizadas e disponíveis aos interessados, escrever testes automatizados para as unidades, componentes e sistemas, integrar continuamente o trabalhos das equipes, praticar *commits* frequentes e de poucas porções de código por vez, entre outras práticas, são atitudes de um time que está preparado para Entrega Continua, mesmo que ainda não exista a automatização no provisionamento de infraestrutura e instalação da aplicação.

Automatizar o provisionamento de ambientes e implantar automaticamente a aplicação é o ponto crítico, onde é possível dizer que se está efetivamente aplicando a Entrega Contínua. Esse ponto, porém, nem sempre é possível para qualquer tipo de aplicação. Se não é possível identificar quem são os clientes finais, nem quais são as caracteristicas do ambiente onde a aplicação será executada, torna-se inviável aplicar esforços para automatizar essas etapas. Se a aplicação também não está completamente definida e as decisões críticas do projeto não podem ser determinadas, aplicar Entrega Contínua pode ser um desperdício de tempo e dinheiro. O esforço para construir um

ambiente de Entrega Contínua pode ser perdido se uma decisão mal tomada no início do projeto precisar ser modificada, como, por exemplo, trocar a linguagem de programação da aplicação.

Existem vantagens e desvantagem em aplicar a Entrega Contínua desde o início do projeto. A principal vantagem é a diminuição do *gap* existente entre a equipe de desenvolvimento e o cliente final. Praticar a Entrega Contínua estimula a participação do cliente no projeto. Ele pode ver, diariamente o progresso da aplicação, quando é possível entregar o que foi feito naquele dia para ele, mesmo que em ambiente de testes. As respostas a desafíos e a mudanças também é rápida. Problemas, quando surgem, são possíveis de serem corrigidos e ter sua solução entregue rapidamente. Todos os envolvidos tem uma visão clara do projeto, seu estado, e se problemas ocorrem, é possível saber onde aconteceu, quando aconteceu, porque aconteceu e como corrigir. A aplicação desenvolvida sobre essa prática torná-se mais confíavel e estável, por ser testável e rapidamente reproduzível. Já as desvantagens é que essa prática não é tão simples de ser aplicada por equipes menos maduras. O custo para se manter a Entrega Contínua é considerável. O choque na cultura, dependendo do grau de maturidade da equipe pode ser grande e até que se consiga estabilizar a organização para praticar a Entrega Contínua, muitas coisas podem acontecer. É preciso lembrar que, por mais que a mudança seja desejável e bem vista na tecnologia, as pessoas tendem a se incomodar com mudanças bruscas e resistir para que elas não ocorram. Propor entregar continuamente projetos desde a fase de concepção para equipes que estão acostumadas a desenvolver fora dessa perspectiva é um desafio que o líder de um projeto deve ter em mente e estar preparado para lhe dar.

Se o líder de um projeto tem maturidade para lidar com as mudanças que a prática da Entrega Contínua exige e consegue levar sua equipe a executar essa prática, o esforço despendido para aplicar a Entrega Contínua irá mostrar seus resultados a médio e longo prazo. Quando as equipes e o cliente conseguirem provisionar um ambiente e ter a aplicação instalada e funcional na versão que eles precisam simplesmente clicando em um botão, será possível constatar todos os benefícios que a Entrega Contínua proporciona. Todos os interessados serão capazes de verificar que a entrega de valor do Software será real e eficaz.

Por fim, é possível verificar que o ambiente de Entrega Contínua amadurece e se modifica com o tempo e com as novas solicitações que aparecem. Assim, a prática pode iniciar com o estabelecimento de um ambiente mínimo viável. Conforme o projeto cresce e novas necessidades aparecem, o ambiente de Entrega Contínua deve ser melhorado, incrementando as novas necessidades e as novas etapas do processo. O uso de um modelo de maturidade de Entrega Contínua pode auxiliar bastante nesse processo, apesar desse modelo não ter sido abordado diretamente nesse trabalho, ficando como uma proposta de

trabalho futuro.

É preciso conhecer bem o ambiente onde a aplicação irá executar, as caracteristicas e capacidades do ambiente do cliente, os requisitos funcionais e não funcionais do Software. É necessário que as decisões de projeto sejam tomadas acertivamente no início do projeto e que os interessados pelo projeto estejam alinhados quanto o método aplicado e o Software desenvolvido. Uma vez que esses requisitos são atendidos, o software desenvolvido será perceptivelmente melhor. Ao entregar continuamente, a equipe de desenvolvimento estará efetivamente agregando valor para o cliente final, amadurecendo seus processos e se impondo cada vez mais no mercado como competente e profissional.

