AGILIDADE NO DESENVOLVIMENTO DE SOFTWARE UTILIZANDO INTEGRAÇÃO CONTÍNUA

Felipe Loge dos Santos Lira[1]
Ronaldo Penha Zanoni[2]
Anderson Francisco Talon[3]

Resumo: A pesquisa desenvolvida procura demonstrar a eficácia da utilização da Integração Contínua para um desenvolvimento ágil de aplicações, utilizando GIT, Jenkins e Docker, visando apresentar individualmente o objetivo de cada ferramenta, e como a integração de todas resulta em um software de qualidade com entregas contínuas para o cliente, em um tempo reduzido, sem a necessidade de testar as funcionalidades que já estão funcionando, pois este processo é todo automático.

Palavras-chave: Desenvolvimento Ágil. Integração Contínua. Qualidade de Software.

1 INTRODUÇÃO

Sistemas de software estão cada vez mais presente no nosso dia-a-dia, desde aplicações empresariais como ERP, até aplicações de uso pessoal. A maioria das pessoas já passaram por uma experiência de usuário final em que o software não funcionou corretamente como o esperado. A falha de uma funcionalidade do sistema pode resultar em perdas de tempo e financeiras. Desta forma, os testes possuem um papel importante no desenvolvimento de um software.

Entretanto, o processo de testes também pode apresentar problemas, como o retrabalho, aumentando os custos e tempo de desenvolvimento do software. Para cada nova versão, há o risco de interromper alguma funcionalidade existente, pois a equipe pode acabar esquecendo de testar algo em específico, ou então o servidor que está sendo utilizado em produção, possui configurações diferentes do servidor utilizado em desenvolvimento local, interrompendo assim, o funcionamento de algo.

1 Aluno do Curso de Sistemas de Informação do Centro Universitário de Bauru. E-mail: felipeloge@gmail.com.
2 Aluno do Curso de Sistemas de Informação do Centro Universitário de Bauru. E-mail: ronaldozanoni@hotmail.com.
3 Professor do Curso de Sistemas de Informação do Centro Universitário de Bauru. E-mail: anderson.talon@ite.edu.br.

Com a Integração Contínua, é possível garantir a qualidade do produto final, pois todos os testes que já foram realizados antes, serão automatizados, não precisando testá-los novamente, e caso a nova funcionalidade interrompa o funcionamento de alguma existente, a mesma não será enviada para produção. Os ambientes de desenvolvimento e produção serão sempre idênticos, utilizando as mesmas configurações e prevenindo erros comuns em ambientes distintos.

1.1 Objetivo

Esta pesquisa tem como objetivo analisar um processo de desenvolvimento ágil e de qualidade, demonstrando técnicas e ferramentas para realizar o processo. Muitas empresas utilizam pessoas para testar seus softwares, deixando assim, o produto sempre sujeito a erros. Com o passar dos tempos, os usuários estão selecionando melhor seus softwares, não utilizando softwares com erros. Com isso, as empresas precisam melhorar o processo de desenvolvimento para evitar insatisfação e perda de clientes.

1.2 Justificativa

Atualmente, muitas empresas de software encontram problemas ao concluir novas funcionalidades nos sistemas. Através de estudos e pesquisas, a Integração Contínua, de uma forma automatizada, seria uma das possibilidades de garantir a qualidade de uma nova funcionalidade. Este estudo irá abordar a eficácia da Integração Contínua no desenvolvimento de software.

2 TESTES AUTOMATIZADOS

A automação de testes pode ser descrita como a aplicação de ferramentas para reduzir o envolvimento humano nos testes e ter capacidade de executar atividades repetidas em tempo hábil, conforme afirma Bernardo:

> Testes automatizados são programas ou scripts simples que exercitam funcionalidades do sistema sendo testado e fazem verificações automáticas nos efeitos colaterais obtidos. A grande vantagem desta abordagem é que todos os casos de teste podem ser facilmente e rapidamente repetidos a qualquer momento e com pouco esforço. (BERNARDO, 2008).

Executando os testes automatizados é possível detectar inúmeros problemas em diversos cenários, facilitando e colaborando com o desenvolvedor em identificar um comportamento indesejável. (BERNARDO, 2008).

