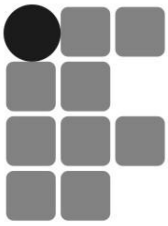


INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA FLUMINENSE
Campus Campos-Centro

Secretaria de Educação Profissional e Tecnológica

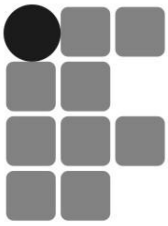
Ministério da Educação

CURSO DE TECNOLOGIA EM DESENVOLVIMENTO DE SOFTWARE

HERMAN DA ROSA CALDARA
RAFAEL DOS SANTOS GONÇALVES
VINÍCIUS DAS CHAGAS SILVA

ESTUDO E PROPOSTA DE USO DE INTEGRAÇÃO CONTÍNUA NO PROCESSO DE DESENVOLVIMENTO DA BIBLIOTECA DIGITAL DA RENAPI

Campos dos Goytacazes/RJ
2010



INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA FLUMINENSE
Campus Campos-Centro

Secretaria de Educação Profissional e Tecnológica

Ministério da Educação

CURSO DE TECNOLOGIA EM DESENVOLVIMENTO DE SOFTWARE

HERMAN DA ROSA CALDARA
RAFAEL DOS SANTOS GONÇALVES
VINÍCIUS DAS CHAGAS SILVA

ESTUDO E PROPOSTA DE USO DE INTEGRAÇÃO CONTÍNUA NO PROCESSO DE DESENVOLVIMENTO DA BIBLIOTECA DIGITAL DA RENAPI

Trabalho de conclusão de curso apresentado ao Instituto Federal Fluminense como requisito parcial para conclusão do Curso de Tecnologia em Desenvolvimento de Software.

Orientador: Prof. Fernando Carvalho

Campos dos Goytacazes/RJ
2010

HERMAN DA ROSA CALDARA
RAFAEL DOS SANTOS GONÇALVES
VINÍCIUS DAS CHAGAS SILVA

ESTUDO E PROPOSTA DE USO DE INTEGRAÇÃO CONTÍNUA NO PROCESSO DE DESENVOLVIMENTO DA BIBLIOTECA DIGITAL DA RENAPI

Trabalho de conclusão de curso apresentado ao Instituto Federal Fluminense como requisito parcial para conclusão do Curso de Tecnologia em Desenvolvimento de Software.

Aprovada em 29 de junho de 2010

Banca avaliadora:

Prof. Fernando Carvalho (Orientador)
Especialista em Produção e Sistemas / IFF Campus Campos
Instituto Federal de Educação, Ciência e Tecnologia Fluminense / Campus Bom Jesus

---

Prof. Rodrigo Soares Manhães
Mestre em Pesquisa Operacional e Inteligência Computacional / UCAM Campos
Universidade Estadual do Norte Fluminense / UENF

Prof. Fábio Duncan de Souza
Mestre em Pesquisa Operacional e Inteligência Computacional / UCAM Campos
Instituto Federal de Educação, Ciência e Tecnologia Fluminense / Campus Campos Centro

Às nossas famílias,

com amor...

AGRADECIMENTOS

Queremos agradecer a Deus, pois sem ele nada seria possível, nossas famílias que nos apoiam em todas decisões, nossos colegas de trabalho que sempre nos ajudam e ao IFF por nos proporcionar recursos financeiros e materiais para o desenvolvimento deste trabalho.

Integração Contínua é uma atitude, não uma ferramenta.

James Shore

RESUMO

Este trabalho de conclusão de curso apresenta considerações sobre a técnica da Integração Contínua, demonstrando práticas de desenvolvimento e conclusões referentes a integração de sistemas. Neste trabalho foi criado um referencial teórico e implantado um ambiente de Integração Contínua Assíncrona com o objetivo de documentar esta prática em um projeto real. Para experimentação e validação desta técnica, ela foi aplicada em um estudo de caso para demonstrar o uso desta.

PALAVRAS-CHAVE: Integração Contínua, *Build* Automatizado, Servidor de integração, Desenvolvimento de Software

ABSTRACT

This work of course completion presents considerations for the Continuous Integration technique demonstrating development practices and conclusions about how to integrate systems. A theoretical referential was created in this work and implanted an Asynchronous Continuous Integration environment with the goal of documenting about practices in a real project. For this technique experimentation and validation, it was applied in a case study for a demonstration of its use.

KEYWORDS: Continuous Integration, Automated Build, Integration Server, Software Development

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

A atual conjuntura econômica mundial configura um ambiente de extrema competição, onde os mecanismos de diferenciação se interagem, condicionando o desempenho das empresas em seus respectivos mercados. Nesse contexto, a busca da eficiência e eficácia é uma exigência que se impõe a todos os processos e níveis organizacionais presentes nas atividades produtivas. Para não perderem suas fatias de mercado, as empresas tendem a se posicionar mais agressivamente nessa disputa. Nesse caso, a grande organização muitas vezes leva vantagem por estar mais bem estruturada.

À medida em que a competição se torna mais acirrada, maior é a importância dos ganhos de produtividade trazidos pela tecnologia da informação. Essa tendência influencia toda a cadeia produtiva, o que faz prever que qualquer empresa, independente de seu porte ou tipo de atividade, terá que considerar os impactos que seus softwares trarão para seus negócios, seu mercado e sua concorrência. As empresas estão utilizando cada vez mais os softwares como ferramenta de competitividade, com impactos importantes e positivos nos seus negócios, nos mais variados ramos de atividade.

Percebeu-se que com a mudança constante do mercado ou até mesmo nas próprias empresas, os softwares precisam acompanhar estas mudanças, com a maior rapidez possível, garantindo assim qualidade, mantendo confiabilidade, eficiência e escala de produção. Controlar a qualidade de sistemas de software é um grande desafio devido à alta complexidade dos produtos e às inúmeras dificuldades relacionadas ao processo de desenvolvimento, que envolve questões humanas, técnicas, burocráticas, de negócio e políticas. Idealmente, os sistemas de software devem, não só fazer corretamente o que o cliente precisa, mas também fazê-lo de forma segura, eficiente, escalável, flexível e de fácil manutenção e evolução (OLIVEIRA, 2004). Com isso, surge a idéia dos testes automatizados, que fazem com que o custo e o risco do software caiam consideravelmente, aumentando a confiabilidade, pois com estes o desenvolvedor tem certeza que o software faz exatamente o que deveria fazer (MYERS, 2004). 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.

A reprodutibilidade dos testes permite simular identicamente, e quantas vezes for desejada, situações específicas, garantindo que passos importantes não sejam ignorados por falha

humana, facilitando a identificação de um possível comportamento não desejado (MALDONADO; DELAMARO; JINO, 2007).

Além disso, como os casos para verificação são descritos através de um código interpretado por um computador, é possível criar situações de testes bem mais elaboradas e complexas do que as realizadas manualmente, possibilitando qualquer combinação de comandos e operações (MALDONADO; DELAMARO; JINO, 2007). Utilizando testes automatizados, é possível simular centenas de usuários acessando um sistema ou inserir milhares de registros em uma base de dados, o que não é factível com testes manuais.

Todas estas características ajudam a solucionar os problemas encontrados nos testes manuais, diminuindo a quantidade de erros e aumentando a qualidade do software. Como é mais rápido e fácil executar todos os testes a qualquer momento, mudanças no sistema podem ser feitas com segurança, o que aumenta a vida útil e qualidade do produto. Com isso surgiu a idéia da Integração Contínua, que consiste em uma prática em que os desenvolvedores de uma equipe integram seu trabalho pelo menos uma vez por dia, o que consequentemente levará a múltiplas integrações do sistema (FOWLER, 2006), facilitando assim a detecção de erros inesperados (DUVALL; MATYAS; GLOVER, 2007).

1.1 Justificativa do trabalho

A Integração Contínua foi escolhida como tema deste trabalho por se tratar de uma técnica recente, com poucas referências teóricas que abordam especificamente o assunto. Além disso, o desejo de criar um material que se torne uma referência, em português, para que outras pessoas possam estudar sobre o assunto.

1.2 Objetivo

O principal objetivo deste trabalho é, através de um estudo de caso, documentar e implantar a Integração Contínua em um projeto real. Almeja-se, também, fornecer ao leitor deste trabalho, conhecimento técnico sobre a aplicação dos conceitos da Integração Contínua. Desta forma, deseja-se descrever estes conceitos para equipes de desenvolvimento de software que vislumbram os benefícios desta técnica. Ademais, aumentar a confiabilidade do projeto da Biblioteca Digital da RENAPI, visando um projeto mais robusto e menos suscetível a erros.

1.3 Estrutura do trabalho

Este trabalho se divide em 5 capítulos e está estruturado da seguinte maneira:

No capítulo 2 é feita uma fundamentação de conceitos e técnicas necessárias para o entendimento deste trabalho.

No capítulo 3 é feita uma descrição da Integração Contínua, seus componentes, suas práticas e suas vantagens.

Já no capítulo 4, é apresentado um estudo de caso, em que os autores deste trabalho são responsáveis por implantar um ambiente de Integração Contínua, em um projeto real de desenvolvimento de software.

E por fim, o capítulo 5 apresenta os resultados obtidos com o estudo de caso bem como os trabalhos futuros a serem desenvolvidos no processo da Integração Contínua implantado no projeto.

2 CONCEITOS E TÉCNICAS NECESSÁRIAS

Neste capítulo é feita uma revisão de assuntos relacionados, visando embasamento conceitual, para o entendimento deste trabalho.

2.1 Testes manuais

Hoje, com o notável crescimento da demanda e a constante mudança do mercado, e consequentemente as necessidades das empresas, surge a importância dos softwares acompanharem estas mudanças tanto na rapidez, quanto na qualidade. Mas como se garante a qualidade de um software? Um dos caminhos são os testes (PRESSMAN, 1995). Com eles pode-se obter a certeza, do que foi desenvolvido ou modificado, não resultará em nenhuma falha tanto na fase de desenvolvimento quanto na de produção.

Existem dois tipos de testes: o automático e o manual, no qual neste tópico, são explanados os testes manuais. Os testes manuais são aqueles em que pessoas são alocadas especialmente para testar o sistema, na busca por erros, como por exemplo, clicar em todos os links para testar suas respectivas integridades, escrever caracteres especiais em campos que não deveriam aceitá-lo para ver se o sistema realmente detectará esta falha, etc. Imagine um sistema que acesse um banco de dados, por exemplo. O teste deve cobrir a chamada pela interface de usuário, passando pelas camadas de negócio até chegar no banco. Este método, comparado com os testes automatizados, é um processo mais falho por este ser mais lento, de desempenho inferior e por consumir muito mais tempo. Para Pressman (1995), alguns programas, mesmo que pequenos, podem apresentar problemas ao serem testados, por estes possuírem uma grande quantidade de possibilidades a ser executada. Como exemplificado por Pressman (1995), "[...] um programa PASCAL de 100 linhas com um único laço que pode ser executado não mais do que 20 vezes. Há aproximandamente 10^{14} caminhos que podem ser executados!" Com isso, o número de pessoas para realizar testes manuais e ter um retorno satisfatório seria grande, o que consequentemente, aumentaria o custo do projeto.

Outro problema é quando novas funcionalidades são adicionadas ao programa, o que obriga a execução de todos os testes novamente, fazendo com que todo procedimento seja repetido, acarretando um grande desperdício de tempo.

2.2 Desenvolvimento Orientado a Testes

Desenvolvimento Orientado a Testes ou *Test-Driven Development* (TDD), é uma abordagem para desenvolvimento de software que consiste na criação do teste antes do código (*test-first*) e refatoração do mesmo. Sua filosofia pode ser resumida em uma frase: somente escreva código para fazer um teste falho passar (KOSKELA, 2007). O fluxograma do ciclo do TDD é apresentado na Figura 2.1:

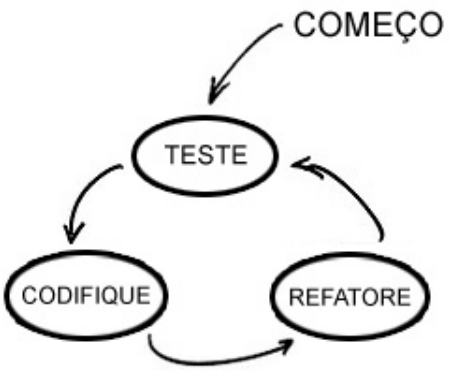


Figura 2.1: Fluxograma do ciclo do TDD

Astels (2003) define o TDD como um estilo de desenvolvimento onde:

- É mantida uma suíte exaustiva de testes;
- Nenhum código entra em produção a menos que possua testes associados que o valide;
- Os testes são escritos primeiro;
- Os testes determinam que código é preciso escrever.

