LETÍCIA DOS SANTOS SILVA

LISTA DE VERIFICAÇÃO PARA A IMPLANTAÇÃO DA INTEGRAÇÃO CONTÍNUA

Monografia apresentada ao PECE - Programa de Educação Continuada em Engenharia da Escola Politécnica da Universidade de São Paulo como parte dos requisitos para conclusão do curso de MBA em Tecnologia de Software.

São Paulo

2016

LETÍCIA DOS SANTOS SILVA

LISTA DE VERIFICAÇÃO PARA A IMPLANTAÇÃO DA INTEGRAÇÃO CONTÍNUA

Monografia apresentada ao PECE - Programa de Educação Continuada em Engenharia da Escola Politécnica da Universidade de São Paulo como parte dos requisitos para a conclusão do curso de MBA em Tecnologia de Software.

Área de Concentração: Tecnologia de Software

Orientador: Prof. Dr. Paulo Sério Muniz Silva

São Paulo

2016

Catalogação-na-publicação

Silva, Leticia dos Santos
Lista de verificação para implantação da integração contínua / L. S. Silva -- São Paulo, 2016.
74 p.

Monografia (MBA em Especialização em Tecnologia de Software) - Escola Politécnica da Universidade de São Paulo. PECE - Programa de Educação Continuada em Engenharia.

1. Softwares (Controle) 2. Softwares (Desenvolvimento) 3. Softwares (Geração) 4. Integração de tecnologia I. Universidade de São Paulo. Escola Politécnica. PECE - Programa de Educação Continuada em Engenharia II. t.

AGRADECIMENTOS

Ao Prof. Dr. Paulo Sérgio Muniz Silva por toda a motivação, confiança e apoio durante a orientação deste trabalho. Aprendi muito com ele.

A minha família pelo amor, motivação e apoio incondicional durante todos esses anos.

A todos que contribuíram de alguma forma para a realização deste trabalho.

RESUMO

Nos processos típicos do desenvolvimento de software, o código é compilado, os dados são definidos e manipulados através de um banco de dados, os testes são realizados, o código é revisto e, finalmente, o software é implantado. Além disso, as equipes precisam se comunicar a respeito do estado do software. Com a integração contínua muitos desses processos são automatizados, mitigando os riscos relacionados ao desenvolvimento de software, como os decorrentes da descoberta tardia de defeitos e código de baixa qualidade. No entanto, não se pode obter sucesso na implantação da integração contínua em uma empresa de software se não houver uma base sólida de gerência de configuração. Neste sentido, este trabalho propõe duas listas de verificação para auxiliar as empresas de desenvolvimento de software na preparação de seu ambiente de produção para a implantação da integração contínua: uma para avaliar o estado atual da gerência de configuração e outra para avaliar as condições complementares para a implantação da integração contínua propriamente dita. Por fim, este trabalho apresenta um exemplo de aplicação das listas de verificação em uma empresa real que pretende implantar a integração contínua, para analisar alguns aspectos da viabilidade de utilização delas.

Palavras-chave: integração contínua, gerência de configuração de software, controle de versão.

ABSTRACT

On a typical process of software development, the code is compiled, the data is defined and manipulated through a database, the tests are performed, the code is reviewed and finally the software is deployed. Besides that, the teams need to communicate about the software status. Using continuous integration many of these processes are automated, mitigating many of the risks related to software development, such as those arising from the late discovery of defects and poor code quality. However, the success of continuous integration deployment can't be reached in a software company if there isn't a solid configuration management base. In this sense, this work proposes two checklists to help software development companies to prepare their production environment for the deployment of continuous integration: one to evaluate the current state of configuration management and the other to evaluate the additional conditions for the deployment of continuous integration itself. Finally, this work presents a case where the checklists are used in a real company that intends to deploy continuous integration, in order to analyze some aspects of the feasibility of their usage.

Keywords: continuous integration, software configuration management, version control.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

BPMN	Notação de Modelagem de Processo de Negócio *(Business Process Modeling Notation)*
IC	Integração Contínua
GCS	Gerência de Configuração de Software
SI	Segurança da Informação
TBD	Desenvolvimento Baseado em Tronco *(Trunk Based Development)*
TI	Tecnologia da Informação
UI	User Interface
UX	User Experience
XP	*Extreme Programming*

SUMÁRIO

1 INTRODUÇÃO

1.1 Motivações

Métodos de desenvolvimento tradicionais não ditam a frequência com que se deve integrar o código fonte de um projeto. Programadores podem trabalhar separadamente por horas, dias ou até semanas no mesmo código sem perceber os conflitos que estão gerando (BAKAL; ALTHOUSE; VERMA, 2012).

A integração do software não é um problema em um projeto com apenas uma pessoa e poucas dependências de sistemas externos, mas se o número de pessoas trabalhando ao mesmo tempo no projeto aumentar é necessário integrar para garantir que os componentes do software funcionem em conjunto. Esperar até o final do projeto para integrar pode levar a problemas de qualidade de software que são caros e podem gerar atrasos no projeto (DUVALL; MATYAS; GLOVER, 2007). Normalmente, a integração é uma atividade arriscada, pois podem acontecer diversas incompatibilidades entre a unidade e o todo (SAMPAIO, 2012). Os períodos de integração podem tomar um bom tempo e ninguém consegue prever quanto (HUMBLE; FARLEY, 2010) (FOWLER, 2006a).

A integração contínua originou-se como uma das práticas do processo de desenvolvimento *Extreme Programming* (XP) (HUMBLE; FARLEY, 2010) (FOWLER, 2006a). A ideia é que se integre regularmente o código, isso quer dizer, a cada envio de alguém no sistema de controle de versão. Com a integração contínua, é comprovado que o software está funcionando (considerando um conjunto abrangente de testes automatizados) a cada mudança e se sabe quando o software não está funcionando, devendo ser corrigido imediatamente. As equipes que usam integração contínua são capazes de entregar o software muito mais rápido e com menos erros, do que as que não usam (HUMBLE; FARLEY, 2010).

A integração contínua é mais que um processo, sendo apoiada por várias práticas e por alguns requisitos que devem ser cumpridos para o seu uso em um projeto (THOUGHTWORKS, 2015). Consequentemente, usar integração contínua é mais que instalar e configurar algumas ferramentas (DUVALL; MATYAS; GLOVER, 2007).

Em termos gerais, a integração contínua adiciona um importante valor à gerência de configuração de software (ou gerência de configuração), ao assegurar que cada mudança é prontamente integrada, fornecendo informações viáveis sobre a saúde do projeto (MOLITOR, 2012).

Algumas empresas não estão preparadas para implantar a integração contínua, uma vez que não possuem padrões essenciais da gerência de configuração. A gerência de configuração é uma parte muito importante e muitas vezes negligenciada na construção de sistemas de software (BERCZUK; APPLETON, 2002), embora sua estratégia determine como todas as mudanças que acontecem no seu projeto serão controladas. Apesar de os sistemas de controle de versão serem as ferramentas mais óbvias, a decisão de usar uma delas é apenas o primeiro passo na estratégia da gerência de configuração (HUMBLE; FARLEY, 2010). Em outras palavras, a gerência de configuração não se reduz somente à utilização de controle de versão.

O fato é que não é possível implantar integração contínua sem ter uma gerência de configuração eficaz. A gerência de configuração começa com o gerenciamento do código, esta é uma exigência para o sucesso. É necessário um sistema de controle de versão e saber como usá-lo de forma eficaz, isto inclui ser capaz de rotular as linhas de desenvolvimento, criar ramos e gerenciar as correções, independente da ferramenta. É indicado um treinamento para os membros da equipe do uso da ferramenta e dos procedimentos essenciais (AIELLO, 2015).

O *Build* deve ser totalmente automatizado e é também um pré-requisito absoluto para a IC. Ter um processo de *build* do código repetível permite que a equipe corrija as falhas, sem correr o risco de regredir o código devido à versão errada. Nessa etapa é necessário verificar se a linha de código base pode ser construída utilizando os procedimentos existentes. Muitas empresas não conseguem implantar um processo de *build* eficaz (AIELLO, 2015).

Os problemas mais comuns ao avaliar os padrões da gerência de configuração em uma empresa é que a maioria não pode provar que o código que está em produção é o correto e também não pode detectar alterações não autorizadas, o que gera um grande risco. Os procedimentos devem ser confiáveis e deve existir uma maneira de verificar que o código em produção seja o correto. Um procedimento automatizado deve ser implantando para detectar alterações não autorizadas, devido à intenção

maliciosa ou erro humano. Deve existir um procedimento para retornar o sistema a uma versão anterior, caso necessário (AIELLO, 2015).

As equipes que desejam implantar a integração contínua muitas vezes encontram dificuldades com os principais padrões da gerência de configuração que devem ser resolvidas previamente (AIELLO, 2015). Não se trata apenas de uma questão de escolher e implantar as ferramentas, embora isto seja importante, mas é fundamental aplicar os padrões essenciais da gerência de configuração antes de implantar a integração contínua (HUMBLE; FARLEY, 2010). Para introduzir os padrões da gerência de configuração é preciso fazer uma avaliação das práticas atuais e um plano para melhoria dos processos (AIELLO, 2015).

A maioria das empresas de desenvolvimento de software deveriam levar muito mais a sério a gerência de configuração, a sua incapacidade de fazê-lo resulta em muitos atrasos, perda de eficiência no desenvolvimento e aumento do custo de propriedade em uma base contínua (HUMBLE; FARLEY, 2010).

1.2 Objetivo

O objetivo deste trabalho é propor duas listas de verificação para auxiliar as empresas de desenvolvimento de software na preparação de seu ambiente de produção para a implantação da integração contínua (IC).

A lista de verificação para a implantação da gerência de configuração de software (GCS) avalia a estado atual da GCS. A lista de verificação para a implantação da IC avalia as condições complementares para que a implantação da IC possa ser bem sucedida.

Para exemplificar o uso das listas de verificação e realizar uma avaliação preliminar da viabilidade de utilização, elas foram aplicadas em uma empresa real de desenvolvimento de software que não possui integração contínua, mas pretende implantá-la.

1.3 Método de Trabalho

Para atingir o objetivo proposto, empregou-se um método baseado na pesquisa bibliográfica exploratória, na análise de soluções. Uma representação do método de pesquisa utilizado durante o presente trabalho é apresentada na Figura 1.1, utilizando a notação BPMN - *Business Process Modeling Notation* (Notação de Modelagem de Processo de Negócio) (OMG, 2013).

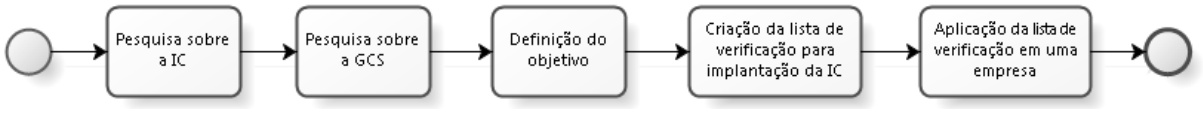


Figura 1.1: Método empregado para a realização do trabalho.