O planejamento dos testes deve ocorrer em diferentes níveis e em paralelo ao desenvolvimento do software. Os principais níveis de testes são (NETO, 2008):

2.1 Teste Unitário

Tem por objetivo explorar a menor unidade do projeto, procurando provocar falhas ocasionadas por defeitos de lógica e de implementação em cada módulo, separadamente. O universo alvo desse tipo de teste são os métodos dos objetos ou mesmo pequenos trechos de código.

2.2 Teste de Integração

Também conhecido como teste *end-2-end*, visa provocar falhas associadas às interfaces entre os módulos quando esses são integrados para construir a estrutura do software que foi estabelecida na fase de projeto.

3 METODOLOGIAS ÁGEIS

As metodologias ágeis surgiram para substituir as metodologias tradicionais e melhorar o processo de desenvolvimento de software. As mais utilizadas atualmente são XP (eXtreme Programming) e Scrum.

> As metodologias ágeis surgiram como uma alternativa as metodologias tradicionais, visando a agilidade no desenvolvimento de softwares, onde é possível fazer a incorporação de modificações que venham a ocorrer no decorrer do projeto. As metodologias ágeis mais usadas atualmente são a SCRUM e o XP (eXtreme Programming). (AUGUSTO, 2013).

3.1 XP

A metodologia XP é caracterizada pelo seu princípio de *feedback* constante, contando com uma entrega de código constante dos desenvolvedores, que ajuda a atingir o objetivo do cliente, sempre tendo um *feedback* relacionado aos requisitos do projeto. Com isso, é possível assegurar que o software recebido pelo cliente, sempre será de alto valor, pois está de acordo com os requisitos (TELES, 2005).

3.2 Scrum

Segundo Schwaber e Sutherland:

> O framework Scrum é um framework onde se pode tratar e resolver problemas complexos e adaptativos, ao mesmo tempo em que se desenvolvem produtos que sejam produtivos e criativos e que tenham o maior valor possível agregado. Umas das características do Scrum é ser: (SCHWABER, SUTHERLAND, 2011)
>
> - Leve;
> - Simples de entender;
> - Extremamente difícil de dominar.

Para resolver os problemas, o Scrum divide o projeto em ciclos de trabalho, conhecidos como *sprint*, que representa etapas do projeto para ser concluídas. Para começar um novo *sprint*, deve-se sempre terminar o anterior, para assim, ter um desenvolvimento organizado.

A cada dia do *sprint*, a equipe realiza uma breve reunião para verificar o andamento do projeto, informando o que foi feito para concluir o *sprint* atual até o momento, e o que falta ser feito. Caso algum membro teve problemas para executar suas atividades, deve-se mencionar o mesmo na reunião, para outros membros ajudarem, se for possível. (GUERRATO, 2013).

4 INTEGRAÇÃO CONTÍNUA (IC)

Segundo Fowler:

> A Integração Contínua é uma prática de desenvolvimento de software onde os membros de um time integram seu trabalho frequentemente, geralmente cada pessoa integra pelo menos diariamente – podendo haver múltiplas integrações por dia. Cada integração é verificada por um build automatizado (incluindo testes) para detectar erros de integração o mais rápido possível. Muitos times acham que essa abordagem leva a uma significante redução nos problemas de integração e permite que um time desenvolva software coeso mais rapidamente. (FOWLER, 2006).

Utilizar IC é uma forma de atingir a qualidade no desenvolvimento de software, pois os desenvolvedores tem *feedback* quase instantâneo sobre suas atividades, podendo assim resolver os problemas, caso existam. Com IC, as entregas de software são mais seguras e constantes, pois todo o processo é automático, e sempre é testado todas as funcionalidades repetidamente, garantindo que não irá interromper o funcionamento de algo que já está funcionando. Caso ocorra algum erro nos testes, a equipe tem um aviso imediato, através de e-mail,

podendo assim, resolver o problema, o que é fundamental para não enviar funcionalidades incompletas em um ambiente de produção. (GUERRA, 2008).

5 FERRAMENTAS PARA INTEGRAÇÃO CONTÍNUA

Para colocar tudo isso em prática, todo o processo de *deploy* (construção da versão final) automático, é necessário algumas ferramentas.

5.1 Controle de versão

Para ter controle de todo processo de desenvolvimento, separar cada funcionalidade em um código diferente, e conseguir integrar o processo de *deploy*, é necessário um controle de versão.

> O controle de versão é um sistema que registra as mudanças feitas em um arquivo ou um conjunto de arquivos ao longo do tempo de forma que você possa recuperar versões específicas. (CHACON, STRAUB, 2009).