TDD causa grande impacto na qualidade de um software. Beck (2004) nos auxilia no entendimento dessa premissa ao afirmar que:

- Toda vez que alguém toma uma decisão e não a testa, existe uma grande probabilidade de que esta decisão esteja errada;
- Funcionalidades de software que não podem ser demonstradas através de testes automatizados simplesmente não existem;
- Testes nos dão à chance de pensar sobre o que é desejado, independente da forma como a solução será implementada.

De forma geral, ao utilizarmos TDD, deve-se escrever um sólido conjunto de testes, denominado suíte de testes, para a verificação do correto funcionamento do sistema. Esta suíte

deverá, idealmente, cobrir todo o código do sistema, provendo confiabilidade ao processo de desenvolvimento de software. Vale ressaltar, que TDD não garante a obtenção de níveis aceitáveis em certos aspectos do software final, como usabilidade, desempenho, entre outros. Muito disto, se deve ao fato de TDD não conseguir extinguir riscos relacionados com a falta, ou definição equivocada, de requisitos (RIBEIRO, 2010).

2.3 Sistemas de controle de versão

Uma das ações mais importantes para permitir que diversos desenvolvedores trabalhem juntos em um mesmo projeto é utilizar um Sistema de Controle de Versão (SCV), também conhecido como "repositório de código" ou simplesmente "repositório". Existem muitos desses sistemas disponíveis no mercado, tais como Subversion, Rational ClearCase, Mercurial, Git, entre outros.

Devido as suas vantagens, os SCVs são sistemas utilizados em muitos tipos de projetos, desde os de pequeno porte aos de grande porte, mostrando a importância desses sistemas. Por isso, atualmente, é difícil encontrar um projeto profissional que não use um Sistema de Controle de Versão, ainda mais com a quantidade de softwares livres que constam nessa área e a imensa qualidade que eles apresentam.

Os Sistemas de Controle de Versão fornecem um local para armazenamento dos arquivos de um projeto e também controlam as versões dos mesmos, possibilitando que desenvolvedores trabalhem no mesmo arquivo ao mesmo tempo (SHORE; WARDEN, 2008). Imagine, por exemplo, que em um projeto haja uma classe Calculadora com capacidade apenas de listar e calcular o valor da soma entre dois números, e que o programador implemente multiplicação e divisão entre dois números. Ao armazenar o código atualizado com as novas funcionalidades, o repositório grava uma nova versão desta classe, ou seja, a classe Calculadora original não é simplesmente substituída por uma nova, ao invés disso, o novo código é anexado ao antigo.

No SCV também é possível guardar versões dos arquivos, possibilitando a recuperação dos mesmos de forma análoga a um botão de "desfazer". Quando ocorre algum tipo de erro em um arquivo e este é colocado acidentalmente no repositório, este mecanismo nos permite desfazer as alterações do último *commit* (envio das modificações feitas pelo usuário ao repositório de código), e obter o arquivo antigo (SHORE; WARDEN, 2008). Voltando ao exemplo da classe Calculadora, imagine que a primeira versão funcionasse perfeitamente e, ao ser implementada a multiplicação e divisão entre dois números, algum tipo de erro fosse introduzido por falta de atenção. Suponha ainda que este erro só fosse percebido quando a classe já estivesse em produção. Nesse tipo de situação, a possibilidade de controlar versões com um SCV é muito útil, porque permite a obtenção de uma versão anterior rapidamente (a que funcionava) e colocá-la em produção até que a versão mais recente (com problemas) seja corrigida.

2.4 Métodos ágeis

Cansados de fazer software da mesma maneira, um grupo de desenvolvedores se reuniu para discutir sobre formas de aumentar o desempenho de seus projetos. Desta reunião, alguns valores e princípios foram definidos e, com base neles, foi criado o Manifesto Ágil, que pode ser visto na Figura 2.2. "O Manifesto Ágil, criado em 2001, descreve a essência de um conjunto de abordagens para desenvolvimento de software criadas ao longo da última década" (IMPROVE IT, 2006b). Dos valores e princípios do Manifesto Ágil são baseados os métodos ágeis de desenvolvimento de software, que contrastam com a maneira tradicional de se desenvolver.

Manifesto para o Desenvolvimento Ágil de Software

Fazendo isso, estamos descobrindo melhores maneiras de desenvolver software e ajudando outras pessoas a fazê-lo. Através deste trabalho, começamos a valorizar:

- Indivíduos e interações sobre processos e ferramentas
- Trabalhar o software sobre documentações compreensíveis
- Colaboração do cliente sobre a negociação do contrato
- Resposta às mudanças sobre o seguimento de um plano

Isto é, enquanto houver valor nos itens à direita, valorizamos ainda mais os itens à esquerda.

Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, Dave Thomas

Os autores acima, 2001
Esta declaração pode ser copiada livremente, sob qualquer forma, exceto sua integridade através deste aviso.

Figura 2.2: Manifesto Ágil. Fonte: (SHORE; WARDEN, 2008)

Dos métodos ágeis mais populares, como Scrum, Lean, Crystal, entre outros, uma das metodologias de desenvolvimento que mais se popularizou foi a *Extreme Programming* (XP). Esta, criada por Kent Beck em 1999, é amplamente utilizada em inúmeros projetos ao redor do mundo.

O XP é composto por valores e práticas, e uma destas práticas é a Integração Contínua, tema abordado neste trabalho. A Integração Contínua é uma técnica que foi amplamente difundida com os métodos ágeis (DUVALL, 2007), o que não quer dizer que esta surgiu e somente se encaixa com os métodos ágeis. Duvall (2007) relata que já trabalhou em projetos que usavam o ciclo de vida "Cascata" e, mesmo assim, a equipe utilizava Integração Contínua.

Outros métodos ágeis não possuem como prática a Integração Contínua, porém, são facilmente adaptáveis. O Scrum, por exemplo, não tem a Integração Contínua como prática nativa, mas é perfeitamente adaptável.

3 INTEGRAÇÃO CONTÍNUA

Neste capítulo é apresentado o referencial teórico sobre a Integração Contínua. São mostrados os tipos de integração, os componentes de um ambiente de Integração Contínua, o *script* que é responsável por integrar o sistema, as práticas e vantagens da Integração Contínua.

3.1 O que é Integração Contínua?

Integração Contínua é um processo que consiste na integração do código fonte pelo menos uma vez ao dia, mantendo a consistência na base de código ao final de cada integração. Fowler (2006) definiu o processo da seguinte maneira:

> 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.

Existem duas formas de executar a integração: a Integração Contínua Síncrona e a Assíncrona.

3.1.1 Integração Contínua Síncrona

Este método de integração, consiste quando o desenvolvedor integra, por vez seu trabalho, compelindo aos outros desenvolvedores esperarem pelo término da integração corrente, para que possam integrar seus respectivos trabalhos (IMPROVE IT, 2006a). Por este motivo, nem todos os projetos podem utilizar este método, pois ele exige que os desenvolvedores trabalhem juntos, normalmente em uma mesma sala, garantindo assim, que só um desenvolvedor integre seu trabalho por vez (IMPROVE IT, 2006a).

Como o desenvolvedor é obrigado a esperar que seu trabalho seja integrado sem apresentar falhas, isto, certamente, evita que novas tarefas sejam exercidas pelo mesmo, sem que a

integração corrente termine. Isto faz com que o desenvolvedor acompanhe detalhadamente todos os testes em tempo de execução, sabendo de imediato se a performance destes está aceitável, consertando-os se necessário. Como consequência disto, nenhum *commit* indevido permanecerá por muito tempo no Sistema de Controle de Versão, visto que o desenvolvedor irá desfazer as alterações que foram enviadas para o repositório caso a integração falhe.

Os componentes deste modelo de integração são representados de acordo com a Figura 3.1.

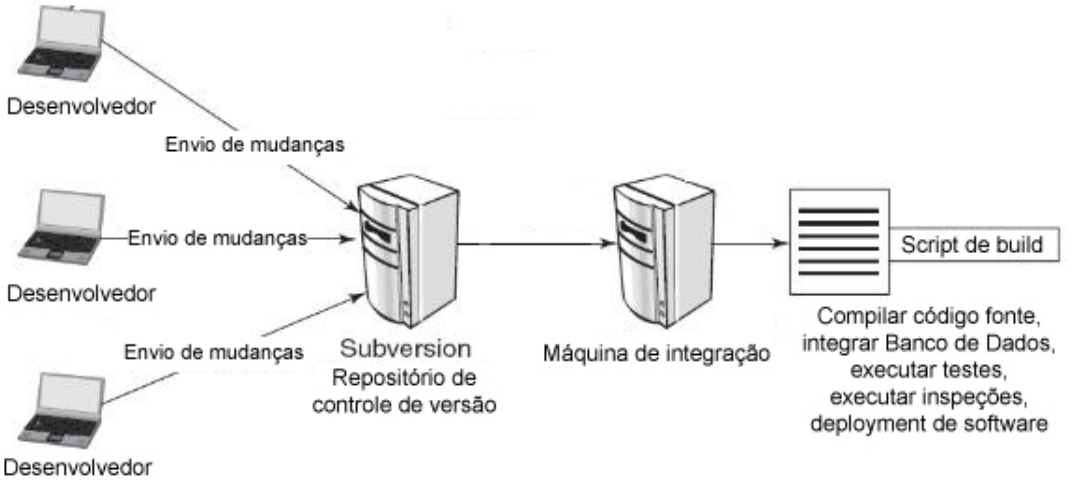


Figura 3.1: Componentes do modelo de Integração Contínua Síncrona. Adaptado de (DUVALL; MATYAS; GLOVER, 2007)

3.1.2 Integração Contínua Assíncrona

Preferencialmente usada em ambientes em que a equipe está geograficamente distribuída – como em projetos *open source*, por exemplo (IMPROVE IT, 2006a). A Integração Contínua Assíncrona consiste basicamente em utilizar um servidor de integração para auxiliar a tarefa de integrar o sistema, ou seja, o processo é feito automaticamente, sem a intervenção direta do desenvolvedor, cujo a única tarefa é submeter as modificações para o SCV, sendo este, monitorado pelo servidor de Integração Contínua, que inicia toda a integração.

Um ponto negativo encontrado nesse modelo de integração refere-se ao fato de uma alteração submetida pelo desenvolvedor que apresentou falhas durante a integração poder criar inconsistência no repositório, uma vez que o responsável por ela pode não estar apto no momento para corrigir tais problemas de forma imediata.

Outro ponto negativo neste modelo trata-se do fato de inúmeras vezes o desenvolvedor só receber a notificação de falha quando já está envolvido em outra tarefa. Isso aumenta o tempo gasto na correção das falhas pelo fato de o desenvolvedor ser compelido a recapitular o contexto mental da tarefa anterior.

Os componentes deste tipo de sistema são representados de acordo com a Figura 3.2.

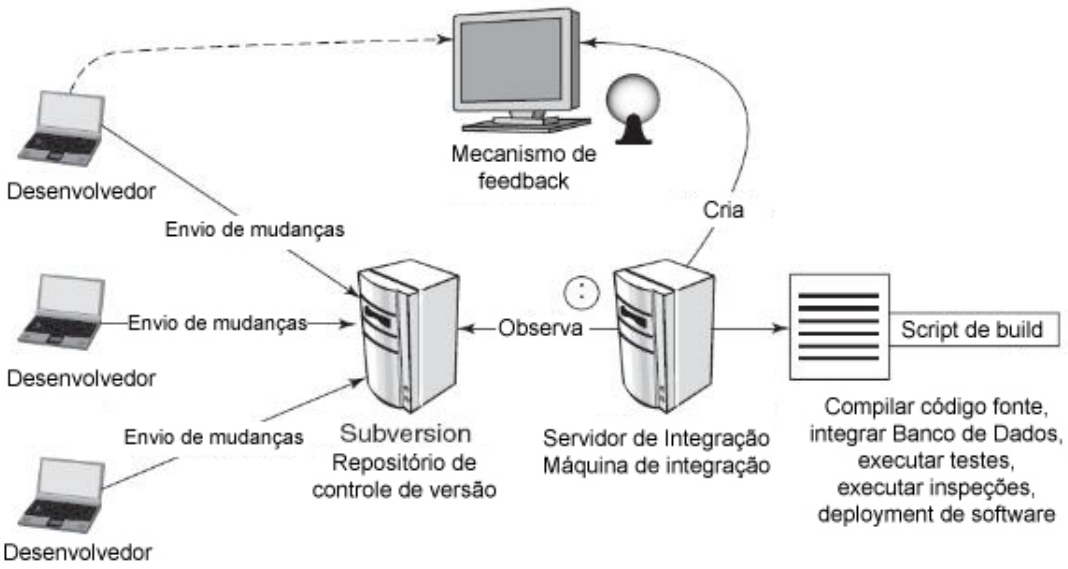


Figura 3.2: Componentes de um sistema de Integração Contínua assíncrona. Adaptado de (DUVALL; MATYAS; GLOVER, 2007)

3.2 Componentes de um sistema de Integração Contínua

Para se obter um sistema de Integração Contínua são necessários vários componentes, em que cada um deles possui sua respectiva responsabilidade. Nas seções seguintes são explicados detalhadamente cada um desses componentes.