A Tabela 1.1 apresenta a descrição de cada uma das atividades representadas no método de pesquisa da Figura 1.1.

Tabela 1.1: Descrição das atividades do Método de Pesquisa.

Atividade	Descrição
Pesquisa sobre a IC	Através de uma pesquisa bibliográfica, buscou-se entender os fundamentos da IC, identificando as principais motivações, características, práticas e pré-requisitos para a implantação da IC.
Pesquisa sobre a GCS	Através de uma pesquisa bibliográfica, buscou-se entender as práticas e principais padrões da GCS, além das estratégias de controle de versão.
Definição do objetivo	As pesquisas sobre a IC e a GCS ajudaram na identificação dos problemas e levaram à definição do objetivo deste trabalho.
Criação das listas de verificação	Definiu-se duas lista de verificação para auxiliar as empresas na preparação do ambiente para a implantação da IC.
Aplicação das listas de verificação em uma empresa	Para exemplificar o uso das listas de verificação, as listas foram aplicadas em uma empresa de desenvolvimento de software que não possui a IC.

1.4 Estrutura do Trabalho

Este trabalho está organizado da seguinte forma:

- Capítulo 1. Introdução (este capítulo): apresenta as motivações, o objetivo, método de trabalho e a estrutura do trabalho.
- Capítulo 2. Gerência de Configuração de Software (GCS): apresenta a definição de GCS, os principais padrões da GCS e duas estratégias de controle de versão (ramo de funcionalidade e desenvolvimento baseado em tronco), destacando a diferença entre as estratégias.
- Capítulo 3. Integração Contínua (IC): apresenta o conceito de IC, os motivos para adotá-la, seus componentes, características, práticas e pré-requisitos para a implantação.
- Capítulo 4. Lista de Verificação para a Implantação da IC: apresenta a motivação para adotar as listas de verificação, as condições necessárias para implantar a GCS e a IC, e a proposta de duas listas de verificação para a implantação da GCS e preparação do ambiente para a implantação da IC.
- Capítulo 5. Exemplo de Aplicação das Listas de Verificação para a Implantação da IC: apresenta o processo de produção de *release* de uma empresa de desenvolvimento de software, o controle das versões do código, a aplicação das listas de verificação propostas e a análise de seus resultados.
- Capítulo 6. Conclusão: apresenta um breve resumo dos principais aspectos deste trabalho, as conclusões da análise dos resultados da aplicação das listas de verificação para a implantação da IC e os possíveis trabalhos futuros.

2 GERÊNCIA DE CONFIGURAÇÃO DE SOFTWARE

A integração contínua pode ser implantada de diversas maneiras. Neste trabalho considera-se que é de vital importância atender aos principais padrões da gerência de configuração antes de implantar a IC. Com isso, serão apresentados os conceitos, as práticas, os principais padrões da gerência de configuração e duas estratégias de controle de versão.

2.1 Visão Geral da Gerência de Configuração

A Gerência de Configuração de Software (GCS) é um processo de apoio no ciclo de vida do software que beneficia a gestão de projetos, desenvolvimento e as atividades de manutenção, garantia de qualidade, assim como os clientes e usuários do produto final (ABRAN; MOORE, 2004). Os objetivos da GCS são (IEEE, 2012):

- Identificar e documentar as características funcionais e físicas de qualquer produto, componente, resultado, ou serviço.
- Controlar as mudanças a estas características.
- Registrar e relatar cada mudança e seu estágio de implementação.
- Apoiar a auditoria dos produtos, resultados, serviços ou componentes para verificar a conformidade com os requisitos.

A GCS estabelece e protege a integridade de um produto ou componente do produto ao longo de seu ciclo de vida, a determinação das necessidades dos usuários e definição de requisitos do produto através dos processos de desenvolvimento, testes e entrega do produto, bem como durante a instalação, operação, manutenção e eventuais reparos (IEEE, 2012).

2.2 Principais Conceitos da GCS

Os principais conceitos da GCS são descritos a seguir:

- ***Build*** - uma versão operacional de um sistema ou componente que incorpora um conjunto de capacidades que o produto final irá fornecer (IEEE, 2012). Executa os passos requeridos para produzir a instancia de um produto (ISO, 2010).

- **Item de Configuração:** agregação de produtos de trabalho que é designado e tratado como uma única entidade no processo de gerência de configuração. Itens de configuração podem variar em complexidade, tamanho e tipo, que vão desde um sistema inteiro, incluindo todo o hardware, software e documentação, até um único módulo ou componente de hardware menor. (IEEE, 2012).

- **Linha de Base** *(Baseline)***:** especificação ou produto formalmente revisado e aceito, e que a partir disso serve como base para desenvolvimento posterior, podendo ser alterado através de procedimentos formais de controle de mudança (IEEE, 2012).

- **Linha de desenvolvimento** *(Codeline)***:** é um conjunto de arquivos e outros artefatos que compõem algum componente de software que se alteram ao longo do tempo. Toda vez que um arquivo ou artefato no sistema de controle de versão é alterado, uma revisão desse artefato é criada. A linha de desenvolvimento contém cada versão de cada artefato ao longo de um caminho evolutivo (BERCZUK; APPLETON, 2002).

- **Linha principal ou Tronco** *(Mainline ou Trunk):* a linha principal de desenvolvimento de software. O principal ponto de partida para a maioria dos ramos (ISO, 2010).

- **Mesclagem** *(Merge)***:** ato de combinar diferentes mudanças em um mesmo arquivo. Muitos sistemas seguem a estratégia otimista de combinar todas as linhas que não entram em conflito (ISO, 2010).

- **Submeter** *(Commit ou Check-in):* integrar mudanças feitas por um desenvolvedor no código fonte em um ramo acessível através do repositório de controle de versão (ISO, 2010).
- **Ramo** (*Branch)*: um novo segmento na linha principal de desenvolvimento (ISO, 2010).
- ***Release:*** coleção de itens de configurações novos ou alterados que são testados e introduzidos no ambiente em conjunto (IEEE, 2012).

2.3 Práticas da GCS

Há várias recomendações para a aplicação da GCS. Uma descrição das boas práticas da GCS que devem ser cumpridas pode ser explicada em (HASS, 2002):

- **Desenvolver uma estratégia de gerência de configuração:** incluindo as atividades da gerência de configuração e planejamento da realização destas atividades.
- **Estabelecer um sistema de gerência de configuração:** incluindo bibliotecas, padrões, procedimentos e ferramentas.
- **Identificar itens de configuração:** itens tais como software, sistemas, módulos, componentes e documentos relacionados ao identificar a documentação, e estabelecer a linha de base, versão de referência e outros detalhes relevantes.
- **Manter a configuração de descrição do item:** manter uma descrição atualizada de cada item de configuração.
- **Gerenciar de mudança:** registrar e relatar o estado dos itens de configuração e pedidos de modificação. Alterações a qualquer item de configuração devem ser revistos e autorizados.
- **Gerenciar releases de produtos:** os *releases* e as entregas de qualquer item de configuração devem ser revisados e autorizados.

- **Manter histórico do item de configuração:** manter detalhes suficientes para recuperar uma versão prévia de uma linha de base quando necessário.
- **Relatar o estado de configuração:** regularmente relatar o estado de cada item de configuração e sua relação na integração do sistema atual.
- **Gerenciar o *release* e a entrega dos itens de configuração:** o armazenamento, *release* e entrega dos itens de configuração devem ser controlados.

2.4 Principais Padrões da GCS

Uma definição de padrão (*pattern*) é uma "solução de um problema em um contexto" (BERCZUK; APPLETON, 2002). Segundo o padrão (*standard*) ISO/IEC/IEEE 24765 (ISO, 2010) que trata de sistemas e engenharia de software:

> "um padrão descreve um problema e a essência da sua solução para permitir que a solução seja reutilizada em diferentes cenários. Não se trata de uma especificação detalhada, mas de uma descrição a partir de uma experiência acumulada" (ISO, 2010, tradução da autora).

Esta seção apresenta os principais padrões da GCS definidos em Berczuk e Appleton (2002), o texto baseia-se longamente nesta referência, para fins descritivos. Os padrões não dependem de ferramentas, embora algumas ferramentas forneçam suporte aos padrões de maneira parcial ou total. Os padrões descritos são:

- Linha de Desenvolvimento Ativa
- Espaço de Trabalho Privado
- *Build* Privado de Sistema
- *Build* de Integração
- Teste de Fumaça
- Teste de Unidade

- Teste de Regressão

2.4.1 Linha de Desenvolvimento Ativa (*Active Development Line*)

A linha principal (*mainline*) consiste de uma linha de desenvolvimento central e atua como base para os ramos (*branches*). Todo desenvolvimento ocorre diretamente sobre a linha principal ou indiretamente através das mesclagens (*merges*). Este padrão ajuda a equilibrar a estabilidade e progresso em um esforço de desenvolvimento ativo.

Quando a equipe desenvolve software de forma concorrente, quanto mais pessoas estão trabalhando em um código aumenta a necessidade de comunicação entre os membros e existe maior possibilidade de conflito nas mudanças.

Os trabalhos dos membros da equipe são integrados na linha principal através dos pontos de sincronização. É necessário um gerenciamento dos pontos de sincronização para que não ocorra *deadlock* ou bloqueio (*blocking*). O *deadlock* pode acontecer quando duas pessoas têm dependência mutua e uma pessoa faz o *check-in* antes da finalização dos testes, sendo provável que os testes falhem na próxima execução. O bloqueio pode acontecer quando o processo de pré *check-in* demora e alguém precisa das mudanças para prosseguir.

Quando alguém faz o *check-in* de uma mudança na linha de desenvolvimento, isto pode causar atrasos para a equipe se a mudança provocar uma falha no trabalho de alguém. Para verificar se existe alguma incompatibilidade é necessário testar antes de fazer o *check-in* das mudanças com a última versão do código fonte. Mas se os testes consumirem muito esforço, isto pode causar atrasos. A demora entre os *check-ins* pode ir contra alguns dos maiores objetivos do projeto.

As pessoas podem executar procedimentos simples antes de submeter o código à linha de desenvolvimento, como um *build* preliminar e algum nível de teste.

Mesmo se o código for testado antes de fazer o *check-in*, problemas de concorrência podem acontecer, por exemplo, duas mudanças testadas individualmente, quando integradas podem quebrar o sistema.

Por mais exaustivo que sejam os testes em uma mudança, existe a probabilidade de uma segunda mudança não compatível ser submetida e os testes da primeira falharem, como ilustra a Figura 2.1.