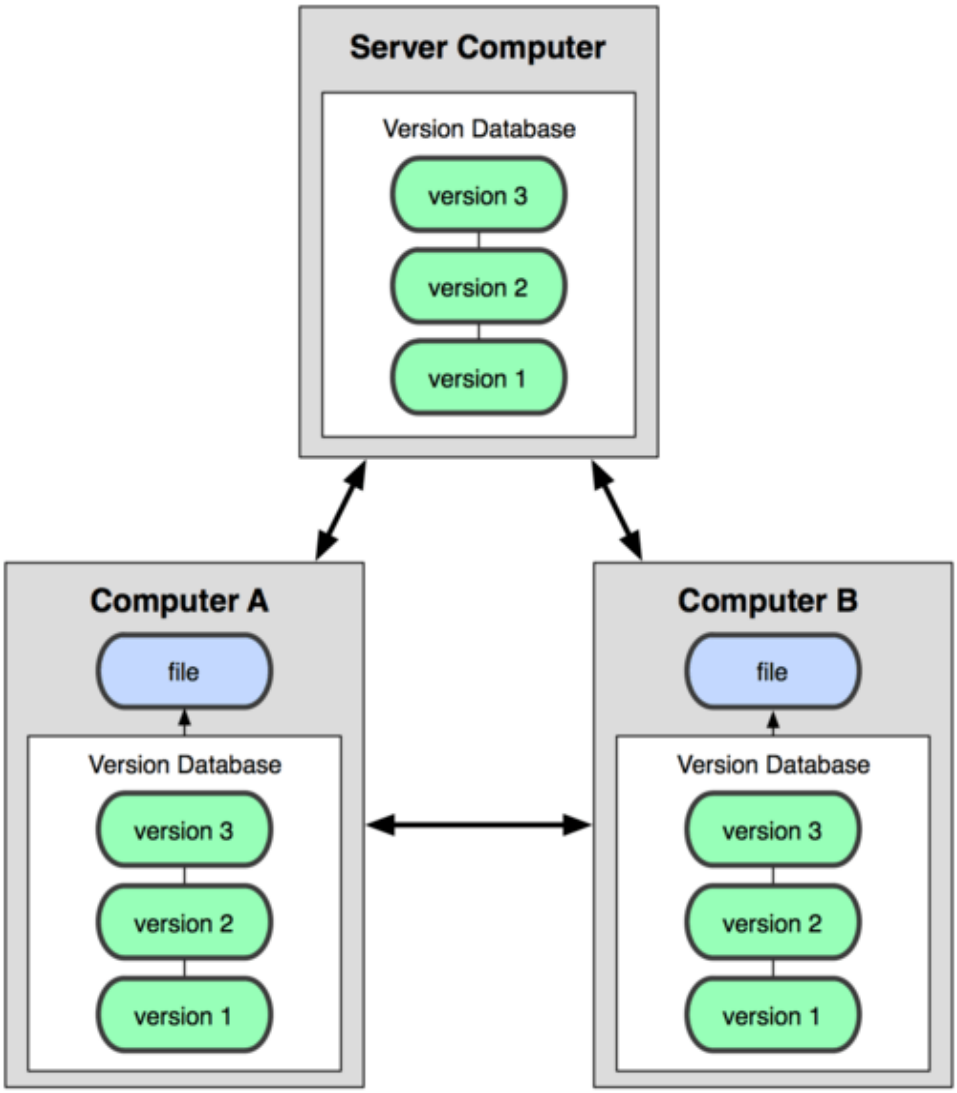
Conforme a figura 1, é demonstrado o uso de um controle de versão por dois usuários, acessando os arquivos de um servidor, e trabalhando em cima destes arquivos com versões locais, podendo assim ter várias versões do mesmo código, cada versão com uma funcionalidade específica, tendo a possibilidade de juntar todo o trabalho a qualquer momento.

Utilizando controle de versão, os desenvolvedores trabalham com cópias, ao invés dos arquivos originais, podendo recuperar qualquer arquivo a qualquer momento.

> Os clientes não apenas fazem cópias das últimas versões dos arquivos: eles são cópias completas do repositório. Assim, se um servidor falha, qualquer um dos repositórios dos clientes pode ser copiado de volta para o servidor para restaurá-lo. Cada *checkout* (resgate) é na prática um backup completo de todos os dados. (CHACON, STRAUB, 2009).

Existem muitas ferramentas para trabalhar com controle de versão, como Git, Subversion e Team Foundation Server.