3.2.1 Desenvolvedor

O desenvolvedor é a peça fundamental em um sistema de Integração Contínua. Ele é o responsável por realizar as alterações no código fonte do sistema, bem como submeter tais alterações para um Sistema de Controle de Versão, com o intuito de disparar o *script* responsável pela integração do sistema, caracteriznado um cenário de Integração Contínua Assíncrona. De outra forma, o desenvolvedor pode executar este *script* manualmente, caracterizando um cenário de Integração Contínua Síncrona.

3.2.2 Sistema de Controle de Versão

Para um sistema de Integração Contínua, o uso de um Sistema de Controle de Versão é obrigatório, sendo um dos itens mais importantes para todo o processo. O Sistema de Controle de Versão assume tal importância em um sistema de Integração Contínua devido ao fato de que

todo o processo é executado através da raiz do repositório. Os SCVs usualmente trabalham com uma *baseline*, ou seja, uma linha principal onde se encontra o código fonte, e é justamente sobre essa linha que todo o processo de integração será executado.

3.2.3 Servidor de Integração Contínua

O servidor de Integração Contínua é o responsável por executar o *script* de integração, também chamado de *build*, quando for encontrada alguma modificação no código fonte que estiver no repositório. Na Integração Contínua Assíncrona, o servidor fica monitorando o repositório do sistema à procura de alterações, coisa que não acontece no modelo de Integração Contínua Síncrona, pelo fato da integração ser feita manualmente.

Essa inspeção que o servidor de Integração Contínua faz, é realizada de tempos em tempos, ou seja, ela ocorre com certa periodicidade. Caso haja alguma modificação no Sistema de Controle de Versão, o servidor irá recuperar os arquivos do repositório e irá rodar o *script* de *build*.

Uma vantagem bastante expressiva nos servidores de Integração Contínua é a capacidade que eles têm de prover uma interface em que os resultados dos *builds* são publicados (FOWLER, 2006). A Figura 3.3 apresenta o Buildbot[1], um servidor de Integração Contínua, exibindo o resultado da integração de três *builds*, os status destes, as atividades que estão sendo executadas naquele exato momento e as atividades que já foram executadas.

É recomendável a utilização de um servidor de Integração Contínua quando se deseja automatizar o processo de integração do sistema, porém esta utilização não é obrigatória.

Seguindo a mesma linha dos Sistemas de Controle de Versão, existem muitos servidores de Integração Contínua gratuitos e *open source*, o que viabiliza a implantação de um sistema de integração. Outra recomendação feita é que se tenha uma máquina separada, chamada de máquina de integração, em que todo o processo irá ocorrer. Ela possuirá um servidor de Integração Contínua, que por sua vez irá fiscalizar o Sistema de Controle de Versão em busca de modificações.

3.2.4 *Script* de *build*

"Um *build* é muito mais do que compilar (ou variações das linguagens dinâmicas). Um *build* deve consistir de compilação, testes, inspeção, e *deployment* – entre outras coisas. Um *build* atua como um processo para colocar código fonte junto e verificar se o software trabalha como uma unidade coesiva." (DUVALL; MATYAS; GLOVER, 2007).

[1] http://buildbot.net

Figura 3.3: Buildbot exibindo dados da integração de um sistema

O *build* é um único *script*, ou conjunto deles, que irá compilar o código fonte do sistema, executar a suíte de testes, realizar a inspeção do código e fazer o *deployment* do software, ou seja, gerar software executável com a última versão do código fonte.

A seção 3.3 - O *script* de *build* apresenta em maiores detalhes as etapas deste *script*.

3.2.5 Mecanismo de *Feedback*

Um dos princípios da Integração Contínua é o *feedback* rápido.Quando uma suíte de teste é executada e caso algum teste falhe, a equipe de desenvolvimento é avisada instantaneamente através do *feedback* (DUVALL; MATYAS; GLOVER, 2007). Recebendo tal informação de falha, o responsável pelo código deve solucionar o problema. O feedback *também* pode ser utilizado para informar a execução do *build* com sucesso, as mudanças no último *build*, os arquivos criados, entre outras coisas.

Em se tratando de Integração Contínua existem várias formas de mecanismo de *feedback*. Pode ser utilizado e-mail, que é a forma mais comum e também o *Short Message Service* (SMS), serviço de envio de mensagens para celulares. Ademais, pode-se obter *feedback* com

Really Simple Syndication (RSS), *plugins* de navegadores, mensageiros instantâneos como por exemplo o Google Talk, *widgets*, entre outros.

Além destes mecanismos citados acima, também é comum a utilização de *Ambient Orb* (Orb de Ambiente), que se trata de uma bola que pode ser configurada para ter uma determinada cor específica de acordo com o resultado do *build*. Outro item que também pode ajudar no *feedback* é o som, que pode ser personalizado para executar um determinado arquivo quando o *build* é feito com sucesso. Caso o *build* falhe, o arquivo a ser executado será outro.

Outro mecanismo utilizado são as *Lava Lamps* (Lâmpadas de Lava), as quais são muito conhecidas no mercado como itens decorativos. Elas podem ser configuradas para acenderem de acordo com a situação do *build*. Usualmente formando um par, as lâmpadas costumam ter cada uma delas uma determinada cor. A lâmpada que irá representar o *build* com sucesso terá a cor verde e a lâmpada que demonstrará falha no *build* terá cor vermelha. Enquanto o *build* funcionar, a lâmpada vermelha não irá acender e a verde permanecerá acesa, acontecendo o inverso se o *build* falhar, o que consequentemente irá apagar a lâmpada verde e acender a vermelha. A Figura 3.4 apresenta um exemplo do uso das Lâmpadas de Lava.

Figura 3.4: Lâmpadas de Lava mostrando a situação do *build*. Fonte: (PRAGMATIC AUTOMATION, 2007)

Esses mecanismos de *feedback*, como o *Ambient Orb*, as Lâmpadas de Lava e os sons criam um ambiente de trabalho engraçado e personalizado, o que acaba motivando os desenvolvedores e prova que é possível desenvolver software de qualidade em um ambiente descontraído e divertido.

3.3 O *script* de *build*

Essa seção apresenta em detalhes o *script* de *build* bem como seus respectivos tipos e mecanismos.

3.3.1 Botão de Integração

Quando se trata de Integração Contínua, é comum alguns autores citarem o termo "botão da integração". O conceito deste, é a criação de um botão virtual, que a um toque, faça com que determinadas etapas sejam executadas e, como resultado final, produzam software funcional. A Figura 3.5 demostra as etapas que são realizadas ao toque do botão da integração.

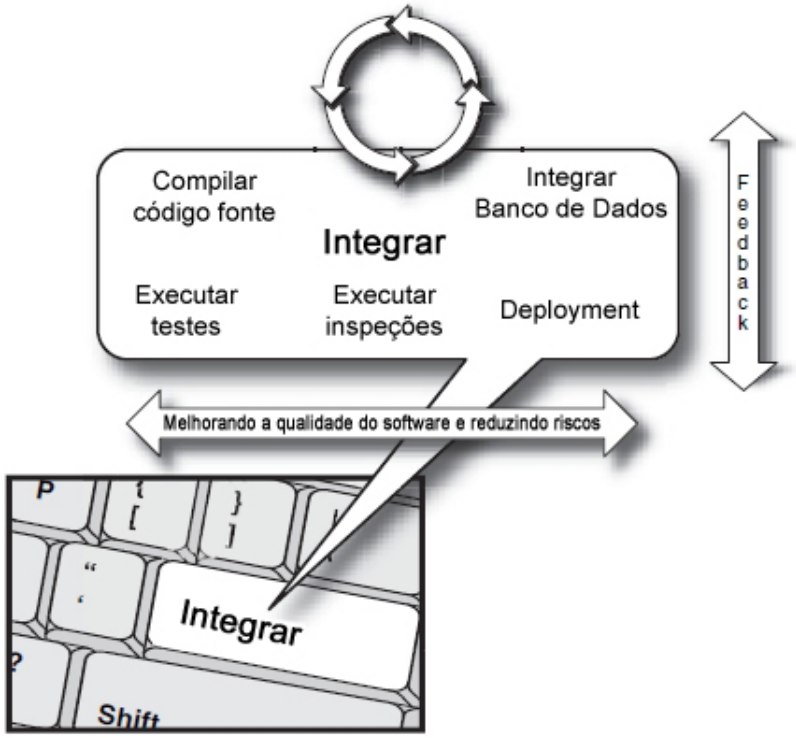


Figura 3.5: Botão da integração e suas etapas. Adaptado de (DUVALL; MATYAS; GLOVER, 2007)

3.3.1.1 Compilação de código fonte

A compilação é uma das etapas mais básicas do *script* de *build*. Compilação se trata do processo de transformação do código fonte em linguagem que a máquina irá entender e executar (DUVALL; MATYAS; GLOVER, 2007).

3.3.1.2 Integração de Banco de Dados

A Integração de Banco de Dados é referente a todo o processamento relacionado ao Banco de Dados do sistema, caso o sistema faça uso de um deles. A Integração do Banco de Dados consiste na execução dos *scripts* de criação e remoção de bancos e tabelas, aplicação de *procedures* e *triggers*, inserção de dados nas tabelas, entre outras coisas.

De acordo com Duvall, Matyas e Glover (2007), muitas pessoas acham que a Integração do Banco de Dados é feita como se fosse um processo independente, separado da integração do restante do sistema. Como o Banco de Dados é uma parte que compõe toda a aplicação, o correto é que a cada modificação em um desses *scripts*, o *build* seja executado para verificar se as modificações feitas não geram inconsistência no sistema.

3.3.1.3 Execução de testes

Trata-se de uma das etapas mais importantes do processo de Integração Contínua, uma vez que, ela é responsável pela verificação do correto funcionamento das partes integrantes do sistema, assim como dos relacionamentos entre elas.

Fowler (2006) apresenta o conceito de *build* auto-testável, onde defende que uma falha ou erro na execução de qualquer um dos testes integrantes da suíte de testes do sistema deve causar a falha de todo o processo de *build*. Essa abordagem, que é permeada de grandes benefícios para o processo de desenvolvimento de software, vem confirmar a indispensabilidade dos testes para o processo de Integração Contínua.

Outra importância acrescentada por Duvall, Matyas e Glover (2007) à essa etapa é a de conferir confiabilidade ao que está sendo desenvolvido. Segundo ele, sem testes automatizados, é difícil para desenvolvedores ou colaboradores do projeto terem confiança nas mudanças do software.

3.3.1.4 Execução de inspeções

A etapa de execução de inspeções é caracterizada pelo processo de análise do código que foi escrito (DUVALL; MATYAS; GLOVER, 2007). Esta técnica, muita das vezes feita manualmente, é realizada com um desenvolvedor da equipe criticando e fazendo sugestões ao código que outro desenvolvedor escreveu.

Essa tarefa de inspeção manual de código pode acarretar em desperdício de tempo, visto que seguir padrões são cansativos, dolorosos e difíceis de serem acompanhados.

Para evitar a dor de cabeça causada pela inspeção manual de código, existem ferramentas gratuitas que fazem o processo automaticamente. Elas têm as funcionalidades de verificar

código duplicado, quantificar o número de linhas não comentadas e até mesmo fazer um *build* falhar caso algum padrão não tenha sido obedecido.