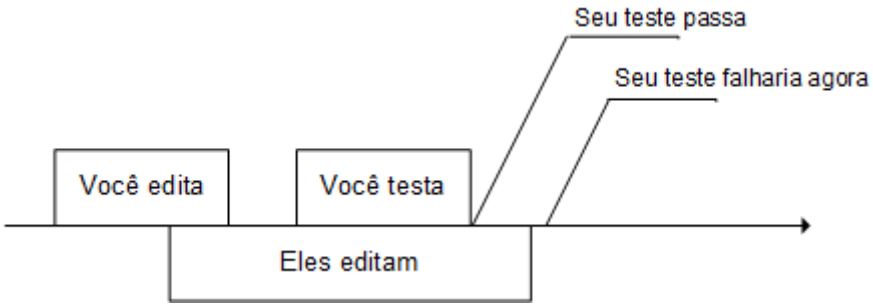


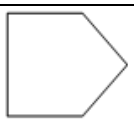
Figura 2.1: Falhas em teste de longa duração. Adaptado de Berczuk e Appleton (2002).

É possível fazer mudanças na estrutura da linha de desenvolvimento para manter parte do código estável, criando ramos em vários pontos, mas isto adiciona complexidade e requer uma mesclagem.

Para evitar fazer o *check-in* de mudanças na linha de desenvolvimento enquanto estiver testando use semáforos, mas apenas uma pessoa pode testar e fazer o *check-in* das mudanças por vez, o que pode gerar um processo lento.

Para representar o estado da linha de desenvolvimento foram consideradas as convenções apresentadas na Tabela 2.1:

Tabela 2.1: Convenções para o estado da linha de desenvolvimento.

Descrição	Representação
Uma versão da linha de código ou uma revisão de um arquivo.	
Mudança de tarefa, que é indicada pela descrição dentro da caixa.	
Rótulo ou revisão identificada.	

A Figura 2.2 apresenta uma linha de desenvolvimento estável, mas lenta.

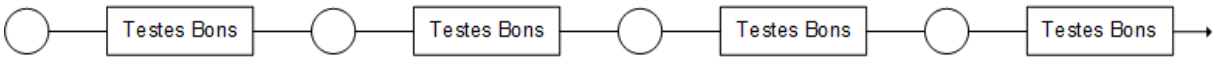


Figura 2.2: Linha de desenvolvimento pouco ativa e estável. Adaptado de Berczuk e Appleton (2002).

A Figura 2.3 apresenta uma linha de desenvolvimento com rápida evolução, mas não utilizável.

Figura 2.3: Linha de desenvolvimento muito ativa e pouco estável. Adaptado de Berczuk e Appleton (2002).

Não se deve adotar uma linha de desenvolvimento ativa perfeita, mas uma linha principal que seja utilizável, estável e ativa o suficiente para as necessidades do projeto.

Uma linha de desenvolvimento ativa terá mudanças frequentes, algumas bem testadas e outras fracamente. Os testes devem ser bons o suficiente para alguém desenvolver nessa linha, como ilustra a Figura 2.4.

Figura 2.4: Linha de desenvolvimento ativa. Adaptado de Berczuk e Appleton (2002).

A Figura 2.5 apresenta como as linhas de base são identificadas e rotuladas quando passam nos testes.

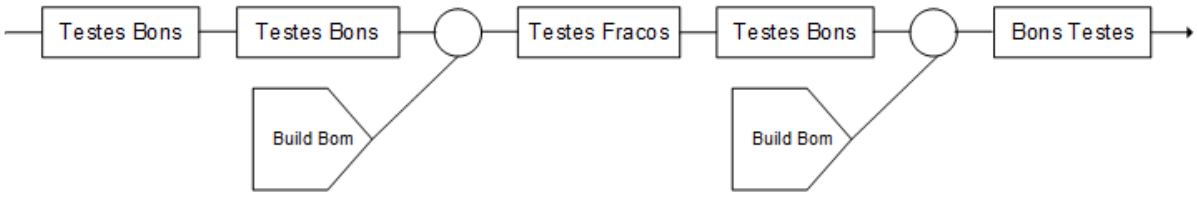


Figura 2.5: Identificação para bases estáveis. Adaptado de Berczuk e Appleton (2002).

O processo de *check-in* deve ser facilmente executado. Se o processo de pré *check-in* leva muito tempo, é provável que os desenvolvedores façam o *check-in* com

menos frequência, o que pode gerar atrasos e aumentar a possibilidade de conflitos durante os testes.

Para evitar problemas na linha principal é recomendado configurar um espaço de trabalho privado para o desenvolvedor, no qual ele pode fazer um *build* privado de sistema, teste de unidade e teste de fumaça.

2.4.2 Espaço de Trabalho Privado *(Private Workspace)*

O espaço de trabalho (*workspace*) é um lugar onde o desenvolvedor mantém todos os artefatos que ele precisa para realizar uma tarefa. Um espaço de trabalho é normalmente associado com a versão especifica desses artefatos.

Em uma linha de desenvolvimento ativa os desenvolvedores fazem mudanças frequentes no código. Para isso, devem trabalhar na versão mais recente do código e de maneira controlada. Este padrão descreve como conciliar a tensão entre desenvolver com a base atual do código e a realidade em que as pessoas não podem trabalhar de maneira eficaz quando o ambiente está em constante mudança.

Os desenvolvedores precisam trabalhar em um código isolado das mudanças externas. O desenvolvimento do código em equipe envolve normalmente as seguintes etapas:

- Escrever e testar as mudanças de código.
- Integrar o código com o trabalho dos outros.

A integração frequente no espaço de trabalho evita o trabalho com artefatos desatualizados e ajuda a isolá-los quando uma falha aparece. A Figura 2.6 apresenta a integração frequente de mudanças isoladas realizadas pelos membros da equipe no espaço de trabalho de um desenvolvedor.

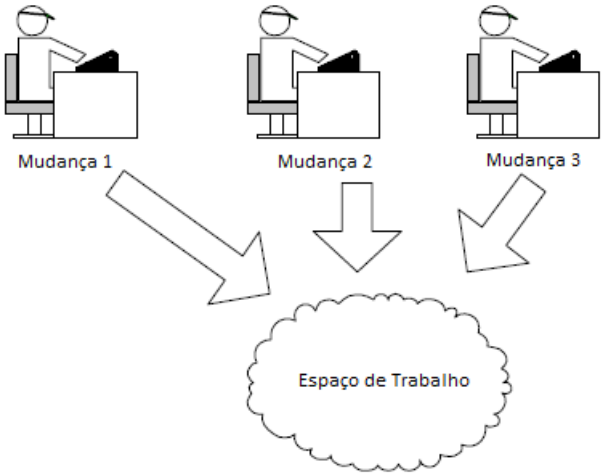


Figura 2.6: Integrando cada mudança que acontece (BERCZUK; APPLETON, 2002).

Integrar muitas mudanças de uma só vez pode tornar mais difícil isolar onde está à falha. A Figura 2.7 apresenta a integração de muitas mudanças realizadas pelos membros da equipe no espaço de trabalho de um desenvolvedor. Se o desenvolvedor não integrar com frequência, a integração poderá demorar bastante, uma vez que terá que resolver os conflitos que surgirem.

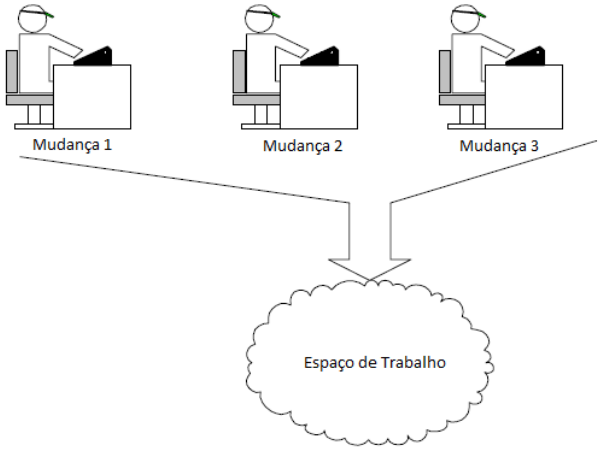


Figura 2.7: Integrando mudanças de uma só vez (BERCZUK; APPLETON, 2002).

Cada membro da equipe pode criar em um ambiente local um espaço de trabalho privado que deve conter tudo que precisa para executar suas tarefas em uma linha de desenvolvimento. Com um espaço de trabalho privado é possível controlar a versão do código e componentes dos quais se está trabalhando.

Um espaço de trabalho privado compreende:

- Código fonte.

- Componentes de *build*.
- Componentes de terceiros.
- Configurações e dados necessários para executar e testar o sistema.
- Scripts de *build*.
- Informações identificando a versão de todos os componentes do sistema.
- Ferramentas de desenvolvimento.

Para desenvolver na linha principal, é recomendável seguir um procedimento que deve:

- Criar um espaço de trabalho privado local com a versão mais recente do sistema.
- Realizar as alterações localmente. Editar os componentes necessários.
- Fazer um *build* privado de sistema para atualizar os objetos derivados.
- Testar suas mudanças com um teste de unidade.
- Atualizar o espaço de trabalho para a última versão de todos os componentes que não tiveram modificação.
- Fazer o *build* novamente e executar o teste de fumaça.

Para trabalhar em múltiplas tarefas é possível ter vários espaços de trabalho, cada um com sua própria configuração.

Um risco com o uso do espaço de trabalho privado é que os desenvolvedores trabalharão com código desatualizado. Uma maneira de evitar isto é fazer o *check-in* a cada tarefa e atualizar a espaço de trabalho antes de iniciar uma nova tarefa.

2.4.3 Build Privado de Sistema *(Private System Build)*

Um espaço de trabalho privado permite ao desenvolvedor isolar-se de mudanças externas ao seu ambiente. Este padrão explica como é possível verificar se o seu

código está consistente com o último código publicado quando submeter as suas alterações.

O único teste para saber se as mudanças são compatíveis é o *build* de integração centralizado. Mas eventualmente o *check-in* pode quebrar o *build*. A Figura 2.8 apresenta o *build* privado no espaço de trabalho do desenvolvedor. O *build* é executado antes do envio das mudanças para o repositório de controle de versão. São realizados testes para certificar que as mudanças de código integram com a versão atual.

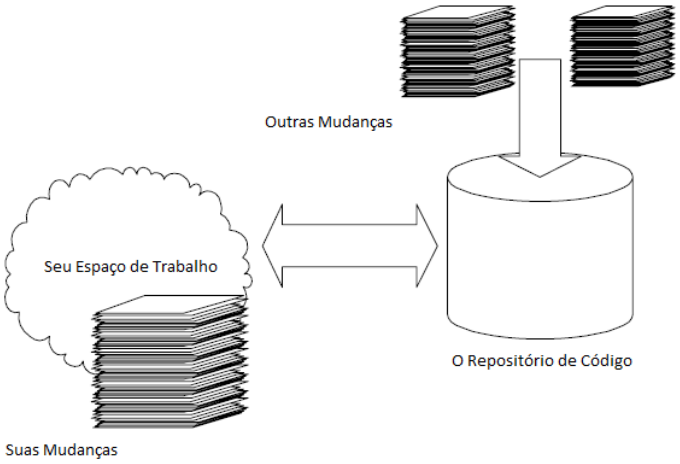


Figura 2.8: O *Build* integra as mudanças de todos (BERCZUK; APPLETON, 2002).