Figura 1 - Diagrama de controle de versão distribuído



Fonte: Git, 2009.

5.1.1 Commit

Baseando-se no livro Pro Git book, entende-se que *commit* é a ação de enviar seu código que está em seu computador, para a nuvem, junto com todo o código já existente. Com isso, outros desenvolvedores podem acessar o código enviado pelo *commit*.

5.1.2 Branch

Baseando-se no livro Pro Git book, entende-se que *branch* no Git é simplesmente uma versão de código de um *commit*. O nome do *branch* padrão no Git é master. Quando você faz *commits*, você tem um *branch* principal (master

branch) que aponta para o último commit que você fez. Cada vez que você faz um *commit* ele avança automaticamente. É possível criar muitos *branchs*, cada um com uma versão específica, e recuperá-los a qualquer momento.

5.1.3 Checkout

Baseando-se no livro Pro Git book, entende-se que *checkout* é a ação/comando feito para trocar de *branch*, utilizando assim, uma outra versão do código para desenvolver.

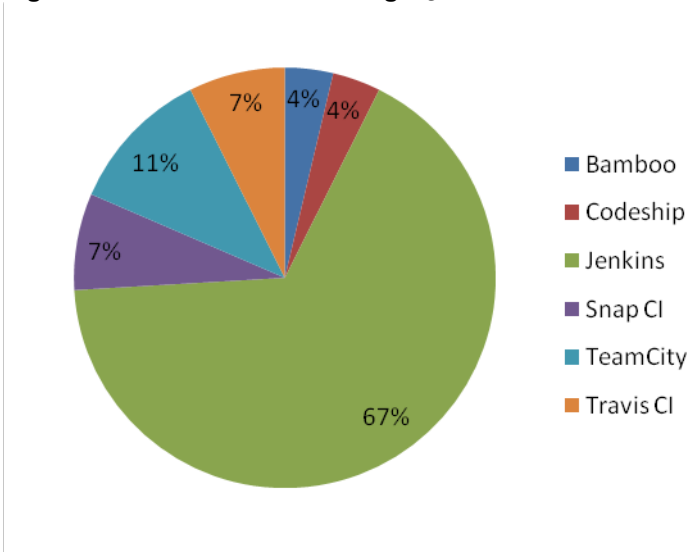
5.1.4 Push

Baseando-se no livro Pro Git book, entende-se que *push* é a ação/comando feito para enviar as alterações do commit para o servidor.

5.2 Jenkins

Conforme a questão de número 2 da pesquisa (mais informações sobre a pesquisa é encontrada no tópico 6), Jenkins é a ferramenta mais utilizada atualmente, com 67% dos usuários. Com isso, será explicado um pouco melhor o que é o Jenkins.

Figura 2 - Ferramentas de Integração Contínua



Fonte: Os autores, 2015.

Jenkins é uma ferramenta de código aberto para realizar integração contínua. A funcionalidade básica é executar uma lista pré-definida de passos para comandos, scripts.

Jenkins também pode ficar monitorando comandos, como por exemplo, alteração de código em uma versão específica, e após executado, gera uma versão de seu produto, com todas as mudanças feitas, e automaticamente utiliza os testes criados durante o processo para garantir que o código não interrompeu nenhum funcionamento.

Após executado o processo, toda equipe do projeto é avisada se o código está pronto para ser utilizado ou se interrompeu alguma funcionalidade.

O Jenkins pode ser iniciado por linha de comando ou por uma aplicação web.

5.3 Docker

Docker é uma plataforma que vem crescendo bastante, é aberta para desenvolvedores e administradores de sistemas. Não possui custos e sua principal utilidade para ser uma ferramenta para Integração Contínua é o fato da mesma conseguir utilizar um único ambiente em diferentes locais.

Pode ser utilizado para criar um ambiente de *deploy*, onde o Jenkins irá executar seus comandos, e pode ser utilizado para configurar um ambiente de desenvolvimento para desenvolvedores.

Desenvolvedores que utilizam o framework Ionic por exemplo, necessitam de muitas ferramentas globais que não possuem um gerenciador de versões, com isso, ocorre muitos problemas no momento de gerar um build pelo fato de que cada desenvolvedor pode estar usando uma versão diferente de uma ferramenta.