3.3.1.5 *Deployment*

Deployment é o processo que torna possível a entrega do software funcional ao cliente, com as últimas modificações ocorridas, em qualquer lugar, a qualquer hora e com o mínimo de esforço, viabilizando um ambiente de teste para a avaliação do cliente (DUVALL; MATYAS; GLOVER, 2007). Esta entrega é denominada, no método ágil XP, como *release*, e é normalmente feita a pedido do cliente ou negociada no início do desenvolvimento (será entregue ao cliente um software funcional toda semana, por exemplo).

O *deployment* pode ser uma etapa do *script* de *build*, mas normalmente não é, por dois principais motivos: o primeiro, por ser um processo lento que abrange outras etapas, como compilação do código fonte, integração do banco de dados, execução dos testes, execução de inspeções, entre outras coisas; o segundo é pelo fato de que cada *commit* feito para o SCV, faz com que o *script* de *build* seja executado. Visto que, muitos destes *commits* são efetuados por dia, não entende-se aqui a nescessidade de se ter inúmeras *releases* por dia, já que, elas são a pedido do cliente ou programadas, como já foi mencionado no parágrafo anterior, além de evitar, gastos desnecessários de processamento e de tempo.

3.3.1.6 Documentação

Um dos objetivos do *build* pode ser a criação automática da documentação do software. Para Duvall, Matyas e Glover (2007), a melhor documentação acaba sendo o próprio código fonte, quando este apresenta-se de forma clara e concisa atráves de nomes de métodos, variáveis, classes, entre outros. Todavia, somente este código bem escrito, nem sempre é o suficiente. Determinados projetos necessitam de diagramas de UML, documentação de *Application Programming Interface* (API), etc. Gerar esta documentação manualmente é trabalhosa, pois qualquer modificação no código costuma alterar a documentação. Porém, existem ferramentas que criam esta documentação automaticamente, e podem ser embutidas no *script* de *build*, obtendo sempre a documentação atualizada e, consequentemente, evitando desperdício de tempo.

3.3.2 Tipos de *build*

O *build* pode ocorrer em 3 níveis de hierarquia:

- Privado - Neste tipo de *build*, a integração é feita na máquina do desenvolvedor. O desenvolvedor, ou o par (em práticas ágeis), roda um *build* privado antes de enviar o novo código para o repositório, para integrar as mudanças dele com as possíveis mudanças que outros desenvolvedores da equipe tenham feito. As etapas para executar um *build* privado são:

  1. Pegar do repositório o código a ser alterado
  2. Fazer as alterações necessárias no código
  3. Pegar as últimas mudanças do repositório
  4. Rodar o *build* privado
  5. Comitar o código modificado para o repositório

  A Figura 3.6 apresenta o esquema das etapas para se executar um *build* privado.

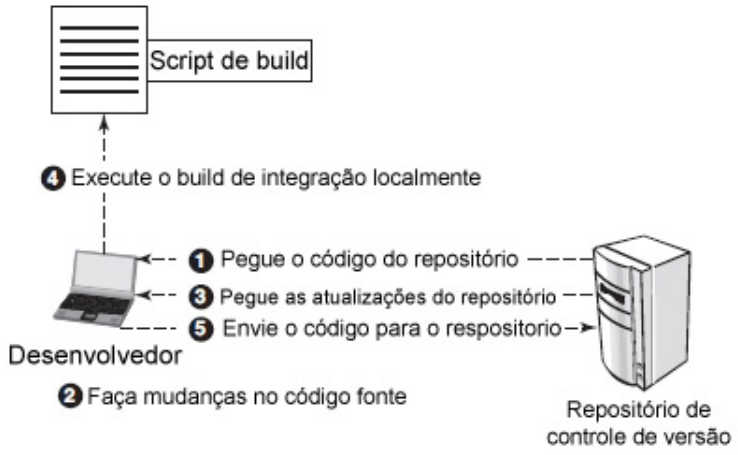


Figura 3.6: Esquema das etapas para se executar um *build* privado. Adaptado de (DUVALL; MATYAS; GLOVER, 2007)

- Integração - Este *build* de Integração tem como objetivo integrar todas as mudanças ocorridas no repositório. Ele integra todas as alterações que o desenvolvedor fez, com o código que está localizado no Sistema de Controle de Versão. O ideal é que esse *build* seja executado em uma máquina separada.

- Release - Um dos pilares da Integração Contínua é a possibilidade de se criar software funcionando a qualquer momento com apenas um comando. Este é o objetivo deste *build*, entregar software funcionando para o usuário final, para que ele possa utilizar no ambiente de trabalho dele. Este *build* usualmente é executado com certa periodicidade específica, por exemplo, um mês.

3.3.3 Mecanismos de *build*

Nem todos os *builds* são executados da mesma forma. Eles são executados levando-se em consideração o seu propósito e frequência. Alguns *builds* podem ter sua execução feita manualmente por realizar a execução de testes mais demorados, por exemplo. Outros podem ser disparados automaticamente, bastando somente acontecer alguma mudança no Sistema de Controle de Versão. Portanto, existem alguns mecanismos de *build*, como descrito abaixo:

- Sob-demanda - O processo de *build* é disparado manualmente pelo desenvolvedor.
- Programado - Neste mecanismo, o *build* é disparado por algum evento pré-definido, como um intervalo de tempo. O *build* será executado independentemente de ter ocorrido, ou não, alguma alteração no repositório de código.
- À procura de mudanças - Um processo do servidor de Integração Contínua fica sendo executado durante um intervalo de tempo regular em busca de mudanças no repositório do software. Caso haja alguma alteração no código existente, o processo de *build* é disparado automaticamente.
- Guiado por evento - Mecanismo muito parecido com o de procura por mudanças, porém, o processo de verificação de mudanças no código é feito pelo próprio repositório. Se ele detecta uma mudança, o processo de *build* é inicializado.

  A Figura 3.7 mostra quais mecanismos cada *build* pode executar.

Tipo de build	Mecanismo de build
Privado	Sob-demanda
Integração	Sob-demanda, programado, a procura de mudanças, guiado por evento
Release	Sob-demanda, programado

Figura 3.7: Relação entre mecanismos e tipos de *build*. Fonte: (DUVALL; MATYAS; GLOVER, 2007)

3.4 Práticas da Integração Contínua

Neste item é explicado algumas práticas comuns da Integração Contínua.

3.4.1 Manter um único repositório de código

Projetos de software, usualmente, são compostos por várias pessoas. Cada pessoa da equipe irá criar inúmeros arquivos que, em conjunto, resultarão no produto final. Ao longo do projeto, a quantidade de arquivos criados irá crescer rapidamente, o que torna penosa a tarefa de coordenar manualmente todo o trabalho de uma equipe. Para tal serviço é que são utilizados os Sistemas de Controle de Versão, os quais também são chamados de repositórios.

Em um ambiente de Integração Contínua é muito importante a utilização de um Sistema de Controle de Versão, com o intuito de manter todos os arquivos necessários juntos para a construção do sistema. Um problema que acontece muitas vezes nas equipes é que elas não colocam todos os arquivos no repositório, pensando que somente devem estar lá arquivos de código fonte. Logo, vê-se a importância de um sistema para centralização dos arquivos necessários para o sistema.

3.4.2 *Build* automatizado

Gerar software funcional com o código que está no repositório requer a realização de várias etapas como a compilação de código fonte, execução de *scripts*, criação do banco de dados, entre outras. Realizar estas etapas manualmente pode se tornar uma tarefa entediante e inconveniente, visto que estas podem, e devem, ser automatizadas. O patamar ideal é que o software possa ser criado com apenas um simples comando, através de um *script* de *build* automatizado.

Quando é dito que um comando deve criar software funcional, deve-se levar em conta, que todos os arquivos necessários para criar o sistema estejam no *script* de *build*. Então, qualquer pessoa deve ser capaz de pegar uma máquina virgem, obter os arquivos do repositório, executar um único comando e ter um sistema rodando na máquina (FOWLER, 2006). O que acontece muitas vezes é que as equipes esquecem de colocar tudo que é essencial no *build*, fazendo com que, para criar o software final, seja preciso realizar determinadas tarefas, antes ou depois da execução do *script* de *build*.

Fowler (2006) também diz que o build só deve compilar os arquivos que foram modificados. Com isso, o tempo de execução de um *build* pode ser reduzido consideravelmente, o que é uma prática aconselhável.

Quando se diz em *script* de *build*, imagina-se que ele é apenas um único *script* que faz todo o serviço, e nem sempre é isso, ele também pode ser um conjunto de *scripts*, onde cada um faz uma determinada tarefa. Alguns projetos, por exemplo, possuem testes que são mais demorados, como por exemplo, testes de carga e performance. Esperar por testes demorados em um *build*, obviamente, não é uma boa prática, o que condiz com a prática do XP "*Ten minute*

build" (Build de dez minutos) (BECK, 2004). Por tal motivo, nem sempre é aconselhável que esses testes fiquem no *script* de *build*. Por isso, é normal que diferentes tipos de *build* sejam criados e que cada um deles execute diferentes tarefas, sendo cada um executado em determinados momentos do projeto.

3.4.3 *Build* auto-testável

Uma boa forma de capturar defeitos mais rapidamente e eficientemente é incluir testes automatizados no processo de *build* (FOWLER, 2006). Levando-se em consideração que um dos princípios da Integração Contínua é o *feedback* instantâneo, uma suíte de testes abrangente é a melhor forma de verificar se o sistema está integrado corretamente.

Com o avanço do método ágil *Extreme Programming* (XP) e a prática *Test Driven Development* (TDD), o desenvolvimento de testes ficou muito mais viável e popular. Apesar destas práticas serem recomendadas, elas não são obrigatórias, pois o que realmente importa para realizar a Integração Contínua é o código coberto por testes, independente da técnica com que eles foram construídos.

Segundo Fowler (2006), para que um *build* seja auto-testável, a falha de um teste deve causar a falha do *build*. Daí surge a importância do *build* auto-testável ter uma boa cobertura de testes.

3.4.4 Todos enviam alterações para o repositório todos os dias

De acordo com a definição de Fowler (2006) acerca da Integração Contínua, os desenvolvedores devem enviar código para o repositório, ao menos uma vez por dia, ou seja, quanto mais *commits* forem feitos pelos desenvolvedores, melhor será o processo de integração.

Enviar código frequentemente para o repositório é uma boa prática, porque reduz conflitos entre os códigos. O que costuma acontecer é a nescessidade de mais de um desenvolvedor ter que trabalhar no mesmo arquivo, o que pode gerar conflitos, pois o código pode ser alterado de tal forma que os testes falhem ao serem executados. Embora esses conflitos aconteçam, quanto mais rápido eles forem descobertos, mais rápido eles serão corrigidos. Imagine que o tempo entre o envio de arquivos para o repositório seja curto. Caso haja conflito, haverão poucos lugares para verificar aonde o problema está. Agora imagine que os arquivos sejam enviados para o repositório com pouca frequência. A quantidade de lugares em que o *bug* pode estar escondido aumenta consideravelmente, dificultando sua resolução.

Além disto, envios frequentes ao repositório encorajam aos desenvolvedores a quebrar seus códigos em pequenas partes de poucas horas cada (FOWLER, 2006). Isto, que combina fielmente com o princípio do método XP, *Baby Steps* (Passos de bebê) (IMPROVE IT, 2006c),

faz com que o código seja feito gradativamente e com segurança, evitando conflitos e criando a sensação de progresso.

3.4.5 Todo *commit* deve atualizar a *baseline* na máquina de integração

Independente do modelo de Integração Contínua que a equipe estiver utilizando, todo *commit* efetuado deve gerar o sistema com as última modificações encontradas na *baseline* (local do repositório que contém as últimas alterações do sistema), em uma máquina dedicada, a qual é chamada de máquina de integração.

Quando o desenvolvedor altera o código em que está trabalhando, a intenção é que as modificações, que funcionaram em uma máquina, funcionem também nas máquinas dos demais desenvolvedores do projeto (SHORE; WARDEN, 2008). Por este motivo, Duvall, Matyas e Glover (2007) salienta que o ambiente de trabalho seja exatamente igual ao ambiente de produção. Todavia, nem sempre é isso o que acontece. Muitas vezes, o código que funciona na máquina do desenvolvedor não funciona em outros ambientes. É daí que surge a idéia da máquina de integração. Esta contém um ambiente bom e puro que serve para ratificar que as alterações feitas irão funcionar em qualquer máquina do ambiente de desenvolvimento.