Antes de submeter o código deve ser realizado o *build* no sistema usando um *build* privado de sistema que é similar ao *build* noturno (*nightly build*). O *build* privado de sistema realiza um *build* local no espaço de trabalho do desenvolvedor, para certificar que as mudanças não quebram o *build*.

O *build* privado de sistema deve ter os seguintes atributos:

- Ser parecido com o *build* de integração *(integration build)*, usando pelo menos o mesmo compilador, versão dos componentes externos e estrutura de diretórios.
- Incluir todas as dependências.
- Incluir todos os componentes que são dependentes da mudança.

A arquitetura de software irá ajudar a determinar os componentes necessários para o *build*. A Figura 2.9 apresenta os componentes de um *build* privado de sistema.

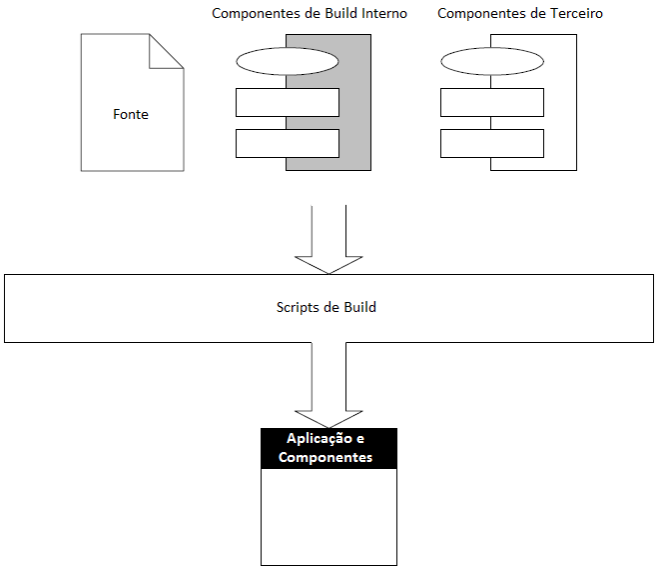


Figura 2.9: Componentes do *build* privado de sistema (BERCZUK; APPLETON, 2002).

O *build* privado de sistema pode demorar, mas este tempo é gasto por apenas uma pessoa e pode evitar problemas para toda a equipe.

Para certificar que não há falhas grosseiras nas funcionalidades, é recomendado fazer o teste de fumaça. Se o sistema é muito grande, o *build* de todas as dependências dos componentes modificados pode não ser eficiente, uma vez que as dependências serão validadas no *build* de integração.

2.4.4 Build de Integração *(Integration Build)*

Os desenvolvedores realizam mudanças independentes nos espaços de trabalho que devem ser integradas e todo o sistema deve ser construído consistentemente. Este padrão fornece mecanismos que ajudam a garantir que o código do sistema sempre funcione. Uma vez que muitas pessoas estão fazendo mudanças no código, um desenvolvedor não consegue garantir que todo o sistema funciona depois que as mudanças são integradas na linha principal. Alguém pode fazer mudanças em

paralelo, incompatíveis com as de outros desenvolvedores. A comunicação não é suficiente para evitar essas situações.

Quando se faz o *check-in* das mudanças de código é possível introduzir erros de *build*. O melhor a fazer é tentar realizar o *build* de cada parte do sistema, embora o *build* completo possa levar muito tempo. Por outro lado, o tempo gasto por uma pessoa para fazer o *build* completo pode ser pouco se comparado ao tempo que a equipe gasta para resolver os problemas se o *build* quebrar, além de atrasar os outros membros da equipe.

O processo de *build* deve ser:

- Reprodutível.
- O mais próximo possível de um *build* de produto final.
- Automatizado.
- Ter um mecanismo de notificação para identificar os erros e inconsistências. Quanto mais rápido os erros de *build* são identificados, mais rápido são corrigidos.

O processo de *build* de integração pode ser comparado a um quebra-cabeça, pois é necessário verificar se as peças são compatíveis, como ilustra a Figura 2.10.

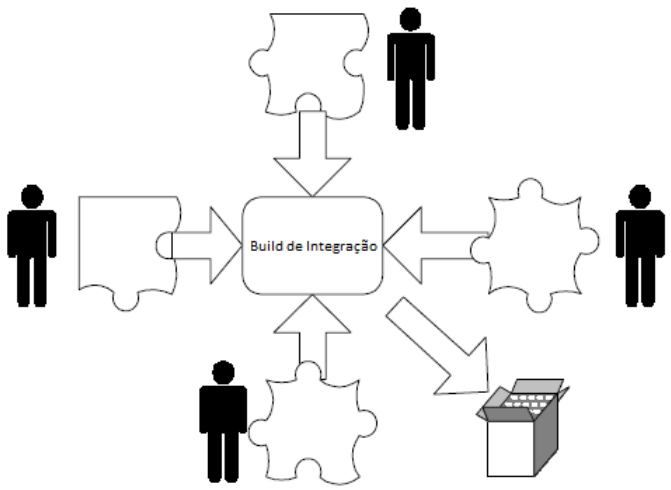


Figura 2.10: O processo de *build* de integração (BERCZUK; APPLETON, 2002).

O *build* deve ser realizado em um espaço de trabalho que contenha os componentes que estão sendo integrados. A determinação da frequência de execução do *build* de integração baseia-se nos seguintes fatores:

- A duração do *build*;
- A frequência com que acontecem as mudanças.

Se o *build* do sistema leva muito tempo, ou se a frequência de mudanças é baixa, devem ser realizados *builds* diários. Se o *build* é rápido, deve ser considerada a execução do *build* a cada *check-in*. Isto pode parecer intenso, mas ajuda a determinar as mudanças que quebraram o *build*.

Mesmo que o sistema funcione podem existir falhas. Após o *build* de integração, deve ser realizado um teste de fumaça ou um teste de regressão para garantir que o *build* de integração é utilizável.

2.4.5 Teste de Fumaça *(Smoke Test)*

Um *build* de integração ou um *build* privado de sistema são usados para verificar problemas de integração do *build*. Mas mesmo se o código funcionar é necessário verificar se as mudanças não causaram erros em tempo de execução. Essa verificação é essencial para manter a linha de desenvolvimento ativa. Este padrão aborda as decisões necessárias para validar um *build*.

É importante decidir quais testes executar depois do *build* e antes do *check-in*. Pode-se escrever testes para as partes mais críticas ou propensas a falhas de código, mas é difícil desenvolver testes completos. Podemos escrever testes para todas as partes do sistema, mas pode-se consumir muito tempo e atrasar o progresso do projeto. A execução de testes detalhados é demorada, mas se o *check-in* da mudança quebra o sistema, o tempo de todos é desperdiçado.

O escopo do teste de fumaça não precisa ser exaustivo. Ele deve testar as funcionalidades básicas e simples problemas de integração. Idealmente deve ser automatizado. Uma suíte de testes de unidade pode ser a base para os testes de fumaça, que devem ter enfoque de caixa-preta mais do que de caixa-branca. Esses

testes devem retornar o estado (se o teste passou ou não) e não requerer intervenção manual.

Executar o teste de fumaça a cada *build* não remove a responsabilidade do desenvolvedor de testar suas mudanças antes de submetê-las para o repositório. O teste de fumaça é usado para correção de erros e para verificar as interações entre as funcionalidades existentes e as novas. Quando se adiciona uma nova funcionalidade no sistema, deve-se estender o teste de fumaça para testar também esta funcionalidade. Mas não se deve utilizar testes exaustivos que se enquadram melhor em testes de unidade ou de regressão.

Um teste de fumaça deve:

- Ser rápido de executar, onde “rápido” depende de cada situação.
- Fornecer uma cobertura ampla em todo o sistema.
- Ser executado pelos desenvolvedores, como parte da garantia da qualidade.

Um teste de fumaça deixa lacunas que devem ser preenchidas por um processo de controle de qualidade e testes de regressão. Os desenvolvedores devem desenvolver testes de unidade para cada módulo, de modo a verificar se o módulo ainda está funcionando de forma adequada antes do *check-in* da mudança.

2.4.6 Teste de Unidade *(Unit Test)*

O teste de fumaça pode não ser suficiente para testar as mudanças em detalhes quando se trabalha com módulos, especialmente em um código novo. Este padrão mostra como testar em detalhes as mudanças para garantir a qualidade das linhas de desenvolvimento.

Os testes de unidade são escritos para verificar se uma classe, método ou função funciona após fazer uma mudança. É o procedimento básico que ajuda a manter a estabilidade do desenvolvimento de software (DUVALL; MATYAS; GLOVER, 2007). É mais fácil entender o que está errado no nível unitário do que no nível de sistema. Por outro lado, testar em pequena escala pode ser tedioso.

A integração é o momento em que a maioria dos problemas aparecem, mas quando se tem o resultado de uma falha do teste de integração, é necessário saber o que quebrou. Para isso, deve-se isolar os problemas de integração das mudanças locais e testar o contrato que cada elemento fornece localmente.

Um teste de unidade é um teste que verifica se os elementos dos componentes obedecem a seu contrato. Um bom teste de unidade tem as seguintes propriedades:

- Automatizado.
- Simples de executar.
- Granulação fina. Cada método significativo da interface de uma classe deve ser testado. O ideal é testar elementos que podem gerar erros.
- Isolado. Um teste de unidade não deve interagir com outros testes.
- Deve testar o contrato. Se uma interface muda, os testes para refletir isto devem ser atualizados.

Testes de unidade devem ser executados:

- Enquanto o software é codificado.
- Antes de fazer o *check-in* das mudanças e depois de atualizar seu código para a versão atual.

O teste de unidade pode ser executado para encontrar problemas com o teste de fumaça, teste de regressão ou em resposta a um problema reportado pelo usuário.

Para automatização desse teste deve-se usar um *framework* de teste como *JUnit, NUnit, PyUnit* ou outro similar.

2.4.7 Teste de Regressão *(Regression Test)*

O teste de fumaça é rápido, mas não exaustivo. Para estabelecer um candidato a *release*, é preciso garantir que o código seja robusto. Este padrão explica como gerar *builds* que não sejam piores que o ultimo *build*.

Sistemas de Software são complexos e a cada mudança é possível quebrar algo aparentemente não relacionado com a mudança. Ao corrigir um defeito é possível introduzir outro. Sem mudança, não se faz progresso, mas o impacto de mudar é difícil de mensurar, especialmente em termos de como a unidade do código interage com o resto do sistema.