Com o Docker, todos os desenvolvedores conseguem acessar o mesmo ambiente para trabalhar. Ao iniciar um *container* no Docker, é possível compartilhar o código local com o novo ambiente e as alterações serem espelhadas em tempo real. Com isso, os desenvolvedores utilizariam o código local para desenvolver, e os comandos para testar e executar *builds*, seriam efetuados no *container* do Docker.

O maior diferencial do Docker em relação a outras tecnologias, é a utilização de *containers* ao invés de máquina virtual, o que acaba deixando o mesmo mais rápido que uma máquina virtual, mantendo o mesmo objetivo.

Máquina virtual é o nome dado a uma máquina, implementada através de software, que executa programas como um computador real. Com ela, é possível rodar outros sistemas operacionais dentro de uma janela, conseguindo acessar qualquer software conforme necessidade. VirtualBox é um exemplo de software para realizar virtualização.

Containers são semelhantes a máquinas virtuais, porém utiliza somente o necessário do sistema operacional, fazendo com que seja mais rápido que uma VM qualquer. (SILVA, 2015).

6 PESQUISA

Foi realizada uma pesquisa com mais de 20 pessoas que trabalham com Integração Contínua, com o objetivo de analisar a melhoria do desempenho utilizando a prática de desenvolvimento com IC.

É importante ressaltar que a unidade de medida utilizada nessa pesquisa é identificada com o termo *bug*, que significa uma quantidade fracionada relacionada a erros de projeto.

A pesquisa abordou as seguintes questões:

1. *Plataforma do projeto* - Importante para saber qual plataforma está utilizando com mais frequência a IC.
2. *Qual ferramenta de IC foi utilizada?* - Importante para aprofundar na ferramenta mais utilizada.
3. *Quantos bugs voltavam em um projeto sem utilizar IC?* - Importante para a análise da melhoria.
4. *Quantos bugs voltavam em um projeto utilizando IC?* - Importante para a análise da melhoria.
5. *Quantos deploys eram feitos semanalmente antes de se utilizar IC?* - Importante para a análise da melhoria.
6. *Quantos deploys eram feitos semanalmente após se utilizar IC?* - Importante para a análise da melhoria.
7. *Tempo estimado para se fazer um deploy em produção sem utilizar IC (desde o momento de gerar o pacote de build até colocar em produção)* - Importante para a análise da melhoria.

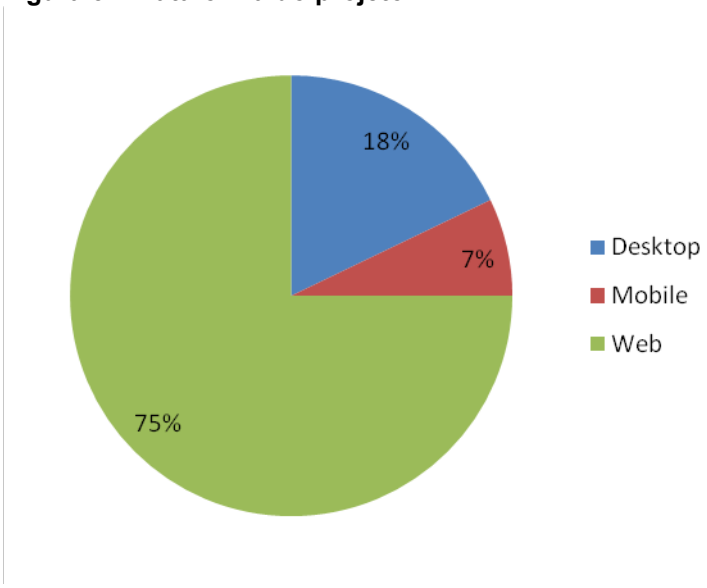
8. *Tempo estimado para se fazer deploy em produção utilizando IC (desde o momento de gerar o pacote de build até colocar em produção)* - Importante para a análise da melhoria.

7 ANÁLISE

Com base na pesquisa realizada,foi feita uma análise sobre as questões abordadas, informando a melhoria ou não, de um processo com IC.

Atualmente, conforme a figura 3, a Web é a principal plataforma utilizada para colocar em prática a IC, com 75% dos entrevistados.

Figura 3 - Plataforma do projeto



Fonte: Os autores, 2015.

Antes de utilizar IC em um desenvolvimento de software, retornavam em média, **22,63** *bugs* no decorrer do projeto. Após a aplicação da IC, a média dos *bugs* reduziram para **4,23**. Portanto, é possível notar uma redução de **81,30%** dos *bugs* nos projetos. Estes dados trazem mais confiança para os desenvolvedores e para toda equipe.