Uma recomendação feita acerca do uso da máquina de integração é que esta só deve ser utilizada para integrar o sistema e não ser usada para correção de problemas (SHORE; WARDEN, 2008). A correção deve ser feita na máquina do desenvolvedor pois, provavelmente, ele deve ter esquecido de realizar o *commit* de um ou mais arquivos. Se o problema for corrigido na máquina de integração, os desenvolvedores que pegarem o código do repositório também irão encontrar os mesmos problemas.

3.4.6 Mantenha o build rápido

Segundo Shore e Warden (2008), o problema mais comum que as equipes que praticam a Integração Contínua enfrentam, são os *builds* lentos e para Fowler (2006), o principal foco da Integração Contínua é prover *feedback* rápido. Logo, é impossível obter *feeedback* instantâneo com *builds* demorados. Por isso, existe a recomendação de sempre se manter o *build* rápido.

Shore e Warden (2008) também diz que, sempre que possível, manter o tempo do *build* abaixo de 10 minutos. Aconselha-se este período de tempo, pois é um intervalo em que os desenvolvedores podem conversar entre si sobre o projeto, planejar as próximas tarefas a serem feitas ou, até mesmo, aproveitar o tempo para descansar. Ademais, quanto mais rápido o *build* for, maior o tempo que os desenvolvedores passarão implementando o sistema.

O que faz com que muitos *builds* fiquem lentos, na maioria dos casos, é a execução de testes demasiadamente demorados. As vezes, o mais aconselhável é a criação de outros *scripts*

que possam executar esses testes mais demorados separadamente (SHORE; WARDEN, 2008). Por exemplo, testes de carga, desempenho, estabilidade, são testes lentos e não precisam ser executados em cada integração. Separar os testes demorados do restante do *build*, irá reduzir drasticamente o tempo gasto na integração, fazendo com que seja possível atingir o limiar de 10 minutos do *build* (BECK, 2004).

3.4.7 Teste em uma cópia do ambiente de produção

Quando testes são criados, um dos seus propósitos, é evitar que erros apareçam durante a fase de produção do sistema. Por tal motivo, é necessário fazer os testes em uma cópia do ambiente de produção.

Com ambientes de testes e produção diferentes, erros que podem acontecer em um ambiente, podem não acontecer no outro e vice-versa. Esse é o objetivo de testar em abientes parecidos. A chance de um erro imprevisto acontecer em produção é reduzido consideravelmente (FOWLER, 2006).

Idealmente, o ambiente de teste deve ser idêntico ao ambiente de produção, porém, isso nem sempre é possível. Então, o objetivo é chegar o mais próximo possível do clone do ambiente de produção. Para isso, deve-se usar a mesma versão do sistema de banco de dados, sistema operacional, bibliotecas e até mesmo o mesmo IP, *hardware*, etc.

3.4.8 Torne fácil para qualquer pessoa o acesso ao último executável

Segundo Fowler (2006) uma das partes mais difíceis na construção de um software, é saber se o que está sendo desenvolvido é realmente o que o cliente espera. Conclui-se que para ajudar neste trabalho, o desenvolvedor deve manter um lugar com o executável da última versão do sistema funcionando, para que possa ser demonstradas, testadas ou simplismente vistas pelo cliente as últimas modificações daquela semana (FOWLER, 2006). Vale ressaltar também, que qualquer um pode acessar o referido executável, sem ter a necessidade, de ter acesso ao *build* ou até mesmo possuir algum conhecimento de informática.

3.4.9 Todos podem ver o que está acontecendo

Para Fowler (2006), o estado do *build* é uma das coisas mais importantes para comunicação do processo de integração. Dependendo da situação do *build*, a equipe de desenvolvedores sabe quais atitudes deve tomar.

Baseado na informação do *build*, muitos servidores de Integração Contínua possuem uma interface na qual qualquer pessoa pode acessar e ver o andamento do *build*, se ele está em

execução, se falhou ou se foi executado sem erros. Já que a situação do *build* está disponível através do servidor, não há necessidade de estar no mesmo local da equipe para ver seu status, ou seja, pessoas dispersas geograficamente do local da equipe poderão acompanhar o *build*. A Figura 3.8 mostra o Hudson[2], um servidor de Integração Contínua, mostrando o status dos *builds* de um determinado projeto.

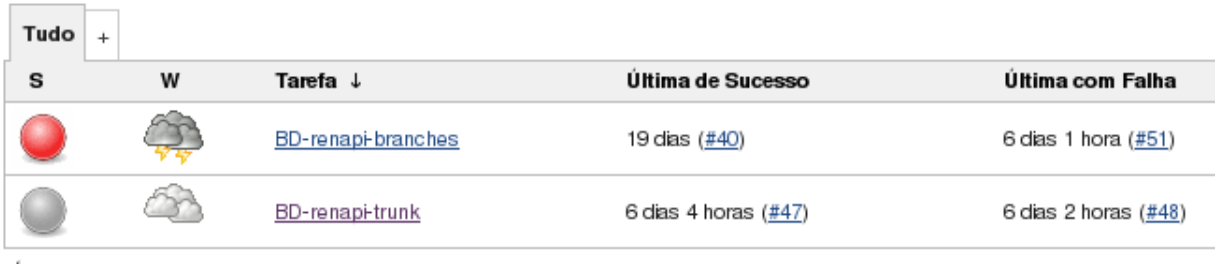


Figura 3.8: Status dos *builds* no Hudson.

3.4.10 *Deploy* automático

Segundo Fowler (2006), é importante ter *scripts* que permitam a implantação da aplicação dentro de qualquer ambiente facilmente. Utilizando *deploy* manual, a implantação do sistema pode não ser aplicada tão facilmente, como disse Fowler. Fazendo uso deste tipo de *deploy*, o responsável deve seguir uma série de etapas para disponibilizar a nova aplicação no ambiente de produção. Exemplificando, o responsável terá que acessar o servidor de produção, parar o servidor *Web*, fazer *backup* da aplicação que está em uso, colocar a nova aplicação e inciar o servidor *Web*. Agora imagine que haja mais etapas a serem realizadas do que as que foram citadas anteriormente. A probabilidade do responsável esquecer alguma etapa aumenta consideravelmente.

Já com o *deploy* automático, a chance de alguém esquecer alguma configuração é extinta, ao passo que no *deploy* manual é necessário realizar várias etapas manualmente. Com a execução de um *script*, as tarefas que eram executadas manualmente, passam a ser automatizadas, reduzindo a chance de erros na hora do *deploy*. Além disso, o processo de *deploy* se torna menos trabalhoso, encorajando os desenvolvedores a fazerem mais *deploys* , mantendo a versão em produção sempre atualizada.

3.5 Vantagens da Integração Contínua

Neste item serão apresentado os principais benefícios que a prática da Integração Contínua pode proporcionar.

[2]http://hudson-ci.org

3.5.1 Defeitos são encontrados e corrigidos o quanto antes

"Visto que a Integração Contínua integra e roda testes e inspeções diversas vezes ao dia, existe uma grande chance de os defeitos serem descobertos quando eles forem introduzidos" (DUVALL; MATYAS; GLOVER, 2007).

Exemplificando o que foi dito acima, imagine que um sistema funcione corretamente. Após um determinado *commit*, o sistema não é mais integrado corretamente e o *build*, consequentemente, falha. A partir disso, existe uma grande possibilidade de o código que está quebrando o *build* estar neste último *commit*, visto que, foi a partir dele que o sistema passou a não ser integrado corretamente.

Logo, a quantidade de possíveis lugares onde o defeito pode estar contido é baixa, facilitando a detecção e a resolução do mesmo, contribuindo para a integridade do software criado.

3.5.2 Feedback instantâneo

Segundo Fowler (2006), o principal foco da integração contínua é o *feedback* instantâneo. O *script* de *build* será responsável por ativar o único, ou vários, mecanismos de *feedback* para alertar a equipe de desenvolvedores que a última integração que ocorreu apresenta falhas e deve ser corrigida o quanto antes, visando sempre manter a base de código consistente.

O propósito do *feedback* é criar uma notificação que irá estimular uma ação rápida e precisa. Para isso, é necessário enviar a informação correta, para as pessoas corretas, no tempo correto e da forma correta (DUVALL; MATYAS; GLOVER, 2007). A informação correta ideal é a notificação do status do *build*, junto do resultado dos testes, das inspeções e do *deployment*. As pessoas corretas dependerão do tipo de informação enviada, ou seja, o ideal é que cada pessoa da equipe receba somente a informação que lhe seja pertinente, visto que enviar *feedback* para todos no projeto, usualmente, faz com que a equipe passe a ignorar as informações recebidas (DUVALL; MATYAS; GLOVER, 2007). O tempo certo é o menor intervalo possível entre o aparecimento do erro e o *feedback* à equipe. Receber o *feedback* de um *build* quebrado dias atrás não acrescenta em nada à equipe e ainda pode ocasionar perda de tempo e frustração ao alocar desenvolvedores para corrigir um erro que já havia sido corrigido. A forma correta se resume na escolha de um ou mais tipos de mecanismos de *feedback*. O critério de escolha será da própria equipe, levando-se em consideração as vantagens e desvantagens de cada mecanismo.

Com o *feedback* instantâneo, uma modificação que não foi integrada corretamente com o restante do código é rapidamente descoberta, pois o desenvolvedor irá obter uma mensagem caso o *build* tenha quebrado. Corrigir esse erro torna-se muito mais fácil devido à pequena gama de lugares onde o erro pode estar. Agora, imagine que em um sistema a integração seja feita somente após o término do desenvolvimento. Provavelmente, irão surgir erros e mais erros

de todos os possíveis lugares, o que torna a correção deles uma tarefa trabalhosa e cansativa.

Além do que já foi dito, o *feedback* irá mostrar o status do sistema. Este mostra para a equipe como está a saúde do software. Verificando a integridade dele é exequível analisar possíveis problemas no processo de integração. Uma base de dados que se mantém por muito tempo corrompida, indica que a equipe não está levando o processo tão sério quanto deveria.

3.5.3 Redução de problemas futuros na instalação do software

Segundo Duvall, Matyas e Glover (2007), através da reconstrução e teste de software em um ambiente limpo, ou seja, em um sistema recém-instalado, usando o mesmo processo e *scripts* que foram utilizados no ambiente de desenvolvimento, é possível reduzir problemas futuros na instalação do software. Para exemplificar, supõe-se que uma determinada funcionalidade do sistema tenha como dependência uma pacote chamado "python-dev", e que por coincidência, esse pacote já se encontra instalado no ambiente do desenvolvedor. Logo, como o desenvolvedor desconhece desta dependência, ele não o adiciona na lista de dependências do sistema, e sua nova funcionalidade é eviada para o SCV.

Obviamente, quando esta funcionalidade for instalada no servidor de produção, a nova funcionalidade não funcionará. Por este motivo, que deve-se executar os testes, bem como instalar os módulos do sistema em um ambiente recém-instalado, facilitando a detecção deste problema.

3.5.4 Redução de processos manuais repetitivos

É da tendência do ser humano automatizar processos manuais que são feitos com grande frequência. A escassez de tempo e o retrabalho são as principais causas do processo de automatização. Por isso, a existência de máquinas e processos automatizados, já que estes diminuem os custos e aumentam a velocidade da produção (LACOMBE, 2004).

Neste âmbito, a Integração Contínua visa reduzir processos repetitivos economizando tempo, custos e esforço, através da automatização da integração. Estes processos repetitivos podem ocorrer em todas as atividades do projeto, incluindo a compilação do código, integração do banco de dados, testes, inspeções, implantação e *feedback* (DUVALL; MATYAS; GLOVER, 2007).

Para Duvall, Matyas e Glover (2007), ao automatizar a Integração Contínua, tem-se uma maior capacidade para assegurar o seguinte:

- O processo funciona da mesma maneira toda vez;

- Um processo ordenado é seguido. Por exemplo, pode-se executar inspeções (Análise estática) antes de executar os testes - em seu *scripts* de build;
- A redução do trabalho em processos repetitivos, liberando as pessoas para realizarem trabalhos mais instigantes, de maior valor.

3.5.5 Geração de software executável ao clique de um botão

Quando se fala em geração de software com o clique de um botão, faz-se referência ao botão da integração (Figura 3.5). A partir deste botão virtual, uma série de etapas é executada, sendo que uma delas é responsável pelo *deployment* do sistema, ou seja, a criação do software executável.