Se um problema acontecer novamente, ele deve ser identificado. Não se deve perder tempo com problemas conhecidos. Um problema que acontece uma vez pode acontecer novamente (regressão). Deve-se ter meios de verificar essas falhas recorrentes.

O teste de regressão do sistema deve ser executado para garantir a estabilidade na linha de desenvolvimento, antes da liberação de um *build* ou da realização de uma mudança de alto risco.

Testes de regressão são testes caixa preta que cobrem o passado ou antecipam falhas. Uma boa estratégia é criar os casos de testes à medida que os problemas são encontrados. Com o passar do tempo haverá uma suíte de testes que cobrirá a maioria dos problemas.

O teste de regressão é projetado para certificar que o software não deu um passo para trás (regrediu). Sempre devem ser executados os mesmos testes para cada ciclo de regressão.

Os testes de regressão podem consumir bastante tempo, talvez não seja viável executá-los antes de cada *check-in* ou depois de cada *build*. Existem vantagens quando se o usa um procedimento automatizado para executar os testes de regressão depois de cada mudança, o que possibilita identificar o ponto em que o sistema regrediu. Esses testes devem fazer parte do *build* noturno.

2.5 Estratégias de Controle de Versão

2.5.1 Convenções Adotadas

Para representar as estratégias de controle de versão foram consideradas as convenções apresentadas na Tabela 2.2:

Tabela 2.2: Convenções para as estratégias de controle de versão.

Sigla	Conceito	Representação
B	Ramo	
C	*Commit*	
M	Mesclagem	
R	*Release*	
-	Enviar / Receber revisão	
-	Linha de desenvolvimento	

2.5.2 Ramo de Funcionalidade *(Feature Branch)*

Segundo Fowler (2009), “a ideia básica do ramo de funcionalidade é que quando se começa a trabalhar uma funcionalidade (ou história de usuário) cria-se um ramo no repositório para trabalhar a funcionalidade” (FOWLER, 2009, tradução da autora).

A Figura 2.11 mostra o uso de ramos de funcionalidades. Na linha principal existe um projeto compartilhado e dois desenvolvedores na primeira (João) e última (Charles) linha.

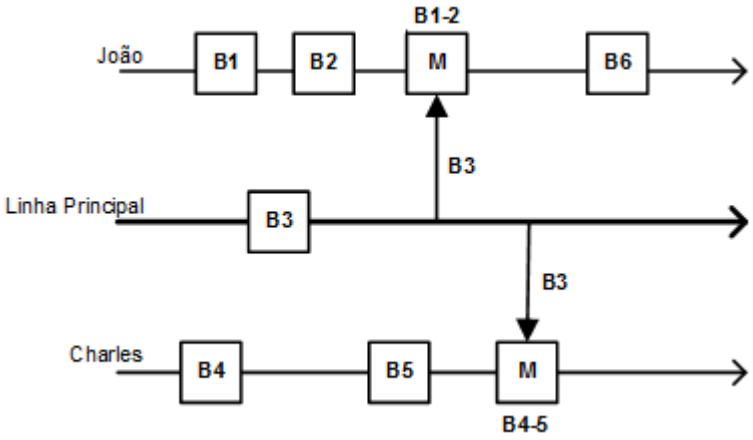


Figura 2.11: Uso de ramos de funcionalidades. Adaptado de Fowler (2009).

As caixas B (exemplo: B1 e B2) representam *commits* locais no ramo. Setas entre ramos representam mesclagens entre ramos realizadas nas caixas M.

Para garantir que tudo gera corretamente, os desenvolvedores podem executar *builds* e testes nos seus ramos.

Alguns pré-requisitos para este padrão funcionar bem (HUMBLE; FARLEY, 2010):

- Qualquer alteração na linha principal deve ser mesclada no ramo em uma base diária.
- O ramo deve ter uma vida curta, idealmente poucos dias, nunca mais que uma iteração.
- Ninguém deve criar um novo ramo a menos que tenha concluído o ramo anterior.
- Permitir que os desenvolvedores mesclem na linha principal somente se a funcionalidade for aceita.
- Refatorações devem ser mescladas imediatamente para minimizar conflitos.
- Parte das responsabilidades do líder técnico é manter a linha principal estável. O líder pode rejeitar algo que possivelmente quebre a linha principal.

A vantagem do ramo de funcionalidade é que cada desenvolvedor pode trabalhar sua funcionalidade e isolado das mudanças dos outros. O uso de ramo de

funcionalidade é motivado pelo desejo de manter a linha principal sempre estável. A equipe pode escolher as funcionalidades para o *release* (HUMBLE; FARLEY, 2010) (FOWLER, 2009).

Por outro lado, o uso de ramo de funcionalidade apresenta alguns problemas ou desvantagens (HUMBLE; FARLEY, 2010) (FOWLER, 2009) (HAMMANT, 2015):

- Ter muitos ramos com vida longa é ruim por causa de problemas na mesclagem. Se existem quatro ramos divergentes, cada um deles será mesclado somente na linha principal. Os testes são usados para descobrir conflitos, mas quanto mais código existir na mesclagem, surgirão mais conflitos e mais difícil será para corrigir.
- Conflitos semânticos são preocupantes. A Figura 2.12 permite representar as desvantagens.

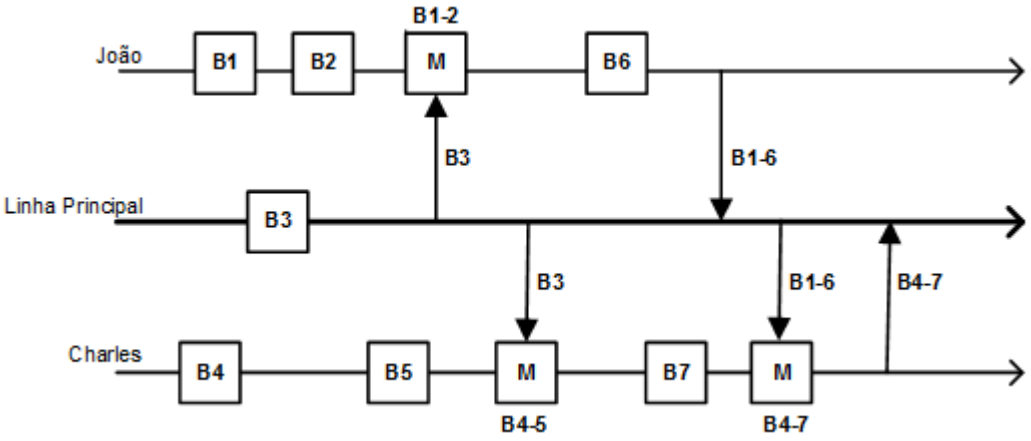


Figura 2.12: Conflitos de mesclagem. Adaptado de Fowler (2009).

Os desenvolvedores podem desenvolver as funcionalidades isoladamente, mas em algum momento o trabalho tem que ser integrado. Nesse caso, o João facilmente atualiza a linha principal com suas mudanças (B1-6). Não existe mesclagem, por que ele incorporou as mudanças da linha principal no seu próprio ramo. As coisas não são tão simples para o Charles, pois ele precisa mesclar todas as suas mudanças (B4-7) com as do João (B1-6). A mesclagem poderia ser simples se os desenvolvedores tivessem trabalhado em partes completamente separadas do código sem nenhuma interação. Mas tratando-se de uma mesclagem complexa, os desenvolvedores editam o

mesmo arquivo, por exemplo, o João alterou o nome de um método que o código do Charles invoca.

- A refatoração é um esforço para manter o código mais limpo. Mas a refatoração em ramo de funcionalidade irá gerar uma mesclagem grande e insegurança na equipe.
- O uso de ramos é incompatível com a IC. Este ponto será tratado na seção 4.6. Se as tarefas forem rápidas, não é necessário criar ramos e usar a estratégia de ramo de funcionalidade.

2.5.3 Desenvolvimento Baseado em Tronco *(Trunk Based Development)*

No desenvolvimento baseado em tronco (TBD) os desenvolvedores realizam o *commit* em um ramo compartilhado no repositório de controle de versão. Esse ramo, como foi visto anteriormente é conhecido como tronco ou linha principal (*trunk*) (HAMMANT, 2013b).

Desenvolvedores podem fazer múltiplos ramos na sua própria estação de trabalho, mas quando a alteração ou correção estiver pronta, devem enviá-los para a linha principal compartilhada. Ramos são criados para *releases*. Desenvolvedores não podem criar ramos no local compartilhado (HAMMANT, 2013b).

Os ramos antigos podem ser excluídos, não podendo receber nada da linha principal. Há suporte para mesclagens somente da linha principal para o ramo de versão. Normalmente, os erros devem ser corrigidos na linha principal e mesclados no ramo de versão. Poucas pessoas podem fazer o *commit* nos ramos e criar ramos de versão *(release branches)*. Desenvolvedores não podem quebrar o *build* ao realizar o *commit*. Isto requer disciplina. Muitas empresas usam verificações de pré *commit*. Embora o TBD possa ser usado sem a IC, em empresas com dezenas de desenvolvedores é recomendada a IC ligada à linha principal (HAMMANT, 2013b).

A Figura 2.13 apresenta o repositório compartilhado, onde os desenvolvedores integram / mesclam diariamente o seu trabalho.

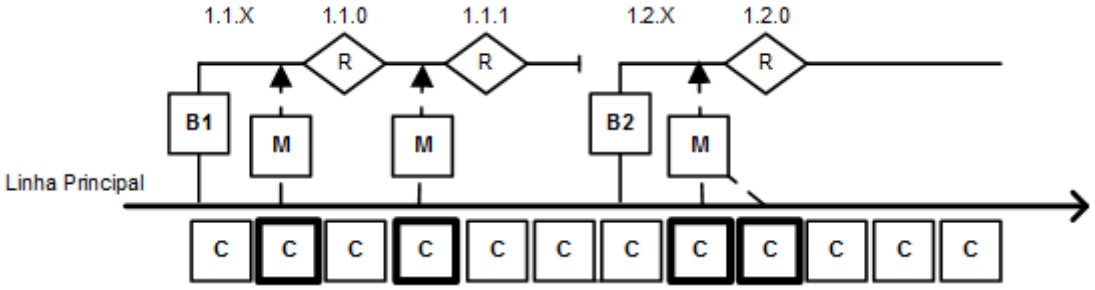


Figura 2.13: Repositório compartilhado na estratégia de TBD. Adaptado de Hammant (2013b).

Na linha principal foi criado o ramo B1 e realizada a mesclagem de um *commit* para a geração da versão 1.1.0. Uma nova mesclagem foi realizada gerando a versão 1.1.1. Um novo ramo foi criado (B2) e foi realizada a mesclagem de dois *commits* gerando-se a versão 1.2.0.

A IC exige o uso de uma linha principal. Segundo Humble e Farley (2010), o desenvolvimento baseado em tronco (ou linha principal) é uma condição para a IC:

> "(Desenvolvimento na Linha Principal) é uma forma extremamente eficaz de desenvolvimento e a única que permite a execução da integração contínua" (HUMBLE; FARLEY, 2010, tradução da autora).