Referências Bibliográficas

[1] AKOND ASHFAQUE UR RAHMAN, ERIC HELMS, L. W.; PARNIN, C. **Synthesizing continuous deployment practices used in software develpment**. *2015 Agile Conference*, 2015.

[2] ANDREAS REHN, T. P. E. P. B. **The continuous delivery maturity model**. https://www.infoq.com/articles/Continuous-Delivery-Maturity-Model, 2013. Acessado em: 6 de julho de 2016.

[3] ANICHE, M. **Test-Driven Development, Test Design No Mundo Real**. Casa do Cï¿½digo, 2012.

[4] ANICHE, M. **Testes Automatizados de Software - Um guia prï¿½tico**. Casa do Cï¿½digo, 2015.

[5] B. KIEPUSZEWSKI, A. T. H. E. C. B. **On structured workflow modelling**. *Lecture Notes in Computer Science*, 2000.

[6] CHEN, L. **Towards architecting for continuous delivery**. *IEEE Computer Society*, 2015.

[7] DIAS, A. F. **ComparaÃ§Ã£o de desempenho entre subversion, mercurial e git**. , 2016. Acessado em: 27 de julho de 2016.

[8] FOWLER , M. F. **Continuous delivery**. http://martinfowler.com/bliki/ContinuousDelivery.html, 2013. Acessado em: 19 de abril de 2016.

[9] FOWLER, HTTP://WWW.MARTINFOWLER.COM/. **Feature toggle**. http://martinfowler.com/bliki/FeatureToggle.html, 2010. Acessado em: 27 de junho de 2016.

[10] FOWLER, HTTP://WWW.MARTINFOWLER.COM/. **Phoenixserver**. http://martinfowler.com/bliki/PhoenixServer.html, 2012. Acessado em: 27 de junho de 2016.

[11] FOWLER, HTTP://WWW.MARTINFOWLER.COM/ , M. F. **Continuous integration**. http://www.martinfowler.com/articles/continuousIntegration.html, 2006. Acessado em: 19 de abril de 2016.

[12] FOWLER, HTTP://WWW.MARTINFOWLER.COM/ , M. F. **Deploymentpipeline**. http://martinfowler.com/bliki/DeploymentPipeline.html, 2013. Acessado em: 27 de abril de 2016.

[13] HUMBLE, J.; FARLEY, D. **Continuous Delivery, Reliable Software Releases Through Build, Test and Deployment Automation**. Addison-Wesley Personal Education, 2011.

[14] KENT BECK, MIKE BEEDLE, A. V. B. A. C. W. C. M. F. J. G. J. H. A. H. R. J. J. K. B. M. R. C. M. S. M. K. S. J. S.; THOMAS, D. **Agile manifesto**. , 2001.

[15] LEN BASS, I. W.; ZHU, L. **DevOps: A Software Architect's Perspective**. Addison-Wesley, New York, Boston, Indianopolis, San Francisco, Toronto, Montreal, London, Munich, Paris, Madrid, Capetown, Sydney, Tokyo, Singapore, Mexico City, 2015.

[16] LUCY ELLEN LWAKATARE, TEEMU KARVONEN, T. K. T. S. P. K. H. H. O. J. B.; OIVO, M. **Towards devops in the embedded systems domain: Why is it so hard?** *49th Hawaii International Conference on System Sciences*, 2016.

[17] MCCONNEL, S. **Code Complete, Um guia prï¿½tico para construï¿½ï¿½o de software**. ARTMED Editora S.A., 2004.

[18] NEELY, S.; STOLT, S. **Continuous delivery? easy! just change everything (well, maybe it is not that easy)**. *2013 Agile Conference*, 2013.

[19] P. DURVAL, S. M.; GLOVER, A. **Continuous Integration, Improving Software Quality and reducing risk**. Addison-Wesley Personal Education, 2007.

[20] SATO, D. **Implantaï¿½ï¿½es canï¿½rio**.

[21] SATO, D. **Implementando implantaï¿½ï¿½es azul-verde com amazon web services(aws)**. https://www.thoughtworks.com/insights/blog/implementando-implantacoes-azul-verde-com-amazon-web-services-aws, 2014. Acessado em: 27 de junho de 2016.

[22] SATO, D. **Mudanï¿½a paralela**.

[23] SMART, J. F. **Jenkins: The Definitive Guide**. O'Really Media, United States, softcover edition, 2011.