Para Duvall, Matyas e Glover (2007), o maior benefício da Integração Contínua é conseguir gerar software executável a qualquer momento, visto que é o item mais "tangível" para pessoas que não estão envolvidas diretamente com o desenvolvimento do software, como clientes e usuários.

O fato de o software poder ser gerado a qualquer momento, evita que haja atrasos na entrega do mesmo, para que o cliente possa, por exemplo, testá-lo. Projetos que não usam a Integração Contínua, terão que integrar, rodar inspeções e testes, antes da entrega do sistema. Como a integração manual não é um "mar de rosas" e a equipe não integra frequentemente, a quantidade de testes e inspeções que irão falhar será enorme e a integração do sistema não acontecerá corretamente, obrigando a equipe de desenvolvimento a consertar estes defeitos. Em consequência disto, a entrega do software pode atrasar em alguns dias, ou até mesmo meses, devido ao tempo gasto para a correção dos defeitos.

Vale lembrar que "o objetivo é estar tecnologicamente pronto para o lançamento, mesmo que não esteja funcionalmente pronto para o lançamento" (SHORE; WARDEN, 2008). Isso mostra que a Integração Contínua pode prover a criação do software a qualquer momento, mesmo que alguma funcionalidade não tenha sido implementada por completo. Suponha que em um sistema, esteja sendo implementado uma nova funcionalidade de cadastro de pessoas. Se em algum momento for necessário a criação do software, ele estará apto para tal, mesmo que não esteja com a funcionalidade totalmente pronta.

4 ESTUDO DE CASO

Este capítulo apresenta um estudo de caso com a implantação de um ambiente de Integração Contínua no projeto da Biblioteca Digital da Rede Nacional de Pesquisa e Inovação (RENAPI). Este estudo de caso foi idealizado devido ao fato do processo de Integração Contínua realizado no Núcleo de Pesquisa em Sistema de Informação (NSI) do IFF-Campos apresentar problemas, junto com o ideal de melhorar a prática de integração no grupo.

4.1 Ambiente estudado

O estudo de caso deste trabalho foi realizado no NSI. O NSI surgiu em abril de 2002 e desde essa época vem utilizando e estimulando o uso de tecnologias de software livres.

Durante a fase de criação deste estudo de caso, o núcleo contava com 32 bolsistas, subdivididos entre vários projetos. Dois destes projetos estavam diretamente relacionados com este trabalho: o projeto da Biblioteca Digital da RENAPI e o Quali-Ágil.

O portal da Biblioteca Digital da RENAPI visa disponibilizar um acervo bibliográfico digital de maneira a contribuir para a disseminação do material científico e tecnológico produzido na rede de Instituições de Educação Profissional Científica e Tecnológica (EPCT) - como artigos, monografias, dissertações, teses e periódicos, promovendo a disseminação nacional e internacional deste conteúdo, visando colaborar na qualificação do material humano da rede e na disseminação de conhecimento.

O Quali-Ágil é um projeto, que visa o estudo e desenvolvimento de ferramentas e metodologias para aumentar a qualidade de projetos de software que usam metodologias ágeis.

A Biblioteca Digital é um dos projetos mais importantes do NSI. Por tal motivo, ele precisa ser estável e confiável, devido a seu grande porte e visibilidade. Para alcançar tais objetivos, a Integração Contínua é uma boa prática. O papel do Quali-Ágil neste trabalho foi ajudar no gerenciamento e implantação do ambiente de Integração Contínua.

4.2 Situação anterior

O NSI não tinha um processo de Integração Contínua consistente. O processo era feito da seguinte maneira: O Buildbot, servidor de Integrção Contínua utilizado pelo NSI, não tinha o papel característico dos servidores de integração, cuja tarefa é fiscalizar o repositório à procura de mudanças. O Buildbot ficou responsável por executar o *script* de *build* em um determinado horário todos os dias, caracterizando o mecanismo de *build* programado. Da forma como era feita a integração no NSI, o modelo de integração utilizado era o Assíncrono.

O *script* de *build* criado utilizava o Buildout[1], sendo esta uma ferramenta *open source* de construção de software criada em Python[2], fornecendo suporte à criação de aplicações.

Quando chegava a hora do *build* ser executado, ele compilava todo o sistema e executava a suíte de testes. Caso algum teste falhasse, a equipe de desenvolvimento recebia um e-mail, sendo este o mecanismo de *feedback* escolhido, com a notificação da falha do *build*.

4.3 Problemas

Uma recomendação para ambientes que usam a Integração Contínua é sempre manter a base de dados estável. Por indisciplina da equipe esta recomendação não era atendida já que, toda vez que algum teste não passava, a equipe recebia a notificação pelo e-mail e mesmo assim o código do repositório ficava inconsistente por muito tempo. A medida correta a ser tomada era corrigir o erro o quanto antes, entretanto o que acontecia era a persistência do erro por dias, ou até mesmo, meses.

Ademais, como a execução do *build* acontecia somente à noite, caso o mesmo viesse a falhar, a tarefa de descobrir a localização do erro era mais difícil. Muitos *commits* eram realizados no período da tarde, todavia nenhum deles ativava o *script* de *build*. Dessa forma, quando o *build* quebrava, era mais difícil saber qual *commit* fez o *build* falhar, visto que foram realizados vários envios de código naquele determinado dia.

O *feedback* da integração não era instantâneo, já que o *build* era programado e só era executado em determinado horário do dia. Logo, para o desenvolvedor saber se a alteração que ele enviou para o repositório foi integrada corretamente, ele precisava esperar até o outro dia para saber o resultado da integração.

Além disso, a idéia do botão da integração não era completa, pois não se conseguia ter software funcional com apenas o clique de um botão. De acordo com o princípio do botão da integração, com apenas um comando deve ser gerado software funcional. Para se obter o

[1] http://www.buildout.org/
[2] http://www.python.org/

software final era necessário a realização de uma série de etapas manuais, o que não condiz com a idéia do botão da integração.

4.4 Proposta para implantação da Integração Contínua

Como a Biblioteca Digital é um projeto de grande porte que está em andamento, decidiu-se por utilizar o modelo de Integração Contínua Assíncrona por este ser de mais rápida implantação e adaptação, comparado com o modelo de integração Síncrono.

O processo começava com os desenvolvedores obtendo o código do repositório da RENAPI para realizar as alterações. Após modificar o código fonte, os desenvolvedores executavam a suíte de testes à procura de falhas. Obviamente, se alguma falha fosse encontrada, eles teriam que corrigí-la. Caso contrário, eles enviariam o código alterado para o repositório da RENAPI. Nesse estágio, havia uma máquina de integração que possuia um servidor de Integração Contínua, o Hudson. Ele tinha como responsabilidade fiscalizar o repositório da RENAPI constantemente à procura de qualquer modificação no código fonte do projeto. Caso o Hudson encontrasse alguma alteração no repositório, ele iria resgatar e executar o *script* de *build* na máquina de integração. Toda suíte de testes era executada e, se alguma falha acontecesse, o *build* falharia e a equipe de desenvolvimento receberia um e-mail avisando acerca da falha.

O ambiente de Integração Contínua realizado no projeto pode ser visto na Figura 4.1

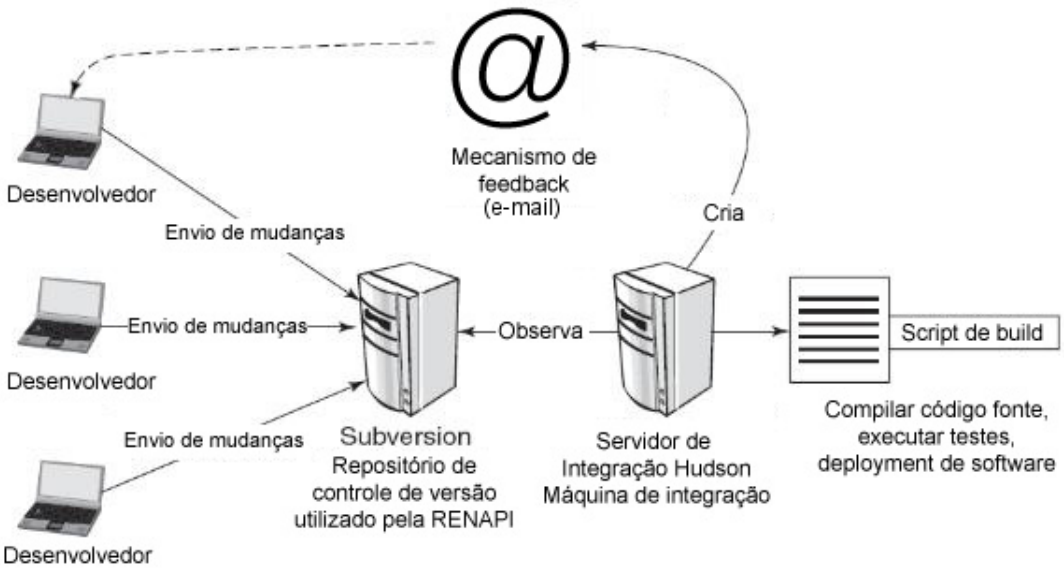


Figura 4.1: Componentes do ambiente de Integração Contínua do estudo de caso.

4.5 Como foi feita a implantação

Para a implantação foi alocada um máquina com processador Pentium 4, 2 GB de Memória RAM e 80 GB de HD. Nesta máquina foi instalado o sistema operacional Debian 5, o mesmo utilizado pelo sistema em produção.

A máquina de integração era totalmente virgem, ou seja, só foi instalado o Debian 5, o Hudson versão 1.357, bem como o Java que é dependência do servidor Hudson. Para instalar o Hudson foi necessário adicionar a chave do repositório Hudson ao sistema com os comandos apresentados na Figura 4.2.

```
wget -O /tmp/key http://hudson-ci.org/debian/hudson-ci.org.key
sudo apt-key add /tmp/key
```

Figura 4.2: Comandos para adicionar chave do repositório Hudson

Após isto, bastou somente instalar o Hudson através dos seguintes comandos, como mostra a Figura 4.3:

```
wget -O /tmp/hudson.deb http://hudson-ci.org/latest/debian/hudson.deb
sudo dpkg --install /tmp/hudson.deb
```

Figura 4.3: Comandos para instalar o Hudson

Instalando o Hudson dessa forma, ele pode ser atualizado pelo gerenciador de pacote do Debian, o apt-get, o que viabiliza o uso da última versão deste servidor de Integração Contínua.

Devidamente instalado, o Hudson foi acessado pela porta 8080, atráves do endereço *http://localhost:8080*. A tela principal do Hudson, pode ser vista na Figura 4.4

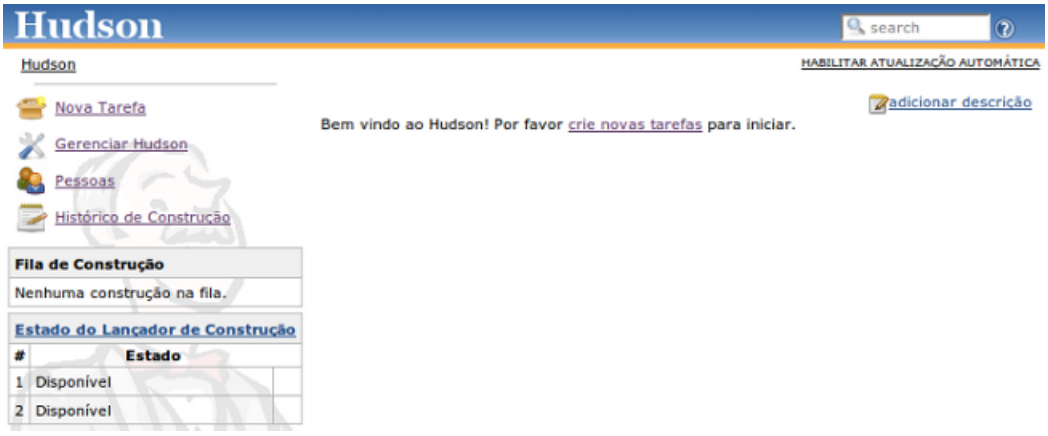


Figura 4.4: Tela principal do Hudson

O primeiro passo foi fazer a configuração da notificação de e-mail. Caso houvesse alguma falha no *build*, a equipe de desenvolvedores deveria receber um e-mail de notificação da quebra do *build*. Para configurar tal notificação acessou-se o *link* "Gerenciar Hudson" na tela principal do Hudson e logo após o ítem "Configurar Sistema".