2.5.4 Ramos para *release* no TBD

A Figura 2.14 apresenta uma alternativa no TBD, a criação de ramo para o release.

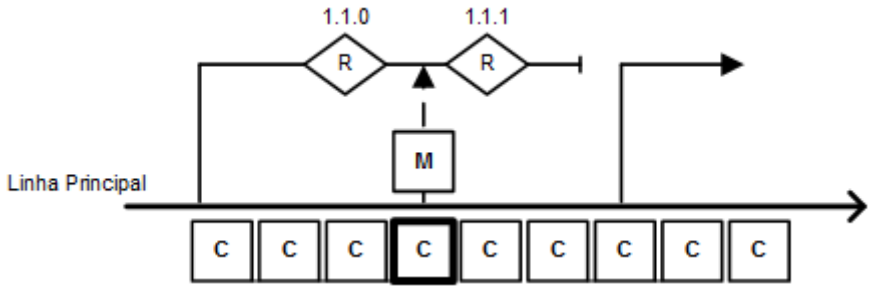


Figura 2.14: Ramos para release no TBD. Adaptado de Hammant (2014).

Todos os desenvolvedores realizam *commits* na linha principal. Um ramo é gerado para o *release*. É indicado alterar as permissões para que a maioria dos

desenvolvedores não realizem *commits* nos ramos. O primeiro *commit* no ramo pode tornar ele pronto para o *release* (HAMMANT, 2014).

Se um defeito for encontrado depois do *release* 1.1.0, por exemplo, a empresa não quiser lançar uma nova versão do produto com os últimos *commits* na linha principal, ela deve lança-la a partir do ramo de versão pré-existente (HAMMANT, 2014).

O *commit* em negrito na Figura 2.14 representa a correção realizada na linha principal. Após reproduzir o ocorrido e corrigi-la na linha principal, realiza-se o *commit* no ramo de versão e mesclagem. Se ocorrerem conflitos na mesclagem é necessário resolve-los prontamente (HAMMANT, 2014).

A correção na linha principal e a mesclagem no ramo de versão é uma maneira de garantir que regressões (ruins) não aconteçam. Regressões não devem ser confundidas com testes de regressão (HAMMANT, 2014).

3 INTEGRAÇÃO CONTÍNUA (IC)

3.1 Conceito

Fowler (2006a) define a Integração Contínua (IC) como:

> "[...] prática de desenvolvimento de software em que os membros de uma equipe integram seu trabalho frequentemente. Geralmente cada pessoa integra pelo menos diariamente, podendo ter múltiplas integrações por dia. Cada integração é verificada por um *build* automatizado (incluindo testes) para detectar erros de integração o mais rápido possível. Muitas equipes acham que essa abordagem leva a uma redução significante dos problemas de integração e permite que a equipe desenvolva software coeso mais rapidamente [...]" (FOWLER, 2006a, tradução da autora).

A IC com uma abordagem automatizada gera benefícios ao automatizar processos repetitivos e propensos a erros. Existem ferramentas que dão suporte à IC como um processo automatizado, usando um servidor de IC para automatizar as práticas da IC. No entanto, uma abordagem manual para a integração (usando build e testes automatizados) pode funcionar (DUVALL; MATYAS; GLOVER, 2007).

3.2 Motivações para a IC

O maior benefício da IC é reduzir riscos, uma vez que é difícil prever quanto tempo levará para finalizar o projeto (FOWLER, 2006a).

A IC procura resolver esse problema, ao permitir a rápida detecção de erros com o aumento da frequência de integrações. Erros no trabalho em andamento são mais difíceis de resolver (FOWLER, 2006a).

Para minimizar os erros de integração, a IC remove um dos maiores obstáculos à implantação frequente. A implantação frequente permite aos usuários obter novos recursos mais rapidamente, dar um *feedback* mais rápido sobre estes recursos e tornar mais colaborativo o ciclo de desenvolvimento. Isto ajuda a quebrar barreiras

entre o cliente e o desenvolvimento, essencial para o sucesso do desenvolvimento de software (FOWLER, 2006a).

A IC também tende a reduzir os custos da detecção de erros. Se uma abordagem contínua não for seguida, os períodos entre as integrações serão mais longos, tornando exponencialmente mais difícil encontrar e corrigir problemas. Os problemas de integração podem atrasar um projeto ou fazê-lo falhar.

IC, portanto, traz múltiplos benefícios para sua organização:

- Evita as longas e tensas integrações.
- Aumenta a visibilidade que permite uma maior comunicação.
- Permite uma rápida análise e correção.
- Minimiza o tempo de depuração liberando mais tempo para o trabalho nas novas funcionalidades.
- Minimiza o tempo de espera para descobrir se o código vai funcionar.
- Reduz os problemas de integração permitindo entregar o software mais rapidamente.

3.3 Cenário Típico da IC

A Figura 3.1 apresenta os componentes de um sistema de IC. Em um cenário típico de IC que utiliza esses componentes, um desenvolvedor submete suas alterações de código (*commit*) ao repositório de controle de versão. O servidor de IC busca (*pull*) alterações no repositório periodicamente. Após o *commit*, o servidor de IC detecta que aconteceram alterações no repositório de controle de versão e executa um script de *build* que integra o software. O servidor de IC gera o resultado dessa execução (*feedback*) e envia à equipe do projeto. O processo se repete sempre que o servidor de IC detecta uma alteração no repositório de controle de versão (DUVALL; MATYAS; GLOVER, 2007).

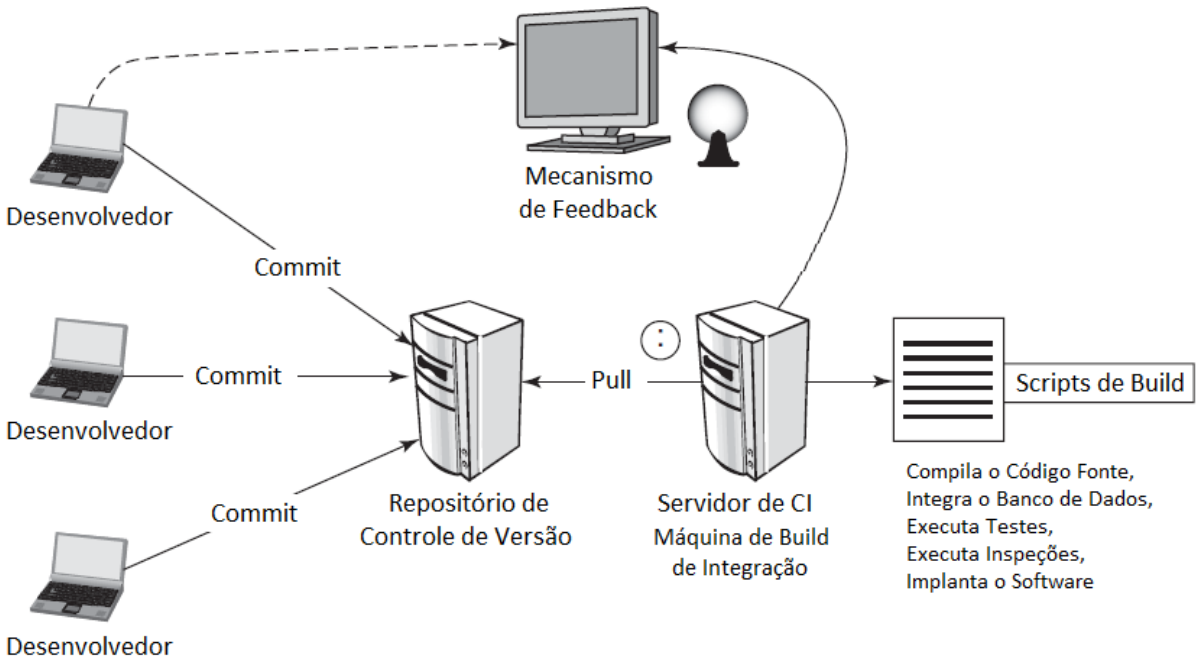


Figura 3.1: Os componentes de um sistema de IC (DUVALL; MATYAS; GLOVER, 2007).

Os componentes típicos de um sistema de IC são, portanto (DUVALL; MATYAS; GLOVER, 2007):

- **Desenvolvedor:** uma vez que o desenvolvedor realiza alterações em suas tarefas, ele executa um *build* particular, integrando suas alterações com as da equipe e submetendo as alterações ao repositório de controle de versão.
- **Repositório de Controle de Versão:** tem por finalidade gerenciar as mudanças no código fonte e outros artefatos de software usando um repositório de acesso controlado. Um repositório de controle de versão permite voltar no tempo e obter diferentes versões do código fonte e outros arquivos. O repositório de controle de versão é essencial mesmo em projetos que não usam IC.
- **Servidor de Integração Contínua:** executa o *build* de integração sempre que uma mudança é submetida ao repositório de controle de versão. O servidor irá recuperar os arquivos alterados e executar os scripts de *build*. É possível configurar a periodicidade dessa execução. Normalmente, o servidor fornece o resultado do *build*. Um servidor de IC é recomendado, mas não é necessário, pois os desenvolvedores podem criar scripts e executar

manualmente o *build* de integração sempre que acontece uma alteração no repositório.

- ***Script* de *Build*:** é composto por um ou mais scripts usados para compilar, testar, inspecionar e implantar o software. É possível usar script de *build* sem um sistema de IC.
- **Mecanismo de *Feedback*:** um dos propósitos da IC é fornecer *feedback* após a execução do *build* de integração, sendo importante saber se existe algum problema com o *build*. Ao receber essas informações é possível corrigir o problema imediatamente. Um e-mail ou serviço de mensagem (SMS) podem ser usados como mecanismos de *feedback*.
- **Máquina de *Build* de Integração:** é uma máquina separada cuja responsabilidade é integrar o software. Essa máquina hospeda o servidor de IC.

3.4 Características Necessárias para a IC

As características necessárias para a IC são (DUVALL; MATYAS; GLOVER, 2007):

- Uma conexão com o repositório de controle de versão;
- Um script de *build*;
- Algum tipo de mecanismo de *feedback*;
- Um processo para integrar as alterações do código fonte.

Uma vez que o *build* automatizado é executado a cada mudança no sistema de controle de versão, outras características são identificadas no sistema de IC (DUVALL; MATYAS; GLOVER, 2007). A Figura 3.2 apresenta essas caracteristicas.