[24] SOMMERVILLE, I. **Engenharia de Software**. Pearson - Universitarios, softcover edition, 2011.

[25] **Developer survey 2015**. , 2015. Acessado em: 27 de julho de 2016.

[26] **SWEBOK V3.0 - Guide to the Software Engineering Body of Knowledge**. IEEE Computer Society, 2014.

[27] WILSENACH, R. **Devopsculture**. http://martinfowler.com/bliki/DevOpsCulture.html, 2015. Acessado em: 8 de julho de 2016.

APÊNDICE A

Ferramentas Utilizadas

A.1 Git

O Git é uma ferramenta de versionamento de artefatos colaboratíva e de código aberto que foi concebida em 2005 pela comunidade desenvolvedora do Kernel do Linux, principalmente na pessoa de Linus Torvalds. O desenvolvimento da ferramenta começou depois do fim do relacionamento entre a comunidade que desenvolvia o kernel do Linux e a empresa BitKeeper que fornecia o sistema de versionamento de artefatos.

O Git é um sistema de controle de versão distribuído. Dentre as vantagens de usar o Git estão sua velocidade na execução das operaçãoes, o *design* simples, o suporte robusto a desenvolvimentos não lineares (esquema de *branches*), ser completamente distribuído, além de ser apto para gerenciar projetos de larga escala eficientemente.

Mais informações sobre o Git no *site* oficial: https://git-scm.com/book/en/v2

A.2 Gitlab

O Gitlab[1] é uma ferramenta que começou a ser desenvolvida em 2011 por Dmitriy Zaporozhets por não estar satisfeito com os gerenciadores de repositórios baseados no Git. Sendo um projeto *Open-Source*, o Gitlab foi pensado para ser uma ferramenta para codificação, teste e implantação simultâneos. A partir de um repositório de controle de versão de artefados baseado no Git, o Gitlab disponibiliza ferramentas de controle de acessos, revisão de código, gerenciamento de atividades, *feed* de atividades, *Wikis* e até Integração Contínua.

[1] Detalhes sobre a história do Gitlab podem ser vistos em https://about.gitlab.com/about/

Sendo estabelecida como uma corporação em 2014, o Gitlab Inc. disponibiliza quatro produtos. Uma versão comunitária gratuíta, auto hospedada que é apoiada pela comunidade; uma versão empresarial paga que disponibiliza funcionalidades adicionais e suporte; uma versão Software como Serviço (*SaaS*) para gerenciamento de repositórios públicos e privados onde o suporte pode ser comprado; e uma versão privada única que gera instâncias da versão Gitlab EE (*Enterprise Edition*) e CE (*Comunnity Edition*) que são gerenciadas pelo próprio Gitlab, mediante pagamento.

A.3 Jenkins

O Jenkins é uma plataforma *Open Source* escrita em Java cujo o objetivo é automatizar as fases do ciclo de vida da Integração Contínua do Software, podendo ir desde a compilação até a implantação em produção dos artefatos que ele gerencia. Ela foi a ferramenta principal do ambiente de Integração Contínua desse trabalho, pois é nela que foram configuradas as tarefas automatizadas a serem executadas a cada modificação que ocorrer no Sistema de Controle de Versão (mais detalhes no Capítulo 3).

O Jenkins surgiu por volta de 2004, com o nome de Hudson, um projeto feito por *hobby* pelo desenvolvedor Kohsuke Kawaguchi que trabalhava na Sun. O projeto foi ganhando espaço e por volta de 2008 a Sun reconheceu o potencial da ferramenta possibilitando ao Kohsuke trabalhar exclusivamente no desenvolvimento desse projeto que ganhou espaço e se tornou a ferramenta lider de mercado por volta de 2010. Com a compra da Sun pela Oracle em 2009 uma tensão surgiu entre os desenvolvedores do Hudson e a Oracle, principalmente no que diz respeito a forma como o projeto era gerenciando e evoluído. Assim em 2011 foi criado um *fork*[2] do projeto Hudson no Github com o nome de Jenkins e assim o projeto foi aberto a comunidade para desenvolvimento, não estando mais ligado à Oracle, o que possibilitou ao Jenkins um crescimento rápido e uma evolução constante, fazendo com que essa ferramenta se tornasse muito utilizada no mercado de desenvolvimento de Software[23].

Essa ideologia *Open Source* faz com que o Jenkins seja bastante flexível e extensível através dos inúmeros plugins disponibilizados pela comunidade para execução de várias das fases do Ciclo de Vida do desenvolvimento do Software, podendo customizar e criar fluxos de trabalhos complexos que se adaptem à necessidade de negócio.