Na seção "Notificação de E-mail" o campo "Servidor SMTP" foi preenchido com o respectivo servidor de SMTP e o campo "Endereço de E-mail do Administrador do Sistema" foi preenchido com o e-mail do responsável pela administração do Hudson. A seção "Notificação de E-mail" é demonstrada na Figura 4.5.

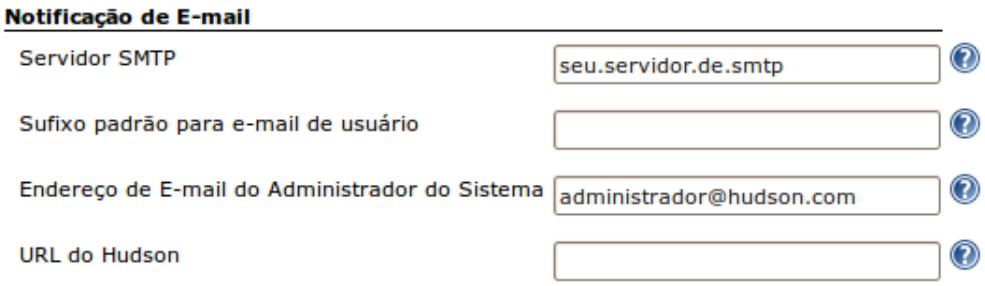


Figura 4.5: Tela de configuração da Notificação de E-mail no Hudson

Feita a configuração da notificação de e-mail, foi instalado o *plugin* do Selenium[3], que é um *framework* para automatização de testes para aplicações *Web*, sendo este usado nos testes de aceitação do projeto da Biblioteca Digital. Para instalar este *plugin* acessou-se o link "Gerenciar Hudson" na tela principal do Hudson e logo após o ítem "Gerenciar Plugins". Em seguida foi acessada a aba "Disponíveis". Nesta aba, aparecem todos os *plugins* que o Hudson oferece para instalação. Na seção "Cluster Manegement and Distributed Build (Gerenciamento de *Cluster* e *Build* Distribuído)" instalou-se o "Selenium Plugin", que é responsável por inicializar este *framework*. A instalação do "Selenium Plugin" pode ser vista na Figura 4.6

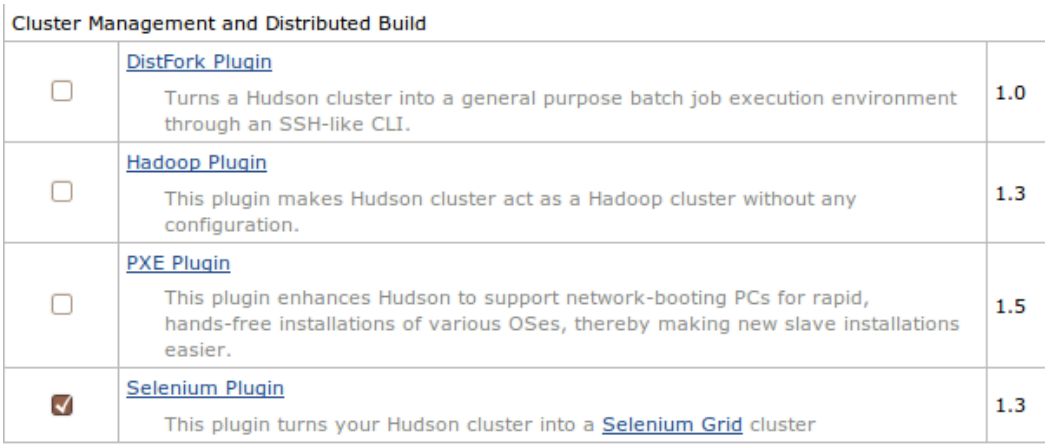


Figura 4.6: Tela de instalação do "Selenium Plugin" no Hudson

[3]http://seleniumhq.org/

Após a instalação desse *plugin*, acessou-se no menu lateral esquerdo da tela principal o ítem “Gerenciar Hudson”, sucedido do ítem “Manage Nodes (Gerenciamento de Nós)”, que apresenta uma tela de configuração e criação de novos nós. No menu lateral esquerdo, há o ítem “New Node (Novo Nó)” que representa a criação de um novo nó. Clicou-se nele e na tela posterior foi preenchido o campo “Node name (Nome do Nó)” com o conteúdo “Selenium Grid” e selecionado a opção “Dumb Slave”, como mostra a Figura 4.7.

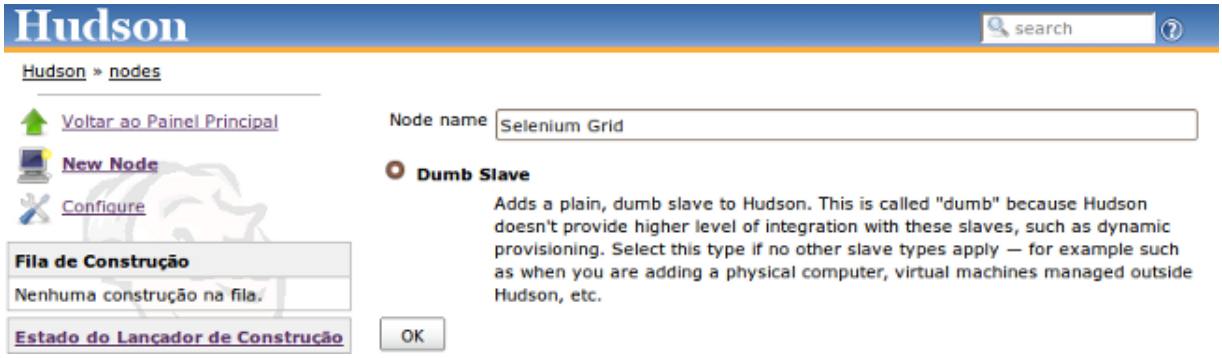


Figura 4.7: Tela de criação de nós no Hudson

Na tela seguinte foi nescessário configurar o números de processos concorrentes no processador, para isso foi preenchido o campo “Número de executores” com o valor “10”, ou seja, será permitido 10 processos concorrentes do Selenium no processador. Após isso, o campo “Diretório de trabalho do sistema remoto” foi preenchido com o caminho do diretório onde o Hudson deposita seus arquivos de configuração. Em “Uso” selecionou-se a opção “Leave this machine for tied jobs only (Deixar esta máquina somente para trabalhos amarrados)”, o que significa que o nó somente será utilizado quando este for configurado por uma determindada tarefa, permanecendo os demais campos com o valor padrão. Esta configuração pode ser observada na Figura 4.8.

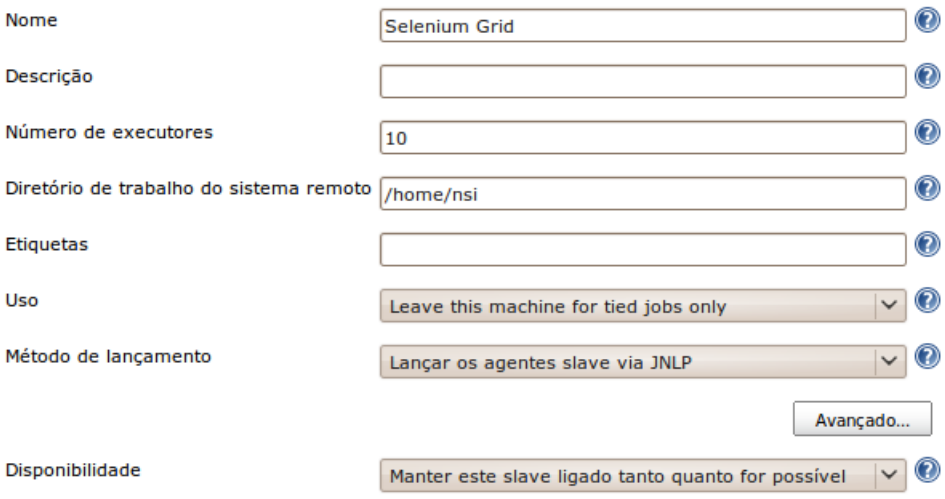


Figura 4.8: Tela de configuração de nós no Hudson

O próximo passo foi iniciar o nó recém criado, Selenium Grid. Para isso foi pressionado o botão “Launch (Iniciar)”, e salvo o arquivo “slave-agent.jnlp” no mesmo diretório configurado no campo “Diretório de trabalho do sistema remoto”. Após isso, foi colocado no arquivo /etc/rc.local a linha de comando “javaws /home/nsi/slave-agent.jnlp”, linha esta que faz com que o Selenium Grid seja inicializado automaticamente, evitando a inicialização manual toda vez que a máquina for ligada.

Depois da configuração do Selenium, acessou-se o link “Nova Tarefa” no menu lateral esquerdo da tela principal. Nesta tela, colocou-se o nome da tarefa como “Biblioteca Digital” e escolheu-se a opção “Construir um projeto de software free-style”, como mostra a Figura 4.9.

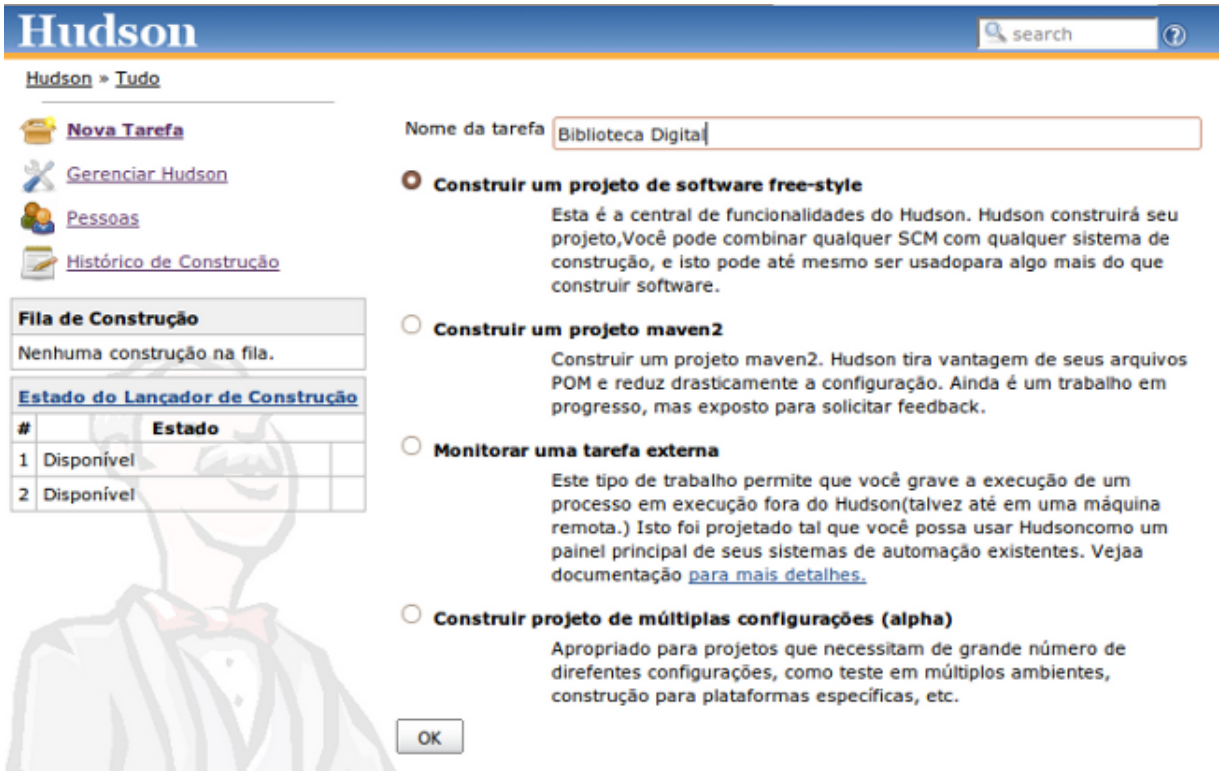


Figura 4.9: Tela de criação de tarefas no Hudson

Após preencher os dados requisitados, clicou-se em “OK” e uma nova tela foi aberta com dados para configuração da nova tarefa. Na opção “Tie this project to a node (Amarrar este projeto a um nó)” escolheu-se o nó criado anteriormente, no caso o Selenium Grid, como demonstra a Figura 4.10.