A quantidade de *deploys* realizados semanalmente antes de utilizar IC foi **1,92**. Após a aplicação da IC, esta quantidade aumentou para **5,41**. Ou seja, a quantidade de *deploys* aplicando IC aumentou em **2,82** vezes, com isso, é possível desenvolver atualizações para seu produto com muito mais frequência, satisfazendo o objetivo de seu cliente e usuário.

O tempo estimado para se fazer um *deploy* em produção, desde o momento de gerar o pacote de *build* até colocar em produção reduziu de **215,69** minutos para **38,5** minutos. É notável uma redução de **82,15%** no tempo de *deploy*, trazendo com isso, uma entrega contínua para o cliente e usuário final de seu sistema.

8 CONCLUSÃO

A Integração Contínua pode não ser a única forma de melhorar um processo de software e nem a mais fácil de se utilizar, porém, sua aplicação resulta em um processo ágil com entregas contínuas e sem precisar de pessoas para ficar testando algo que já foi testado, correndo o risco de esquecer de testar alguma funcionalidade em específico.

Além das entregas contínuas, a aplicação da IC resulta na confiança do que está sendo entregue para o cliente ou para o usuário, pois todos os processos são automáticos e o desenvolvedor tem um *feedback* quase instantâneo sobre o sucesso ou falha na execução dos testes.

Analisando os resultados, podemos observar que houve uma melhoria em todo o processo, diminuindo a quantidade de *bugs* em **81,30%**, aumentando a quantidade de *deploys* em **2,82** vezes, com o tempo de cada *deploy* reduzido em **82,15%**, podendo assim, entregar funcionalidades com mais qualidade e com uma maior frequência em um tempo reduzido, além de todo o processo ser automatizado.

AGILITY IN SOFTWARE DEVELOPMENT USING CONTINUOUS INTEGRATION

Abstract: The research uses the Continuous Integration for an agile development of application, using GIT, Jenkins and Docker. Showing the purpose of each tool, and how the integration of all results in a software of quality with continuous deliveries for the client, in a reduced time, without testing the features that are already working, because the process is all automatic.

Keywords: Agile Development. Continuous Integration. Software Quality.

REFERÊNCIAS

BERNARDO, Paulo Cheque. A Importância dos testes automatizados. **DevMedia**. 2008. Disponível em <http://www.devmedia.com.br/artigo-engenharia-de-software-3-a-importancia-dos-testes-automatizados/9532>. Acesso em: 24 set. 2015.

DIAS NETO, Arilo Cláudio. Introdução a Teste de Software. **DevMedia**. 2008. Disponível em <http://www.devmedia.com.br/artigo-engenharia-de-software-introducao-a-teste-de-software/8035>. Acesso em: 25 set. 2015.

GUERRA, Cauê. Integração Contínua e o processo Agile. **Caelum**. 2008. Disponível em <http://blog.caelum.com.br/integracao-continua>. Acesso em: 24 set. 2015.

GUERRATO, Dani. Desenvolvimento ágil utilizando Scrum. **Tableless**. 2013. Disponível em <http://tableless.com.br/desenvolvimento-agil-utilizando-scrum>. Acesso em: 25 set. 2015.

PAINKA, Marcelo Augusto Lima. **Utilização das Metodologias Ágeis XP e Scrum para o Desenvolvimento Rápido de Aplicações**. 2013. Disponível em: <http://web.unipar.br/~seinpar/2013/artigos/Marcelo%20Augusto%20Lima%20Painka.pdf>. Acesso em 25 set. 2015.

SCHWABER, Ken; SUTHERLAND, Jeff. Um guia definitivo para o Scrum: as regras do jogo. **Scrum Guides**. 2013. Disponível em <http://www.scrumguides.org/docs/scrumguide/v1/Scrum-Guide-Portuguese-BR.pdf>. Acesso em: 26 set. 2015.

SILVA, Wellington. Docker, do Básico a Orquestração e Clusterização – 1. Introdução. **PHP SP**. 2015. Disponível em <http://phpsp.org.br/docker-do-basico-a-orquestracao-e-clusterizacao-introducao/>. Acesso em: 26 set. 2015.

TELES, Vinícius Manhães. **Extreme Programming: aprenda como encantar seus usuários desenvolvendo software com agilidade e alta qualidade**. 1.ed. São Paulo: Novatec, 2004. 316 p.