[2] *Fork* é uma cópia de um repositório de código fonte. Fazer um fork de um repositório permite fazer experimentos e mudanças sem afetar o projeto original. Também é uma nova linha de desenvolvimento de um produto à parte do projeto original.

A.4 Maven

O Maven é uma ferramenta utilizada para facilitar o processo de construir e gerenciar projetos Java. No site do projeto são citados os objetivos da ferramenta:

- Tornar o processo de construção mais fácil;
- Prover um sistema de construção unificado;
- Prover informações de qualidade sobre o projeto;
- Prover orientações para as melhores práticas de desenvolvimento;
- Permitir migrações transparentes para novas funcionalidades;

O Maven também pode ser utilizado para construir projetos escrito em C, Ruby, Scala, dentre outras linguagens. O projeto Maven é hospedado pela Apache Software Fundation[3].

O Maven utiliza um arquivo XML para descrever a construção do Software, bem como suas dependências. Esse arquivo é o **pom.xml**. Nesse arquivo são descritas as dependências entre os módulos, os componentes externos, a ordem de compilação, os diretórios utilizados pelo projeto bem como os *plugins* necessários para construção.

As dependências, bibliotecas de terceiros e *plugins* são baixados automaticamente através da internet, utilizando os repositórios centrais disponíves, tal como o Maven 2 Central Repository [4] ou um repositório local (utilizando Nexus, por exemplo). As dependências baixadas são mantidas localmente, o que diminui o tempo de construções futuras. As versões presentes localmente não precisam ser baixadas novamente pela Internet.

Por utilizar uma arquitetura baseada em *plugins*, é muito fácil desenvolver ou mesmo extender as funcionalidades do Maven. Isso torna o processo de construção de um software personalizável e extensível.

A.5 JUnit

Conforme já dito no Capítulo 4, Seção 4.1.3 - Executar Testes, não é possível dizer que existe Integração Contínua se não existir testes automatizados que garantam a confiabilidade do Software desenvolvido. Testes Unitários são responsáveis por verificar o comportamento das menores unidades de código do sistema, em geral as classes[19].

[3] Site oficial da Apache Software Fundation: https://www.apache.org/
[4] O site do Maven 2 Central Repository é: http://search.maven.org/

No livro de Paul M. Duvall, ele diz que:

> "a confiabilidade de um sistema linear é o produto da confiabilidade de cada um dos componentes do sistema."

. Por exemplo, em um software que possui 3 classes, a confiabilidade será medida pela multiplicação da confiabilidade de cada classe individual. Ou seja, se a confiabilidade de cada classe é de 95% (não importa aqui como essa confiabilidade foi medida), um sistema com 3 classes teria uma confiabilidade de 85% (0.95 * 0.95 * 0.95 = 0.85);

Como nosso sistema será escrito em Java, nossos testes unitários serão desenvolvidos utilizando como ferramenta o JUnit. O JUnit é um *framework* simples para a escrita de testes automatizados. Ele é uma instância da arquitetura *xUnit* para *frameworks* de testes unitários.

A.6 Mockito

Durante a escrita de testes unitários, na implementação de uma classe, é necessário que ela faça interações com outras classes ou sistemas. Na execução de testes unitários essas interações muitas vezes podem ser indesejadas. Por exemplo, uma classe em um determinado momento precisa consultar um objeto no banco de dados. Essa implementação, por si só, obrigará o desenvolvedor a ter conexão com o banco de dados durante a execução dos testes e ainda obrigará o desenvolvedor a garantir que o objeto que está sendo consultado exista no banco com as características necessárias para o teste. E se o banco estiver indisponível? E se o objeto consultado tiver sido modificado, afetando a execução do teste? E se, durante a consulta ocorrer um problema de conexão e a consulta não ser concluída com sucesso? Em todos esses casos o teste irá gerar uma falha na Integração Contínua.

Pensando nisso alguns *frameworks* foram desenvolvidos para simular interações com sistemas externo. Essa simulação significa forçar o teste a simular um resultado mediante uma ação específica. Por exemplo, instruir o teste a retornar um objeto com determinadas características quando a classe que está sendo testada tentar se conectar com o banco de dados para buscar tal objeto. Assim o teste unitário não depende de componentes externos para sua execução.

Neste trabalho foi utilizado o *framework* Mockito[5] para simulação nos testes unitários.