Figura 4.10: Tela de uma tarefa sendo ligada ao Selenium

O próximo passo foi configurar o Sistema de Controle de Versão utilizado pelo projeto

da Biblioteca Digital, o Subversion. Na seção “Gerenciamento de Código Fonte” foi selecionado a opção “Subversion”, preencheu-se o campo “URL do projeto” com o endereço onde se encontra o repositório da Biblioteca Digital. A opção “Usar atualização” foi marcada, para que a cada *build* os arquivos do repostório sejam atualizados automaticamente. A Figura 4.11 demostra este processo de configuração do Subversion.

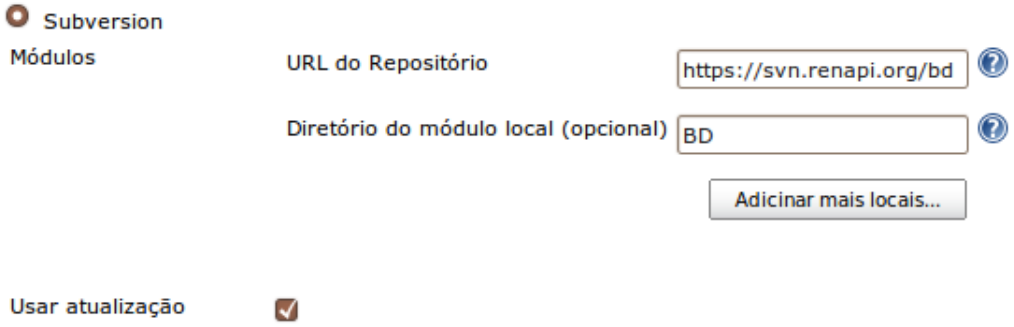


Figura 4.11: Tela de configuração do Subversion em uma tarefa

Na seção “Disparadores de Construção” foi marcada a opção “Consultar periodicamente o SCM”. O campo “Agenda” que apareceu, foi preenchido com “0,30 * * * *”, que segue o padrão Cron, o que significa que o Hudson fiscalizava o repositório no minuto “0” e “30” de cada hora, todos dias do mês, todos meses e todos dias da semana. Se alguma mudança acontecesse no repositório em alguma dessas fiscalizações, o *build* era disparado. Essa configuração é visualizada na Figura 4.12.

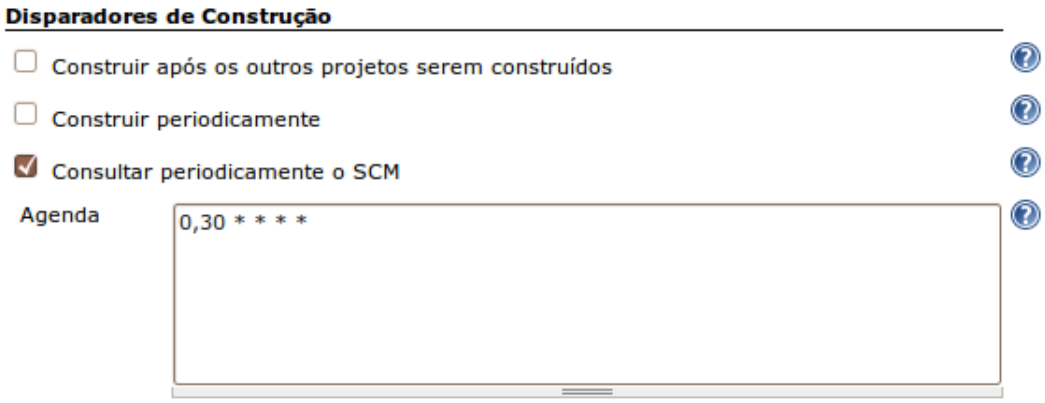


Figura 4.12: Tela de configuração de fiscalização de repositório em um tarefa

Na seção “Construção” foi escolhida a opção “Add Build Step (Adicionar etapa de *build*)” e depois a opção “Executar shell”. A respectiva caixa da opção “Executar shell” foi preenchida com os comandos referentes ao *build*. A Figura 4.13 exibe esta etapa.

Por último, na seção “Ações pós-contrução”, marcou-se a opção “Notificação de E-mail” e no campo “Destinatários” colocou-se o endereço de e-mail de todos os desenvolvedores da

equipe de desenvolvimento da Biblioteca Digital. A Figura 4.14 demonstra como fazer esta configuração.

Figura 4.13: Tela com comandos de execução de um *build* em uma tarefa

Portanto, esta tarefa foi configurada para obter os arquivos do repositório, executar o *script* de *build*, executar testes e enviar e-mail em caso de quebra do *build*.

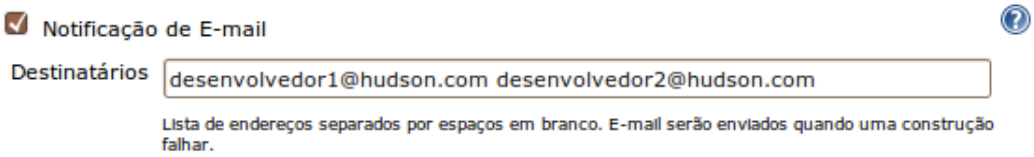


Figura 4.14: Tela com configuração de envio de e-mail em uma tarefa

4.6 Dificuldades para a implantação

Uma dificuldade encontrada foi com o *script* de *build*. Para configurar todo o sistema no *script*, foi necessário um esforço maior do que o esperado. O *script* apresentou problemas que exigiram bastante dos desenvolvedores, como por exemplo alguns trechos de código que estavam estáticos, assim dificultando a dinamização do *build*.

Além disso, como o NSI possui muitos desenvolvedores e pesquisadores, alocar a máquina de integração tornou-se uma tarefa difícil. Como a máquina de integração era um computador pessoal (*desktop*), ela ocupava o espaço que um desenvolvedor ocuparia. Aliado a isto, como a máquina deve ser usada exclusivamente para a integração do sistema, nenhuma pessoa deveria fazer uso da máquina.

Outra dificuldade que tomou bastante tempo foi o fato dos testes do projeto estarem falhando. Como o *script* de *build* executava todos os testes do sistema, foram descobertos alguns erros. Logo, como deve-se sempre manter o repositório consistente, alguns dias foram necessários para consertar os erros, inclusive para que o processo começasse de forma correta.

4.7 Resultados obtidos

No estudos de caso desse trabalho foi utilizado um Sistema de Controle de Versão chamado Subversion, que se encontra no endereço https://svn.renapi.org/bd.

O repositório de código era monitorado, e caso alguma mudança fosse detectada, o *script* de *build* era executado, realizando a instalação dos módulos, a execução dos testes, bem como o *deployment* do sistema.

O *build* é auto-testável, ou seja, fazia com que toda a integração do sistema fosse mais facilmente testada.

Uma máquina de integração foi alocada exclusivamente para o processo de integração. Ela não foi utilizada por nenhum desenvolvedor da equipe e continha somente os requisitos necessários para a integração do sistema.

4.8 Problemas remanescentes

A execução dos testes de cada módulo do sistema fez com que testes unitários e de aceitação fossem executados juntos. Com isso, inúmeras inicializações do Selenium eram feitas para execução dos testes de aceitação de cada módulo, acarretando em um tempo maior de *build*. Essas inicializações ocorriam pelo fato do processo do Selenium ser inicializado e finalizado para cada módulo, ou seja, um único processo não poderia ser aproveitado pelos demais módulos. Logo, esse transtorno do Selenium e a alta complexidade do projeto, fizeram com que a integração de apenas cinco módulos estivesse em torno de quarenta minutos.

O sistema inteiro ainda não estava sendo integrado completamente. Somente uma parte do projeto estava inserida no processo de Integração Contínua. Isso ocorria porque integrar todo o sistema iria aumentar exageradamente o tempo de execução do *build*.

O Hudson, que estava sendo executado na máquina de integração, não podia ser visto pelos desenvolvedores da equipe que estivessem fora do IFF. O ideal era que de qualquer lugar, qualquer pessoa pudesse ver como estava a situação do *build* naquele momento.

5 CONCLUSÕES

5.1 Objetivos alcançados

Com o trabalho realizado foi possível documentar a implantação da Integração Contínua no projeto da Biblioteca Digital da RENAPI.

Com o ambiente de Integração Contínua implantado, mesmo que incompleto, percebeu-se que os desenvolvedores ficam mais confiantes no projeto, visto que a base de código não fica inconsistente por muito tempo. Com toda suíte de testes sendo executada, a confiabilidade do projeto aumenta, bem como a qualidade do código desenvolvido.

Além disso, a documentação criada acerca da teoria da Integração Contínua e da descrição do estudo de caso, criam um bom material para conhecimento da prática e análise de um ambiente de integração em um projeto real.

5.2 Trabalhos futuros

Como o modelo Síncrono de integração é mais confiável e apresenta menos desvantagens do que o modelo Assíncrono, pretende-se utilizar o modelo de Integração Contínua Síncrono ao invés do modelo de Integração Contínua Assíncrono, bastando para isso, disciplinar a equipe para nunca deixar a base de código instável e obter um símbolo de integração para mostrar a equipe quem está integrando naquele exato momento.

Ademais, de acordo com o que foi mencionado na sessaõ 4.8 - Problemas remanescentes, almeja-se separar os testes unitários dos testes de aceitação para agilizar o processo de *build*. Para dinamizar a integração, os testes de aceitação podem ser executados somente uma vez no dia, enquanto os testes unitários devem ser executados a cada *commmit*. Com a redução do tempo do *build*, almeja-se também, fazer a integração de todos os módulos do sistema.

REFERÊNCIAS BIBLIOGRÁFICAS

ASTELS, D. *Test-Driven Development: A Practical Guide*. New Jersey: Prentice Hall, 2003.

BECK, K. *Programação extrema explicada: acolha as mudanças*. Porto Alegre: Bookman, 2004.

DUVALL, P. M. *How continuous integration improves software quality*. 2007. Disponível em http://searchsoftwarequality.techtarget.com/news/interview/0,289202,sid92_gci1261901,00.html, acesso em 03/03/2010.

DUVALL, P. M.; MATYAS, S.; GLOVER, A. *Continuous Integration: Improving Software Quality and Reducing Risk*. Boston: Addison-Wesley, 2007.

FOWLER, M. *Continuous Integration*. 2006. Disponível em http://martinfowler.com/articles/continuousIntegration.html, acesso em 03/03/2010.

IMPROVE IT. *Integração Contínua*. 2006a. Disponível em http://improveit.com.br/xp/praticas/integracao, acesso em 25 de fevereiro de 2010.

IMPROVE IT. *Manifesto Ágil*. 2006b. Disponível em http://improveit.com.br/xp/manifesto_agil, acesso em 03/03/2010.

IMPROVE IT. *Passos de bebê*. 2006c. Disponível em http://improveit.com.br/xp/principios/passos_bebe, acesso em 03/03/2010.

KOSKELA, L. *Test Driven: TDD and Acceptance TDD for Java Developers*. New Jesey: Manning Publications, 2007.

LACOMBE, C. *Anteprojeto de Carta de Preservação do Patrimônio Arquivístico Digital*. 2004. Disponível em http://libdigi.unicamp.br/document/?code=8385, acesso em 03/03/2010.

MALDONADO, J. C.; DELAMARO, M. E.; JINO, M. *Introdução ao Teste de Software*. São Paulo: Campus, 2007.

MYERS, J. G. *The Art of Software Testing*. New Jersey: John Wiley & Sons, Inc, 2004.

OLIVEIRA, G. H. R. *Análise de ferramentas de cobertura de testes baseadas em código*. Recife, 2004.

PRAGMATIC AUTOMATION. *Bubble, Bubble, Build's In Trouble*. 2007. Disponível em http://www.pragmaticautomation.com/cgi-bin/pragauto.cgi/Monitor/Devices/BubbleBubbleBuildsInTrouble.rdoc, acesso em 03/03/2010.

PRESSMAN, R. S. *Engenharia de Software*. São Paulo: MAKRON Books, 1995.

RIBEIRO, C. L. *A Relação Entre TDD e Qualidade de Software*. 2010. Disponível em http://www.infoq.com/br/articles/relacao-tdd-qualidade, acesso em 25 de fevereiro de 2010.

SHORE, J.; WARDEN, S. *A Arte do Desenvolvimento Ágil*. Rio de Janeiro: Alta Books, 2008.