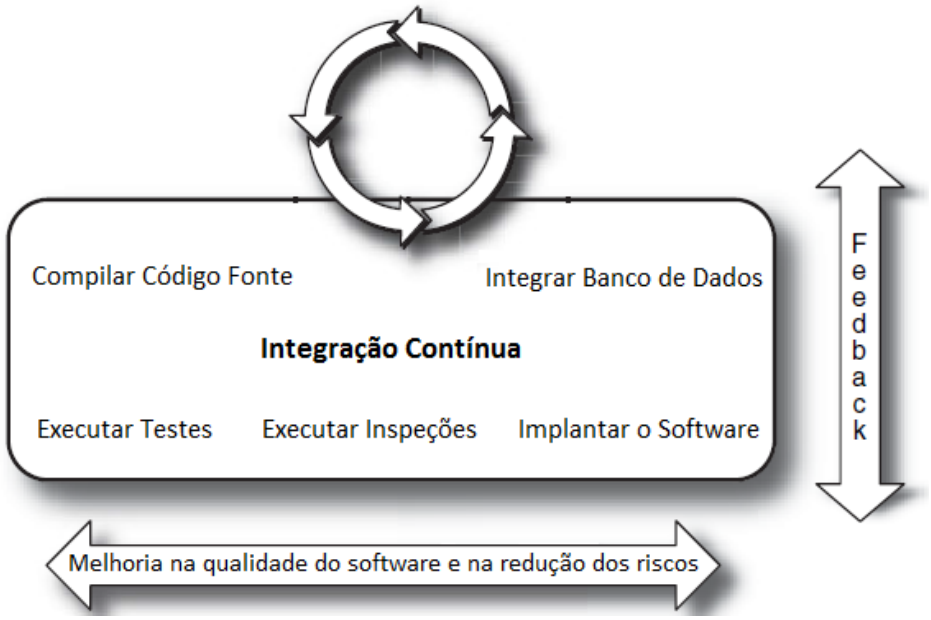


Figura 3.2: Características no Sistema de Integração Contínua (DUVALL; MATYAS; GLOVER, 2007).

Compilação do Código Fonte: é a criação do código executável a partir do código fonte.

Integração do Banco de Dados: o banco de dados é uma parte integrante do software. Ao utilizar um sistema de IC é possível garantir a integração do banco de dados através do repositório de controle de versão. A Figura 3.3 apresenta a integração do banco de dados no processo de *build* de um sistema de IC. Quando um membro da equipe do projeto modifica um script de banco de dados e submete-o para o sistema de controle de versão, o mesmo script de *build* que integra o código fonte integrará o banco de dados.

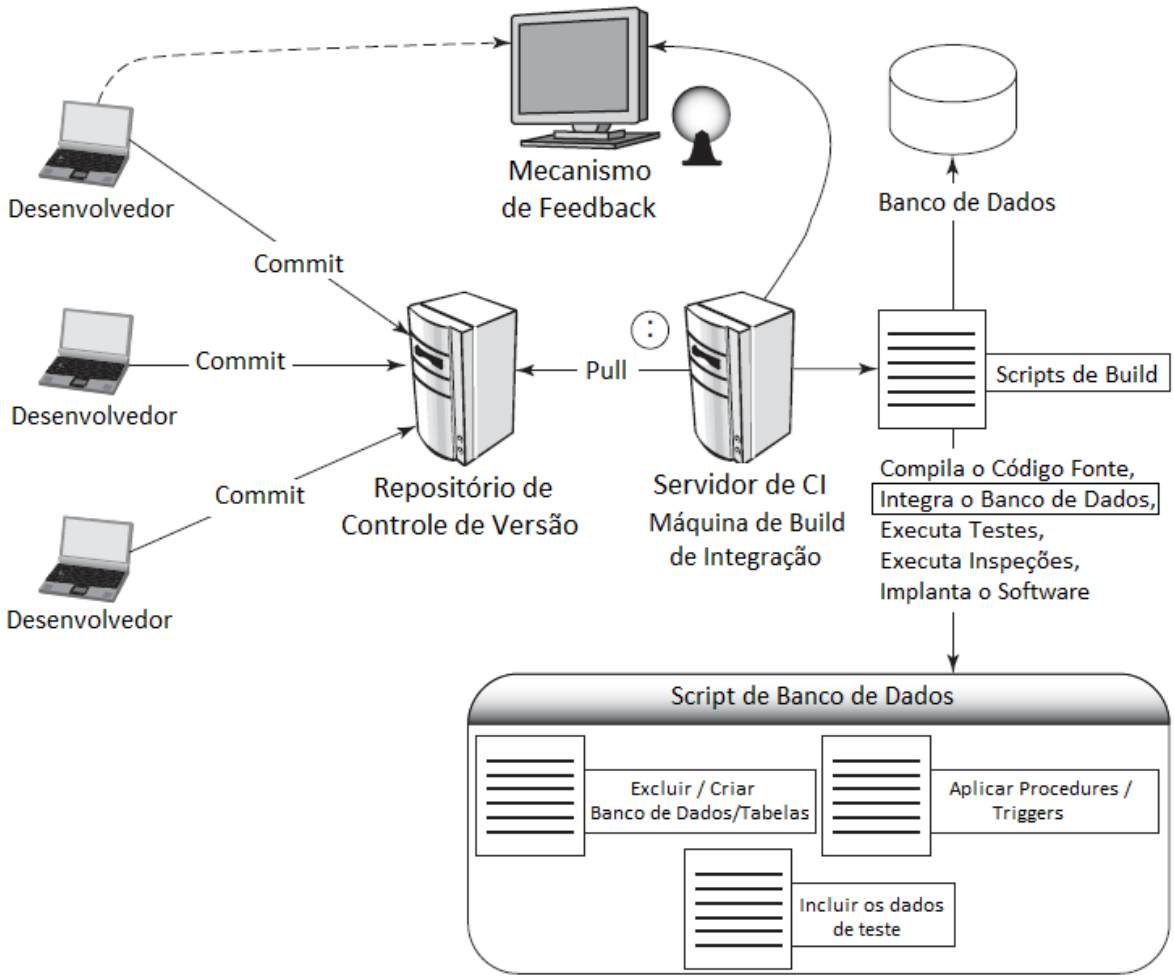


Figura 3.3: Integração do Banco de Dados (DUVALL; MATYAS; GLOVER, 2007).

Teste: sem o uso de testes automatizados é difícil fornecer confiança aos desenvolvedores e partes interessadas do projeto de que o software esteja funcionando apropriadamente. Existem diferentes categorias de testes em um sistema de IC, tais como, testes de unidade, componente, sistema, segurança, desempenho e outros.

Inspeção: inspeção automatizada (análise estática e em tempo de execução) de código pode ser usada para melhorar a qualidade do software através da análise do código e aplicação de regras.

Implantação (*Deployment*): permite a entrega de artefatos de software a qualquer momento. O principal objetivo do sistema de IC é gerar artefatos de software em pacotes com as últimas alterações de código e disponibilizá-los em um ambiente de teste. Com relação aos arquivos do repositório de controle de versão, o *build* deve

ser realizado, todos os testes e inspeções devem ser executados com sucesso, e o *release* deve ser rotulado.

Documentação e *Feedback*: um sistema de IC possibilita o uso de ferramentas para a geração da documentação. Além de ferramentas que possibilitam a geração de diagramas e outras informações baseadas no código fonte do repositório do controle de versão. A essência de um sistema de IC é fornecer um *feedback* rápido para os desenvolvedores e partes interessadas no projeto.

3.5 Práticas

Na implantação de um ambiente de IC é importante investir nas mudanças no comportamento e na disciplina da equipe. As principais práticas que devem ser executadas por indivíduos e equipes em um projeto são (DUVALL; MATYAS; GLOVER, 2007) (FOWLER, 2006a):

- **Submeter Frequentemente:** os desenvolvedores devem submeter o código fonte alterado ao repositório de controle de versão a cada pequena tarefa de poucas horas realizada. Ao fazer isso com frequência, os desenvolvedores descobrem rapidamente se há conflitos. Conflitos que permanecem despercebidos por semanas são mais difíceis de resolver.
- **Não Submeter Código Quebrado:** não submeter código com erro para o repositório de controle de versão.
- **Corrigir *Build* Quebrado Imediatamente:** o build quebrado pode ser um problema de compilação, teste, inspeção, implantação ou com o banco de dados. Quando ocorre esse tipo de problema em um ambiente de IC a correção deve ser imediata.
- **Escrever Testes Automatizados:** O *build* deve ser totalmente automatizado. Escrever os testes em um *framework xUnit* (*JUnit* ou *NUnit*) para a ferramenta de IC executar esses testes de forma automatizada.
- **Todos os Testes e Inspeções Devem Passar:** em um ambiente de IC todos os testes e inspeções automatizados devem passar no *build*. Aceitar um

código que não passa em testes ou inspeções pode resultar em um funcionamento indevido e qualidade inferior do software.

- **Executar *Build* Privado:** para evitar quebrar o *build,* os desenvolvedores devem emular a execução dos *builds* em suas máquinas locais e verificar se há erros, antes de submeter o código ao repositório.

- **Evitar Obtenção de Código Quebrado:** se o *build* estiver quebrado não obter a versão do repositório de controle de versão até que esteja corrigida.

- **Manter um Repositório Único de Código Fonte:** controlar os arquivos do projeto de software é um esforço necessário quando várias pessoas estão envolvidas. Todos os arquivos devem ser armazenados em um repositório de controle de versão que deve ser de conhecimento da equipe.

- **Automatizar o *Build*:** transformar o código fonte em um sistema executável é um processo que envolve compilação, movimentação de arquivos, carga na base de dados, testes e outras atividades. No entanto, a maioria das atividades podem ser automatizadas. Ferramentas como o *Ant*, o *Nant* e o *MSBuild*, por exemplo, podem ser utilizadas para automatizar o *build*.

- **Testes Automatizados no *Build*:** uma maneira de descobrir rapidamente os erros no sistema é incluir testes automatizados no processo de *build*. O resultado da execução deve indicar se os testes que falharam. Os testes não provam a ausência de erros, mas auxiliam na sua detecção e correção.

- **Cada *commit* deve atualizar o repositório em um servidor de integração:** o servidor de integração contínua monitora o repositório. Toda vez que um *commit* é realizado no repositório, o servidor automaticamente verifica os código fonte, inicia o *build* e emite notificação em caso de falha.

- **Manter o *Build* Rápido:** um ponto importante na IC é fornecer *feedback* rápido. Projetos com build de até 10 minutos estão dentro do aceitável, cada minuto que se reduz é tempo ganho para o desenvolvedor. Ao submeter frequentemente é possível ganhar-se bastante tempo.

- **Testar em um ambiente idêntico ao de produção:** se o ambiente de teste for diferente do ambiente de produção, cada diferença resulta em um risco. O

ideal é configurar o ambiente de teste da mesma maneira que o de produção usando o mesmo software de banco de dados, com as mesmas versões e o mesmo sistema operacional.