[5] Veja mais em: http://mockito.org/

A.7 FlyWay

O controle da evolução do banco de dados é assunto importante quando falamos de Integração Continua (assunto do Capítulo 4). Conforme Paul M. Duvall relatou, uma das fases da Integração Continua é a Integração da Base de Dados (assunto abordado na Seção 4.1.2). Assim a evolução da versão do banco de dados deve ser constantemente construída e testada para garantir que o esquema do Banco de Dados está de acordo com o combinado no projeto. Alem disso, a integração da base de dados torna mais fácil para a equipe de desenvolvimento a recriação da base em versões especificas sempre que for necessário, não precisando sobrecarregar o time de Banco de Dados com tarefas repetitivas, como restaurar dados de teste de um banco para um teste específico da aplicação.

No sistema construido neste trabalho foi utilizado o Flyway[6] como Sistema de Gerenciamento de Migrações do Banco de Dados.

A.8 Nexus

Quando falamos de implantação da aplicação estamos falando em disponibilizar o artefato binário produzido pela compilação, testes e inspeção em um Gerenciador de Artefatos. Assim, será possível aos demais membros da equipe acessar tal artefato como dependência dos módulos que ele está desenvolvendo e que porventura necessitem. Faz-se necessário, então, um gerenciador de dependências que disponibilize as dependências necessárias para desenvolver a aplicação. Essas dependências podem ser externas (bibliotecas de terceiros) ou internas (artefatos produzidos pela própria equipe de desenvolvimento do projeto).

Neste trabalho foi utilizado o Nexus[7] para nos servir de repositório de artefatos binários. Ele será responsável por manter os artefatos compilados que forem desenvolvidos para integrar com outras partes da aplicação assim como os artefatos externos que forem necessários.

Cada vez que o ciclo da Integração Contínua executa, os artefatos gerados serão disponibilizados devidamente versionados no Nexus, ou seja, entregues para que o Nexus os mantenha e os disponibilize a quem precisar.

[6] Veja mais em: http://flywaydb.org/
[7] Veja mais em: http://www.sonatype.org/nexus

A.9 Sonar Qube

Para a inspeção dos códigos fontes será utilizado o SonarQube[8]. Essa inspeção visa verificar a conformidade do código escrito com os padrões previamente estabelecidos pela organização.

Com o SonarQube foi possível definir quais são as regras de escrita do código, assim como as métricas que devem ser seguidas pela equipe de desenvolvimento. A execução da análise do SonarQube garante que o código desenvolvido siga tais padrões, sempre notificando a equipe de desenvolvimento quando tais normas não forem seguidas. Por padrão, o Sonar contém várias funcionalidades e suporte e múltiplas linguagens, como Java, C/C++, Objective-C, C, PHP, Flex, Groovy, JavaScript, Python, PL/SQL, COBOL, Swift, entre outras. Entre as funcionalidades disponibilizadas pelo Sonar estão:

- Verificação de duplicação de código: ferramenta que encontra e aponta códigos duplicados, a fim de facilitar a manutenabilidade do código fonte;
- Padrões de Codificação: o Sonar consegue analisar padronizações de codificação. Essas padronizações visam facilitar a legibilidade do código e a compreensão do sistema como um todo;
- Verificação de testes unitários: o Sonar também avalia os testes unitários e a cobertura desses testes no sistema, apontando quais códigos estão cobertos e os que não estão, tornando a aplicação mais confíavel e segura.
- Verificação da complexidade do código: o Sonar analisa o código fonte, os artefatos de Software e seus procedimentos e utiliza de modelos padrões para definir a complexidade do código. Essa complexidade é utilizada como parâmetro para determinar a qualidade do código produzido.
- Potenciais erros: através da análise do código fonte, o Sonar é capaz de encontrar possíveis erros no Software. Ele usa uma ferramenta interna que gerencia e disponibiliza problemas no código de maneira automática. Quando problemas são detectados, o Sonar cria uma tarefa para correção daquele erro e consegue atribuir para colaboradores para resolução. Quando o problema é solucionado e commitado, o Sonar tem a capacidade de reportar e encerrar a atividade de erro automaticamente.

Além de tudo o Sonar é possível de ser integrado com ferramentas de construção e gerenciamento de dependências, como o Maven, Ant e Gradle, além de ferramentas

[8] Veja mais em: http://www.sonarqube.org/

de integração contínua, como Jenkins, Hudson, Atlassian Bamboo, entre outras. Também é possível integrar o Sonar com IDEs como Eclipse, Visual Studio e Intellij IDEA.