- **Tornar fácil a obtenção da última versão do software:** os membros da equipe de desenvolvimento podem obter a última versão do software e executá-lo para demonstrações, testes ou ver as mudanças realizadas.
- **Implantação automática:** scripts permitem implantar o software em qualquer ambiente mais facilmente. A implantação automática permite a execução do software em ambientes, tais como de teste, homologação ou produção, ajudando a reduzir o tempo de implantação e a incidência de erros.

A IC tem como alguns dos seus principais fundamentos (DUVALL; MATYAS; GLOVER, 2007):

- **Redução dos Riscos:** ao integrar muitas vezes ao dia é possível reduzir os riscos no projeto, uma vez que facilita a detecção de defeitos, mede a saúde do software e reduz a necessidade de suposições.
- **Os defeitos são detectados e corrigidos mais cedo:** com o uso da IC, testes e inspeções são executados muitas vezes ao dia, havendo uma grande possibilidade de se descobrir defeitos quando eles são introduzidos ao invés de mais tarde durante o ciclo de testes.
- **A saúde do software é mensurável:** ao incorporar testes e inspeções no processo automatizado de integração, atributos de estado do software podem ser monitorados ao longo do tempo.
- **Redução de suposições:** ao reconstruir e testar o software sempre que ocorre uma mudança no repositório de controle de versão é possível reduzir suposições sobre o funcionamento do software.
- **Redução de Processos Repetitivos**: automatizar os processos manuais e repetitivos ajuda a reduzir o trabalho manual e a capacidade de superar a resistência de membros da equipe para implementar melhorias ao utilizar mecanismos automatizados.

- **Geração de Software Executável:** com a IC é possível disponibilizar o software a qualquer momento. É possível também fazer mudanças no código fonte e integrar com o resto do código. Se acontecer algum problema, os membros do projeto são informados e as correções são feitas imediatamente.
- **Melhoria da Visualização do Projeto:** IC fornece informações dos estados de *build* e métricas de qualidade do projeto oferecendo a percepção de tendências e eficiência na tomada de decisão.
- **Estabelece Maior Confiança no Produto:** a aplicação das práticas de IC pode fornecer uma maior confiança na produção do produto de software. A cada *build* a equipe sabe que os testes são executados no software para verificar o comportamento e o cumprimento de padrões de codificação do projeto. Desenvolvedores tendem a ter mais confiança em fazer mudanças no software, uma vez que o sistema de IC pode informá-lo quando algo está errado.

3.6 Pré-requisitos para a IC

Alguns pré-requisitos devem ser cumpridos antes de implantar a IC (DUVALL; MATYAS; GLOVER, 2007):

- **Controle de Versão:** tudo deve ser submetido a um único repositório de controle de versão: código, testes, scripts de banco de dados, scripts de build e implantação, e qualquer coisa necessária para criar, instalar, executar e testar a aplicação. Existem vários sistemas de controle de versão simples, eficazes, leves e livres. Muitos projetos usam a estratégia de controle de versão de ramo de funcionalidade para gerenciar grandes equipes, mas é impossível ter IC ao usar ramos, por definição, ao trabalhar em um ramo o seu código não será integrado com o de outros desenvolvedores, uma vez que o processo de integração é executado na linha principal.
- ***Build* Automatizado:** é possível realizar o *build* inicialmente através de linha de comando. Independente do mecanismo, uma pessoa ou um computador

podem executar o *build*, testes e processo de implantação de forma automatizada através de linha de comando. Ferramentas de desenvolvimento de software e de IC tornaram-se bastante sofisticadas, sendo possível realizar o *build* do software e executar testes sem o uso de linha de comando.

- **Acordo da Equipe:** a IC é uma prática, não é uma ferramenta. Ela exige comprometimento e disciplina da equipe de desenvolvimento. É necessário que todos submetam pequenos incrementos de mudanças à linha principal e concordem que a tarefa de maior prioridade no projeto é corrigir qualquer alteração que quebre o software. Se as pessoas não adotam a disciplina necessária, suas tentativas de IC não levarão à melhoria da qualidade esperada.

- **Testes automatizados:** é essencial ter um nível de testes automatizados para fornecer a confiança de que a aplicação está funcionando. Testes de unidade (definido na seção 2.4.6), de componentes, de regressão (definido na seção 2.4.7) e de aceitação são considerados essenciais. Testes de componentes testam o comportamento de diversos componentes da aplicação, requerendo normalmente a execução de banco de dados, um sistema de arquivos ou outros sistemas. Esses testes costumam levar mais tempo que os testes de unidade. Testes de aceitação ou testes funcionais testam se a aplicação está de acordo com os critérios de aceitação definidos pelo negócio, incluindo as funcionalidades e características como: disponibilidade e segurança. Testes de aceitação normalmente são executados em toda a aplicação, em um ambiente similar ao de produção, o que pode levar bastante tempo.

- **Espaço de Trabalho Privado:** a definição é apresentada na seção 2.4.2.

- **Submeter Frequentemente:** a prática mais importante da IC é submeter o código frequentemente na linha principal, pelo menos algumas vezes ao dia. Detalhes a respeito são apresentados na seção 3.5.

- **Manter o *Build* e Testes Rápido:** como apresentado na seção 3.5, idealmente o *build* não deve demorar mais do que alguns minutos. Os testes em um *framework xUnit* (*JUnit* ou *NUnit*), por exemplo, fornecem o tempo que

cada teste levou. Deve-se descobrir quais testes estão lentos, e se há uma maneira de otimizá-los. É útil incorporar um conjunto de testes de fumaça para certificar que as funcionalidades mais usadas não quebraram.

4 LISTA DE VERIFICAÇÃO PARA A IMPLANTAÇÃO DA INTEGRAÇÃO CONTÍNUA

As listas de verificação foram criadas para auxiliar as empresas na preparação do ambiente para a implantação da IC. As listas de verificação são instrumentos simples que indicam as ações necessárias antes de iniciar a implantação da IC. A aplicação das listas avalia se os requisitos estão sendo cumpridos e o que ainda é necessário implantar na empresa avaliada. Este capítulo apresenta duas listas de verificação: uma para a verificação dos elementos implantados da gerência de configuração e outra para a verificação dos elementos requeridos para a implantação da integração continua.

Um erro na estratégia de implantação da IC ou até o não cumprimento de um requisito pode levar toda empreitada ao fracasso. Muitas tentativas de implantação acabam dando errado porque não existe uma lista de verificação dos requisitos da implantação ou porque itens da lista não foram atendidos.

4.1 Integração Contínua e Gerência de Configuração de Software

Esta seção apresenta algumas informações sobre a relação entre integração contínua e a gerência de configuração, baseadas em Humble e Farley (2010).

A gerência de configuração de software é a base de tudo. É impossível ter IC, gerência de *release* e implantação de *build* sem a gerência de configuração. A GCS produz um impacto positivo na colaboração da equipe. Não é apenas uma questão de escolher e implantar as ferramentas, embora isto seja importante, é fundamental aplicar os padrões essenciais da gerência de configuração.

Embora os sistemas de controle de versão sejam as ferramentas mais óbvias na GCS, a decisão de usar uma é apenas o primeiro passo no desenvolvimento de uma estratégia de gerência de configuração. A estratégia de GCS irá determinar como gerenciar todas as mudanças que acontecem no projeto. Ela registra a evolução dos sistemas e aplicações.

A GCS é extremamente importante. Se todos os artefatos do projeto não estiverem em um repositório de controle de versão, não se poderá desfrutar dos benefícios da IC. Em suma, para reduzir o tempo do ciclo do software e aumentar a qualidade na IC e testes automatizados é necessário que tudo relacionado ao projeto esteja em um repositório de controle de versão.

O ideal é submeter frequentemente as novas funcionalidades desenvolvidas na linha principal do repositório de controle de versão. Isso possibilita o uso da IC, que mantém o software funcionando e integrado em todos os momentos. Com a IC, o software é sempre testado, reduzindo as possibilidades de conflitos de mesclagem, garantindo que os problemas de integração sejam encontrados imediatamente, quando são baratos para corrigir, resultando em software de alta qualidade.

Para a IC funcionar bem é necessária uma gerência de configuração cuidadosa, não apenas do código fonte, mas também dos dados de teste, scripts de banco de dados, scripts de *build* e scripts de implantação. A revisão mais recente e boa (que funciona e passou nos testes automatizados) deve ser o ponto de partida para as codificações. É de vital importância à gerência de configuração de dependências de terceiros, bibliotecas e componentes, ter a versão correta para o código que está trabalhando.

Implantar a IC força a seguir três práticas: boa gerência de configuração, criação e manutenção de *build* automatizado, e um processo de testes. Para algumas equipes, parece muita coisa, mas isso pode ser feito incrementalmente.

4.2 Condições Necessárias para Implantar a Gerência de Configuração

As informações sobre as condições necessárias para a implantação da GCS são baseadas em (BERCZUK; APPLETON, 2002).

Para a GCS ajudar no trabalho cooperativo de uma equipe é necessário entender como todas as partes do ambiente de desenvolvimento interagem umas com as outras e como as técnicas da GCS podem se encaixar no desenvolvimento de software.

Como se sabe, para desenvolver software são necessários: a definição dos requisitos, elaboração do projeto e do código do produto, a realização de testes e a produção de documentação. Mas o mais importante é uma comunicação eficaz. Comunicação não é apenas o estado do compartilhamento e informações gerais, mas o compartilhamento de informações detalhadas sobre o que as pessoas estão fazendo para que as equipes possam trabalhar juntas e ser mais produtivas.

A escolha das ferramentas não deve ser a principal preocupação. As ferramentas ajudam a tornar as coisas melhores. Mas o mais importante é equilibrar a capacidade das ferramentas com as necessidades da organização e dos desenvolvedores. É fundamental tornar os processos fáceis para que as pessoas sigam.

Uma GCS eficaz é realizada através de padrões. Eles podem ser aplicados incrementalmente, de modo que não é necessário mudar toda a empresa para melhorar a maneira que se trabalha. Os padrões foram apresentados na seção 2.4. É importante entender os padrões antes de utilizá-los. Os detalhes de como aplica-los podem variar, dependendo do tamanho e natureza da equipe. Pode-se resumir os principais padrões vistos anteriormente, como segue:

- **Controle de Versão:** parece óbvio, mas algumas organizações não controlam as versões. Toda equipe precisa ter uma maneira de fazer o controle de versão e precisa usá-lo para comunicar as alterações de código entre os membros (HUMBLE; FARLEY, 2010).

- **Espaço de Trabalho Privado:** o desenvolvimento de software acontece no espaço de trabalho do desenvolvedor. O espaço de trabalho deve conter tudo que o desenvolvedor precisa para executar suas tarefas, incluindo o código fonte, componentes de *build* e componentes de terceiros. A seção 2.4.2 apresenta os detalhes desse padrão.

- ***Build* Privado de Sistema:** o *build* deve ser realizado no espaço de trabalho privado e deve-se certificar que as mudanças não quebram o *build* ou falham o teste de fumaça. A seção 2.4.3 apresenta os detalhes desse padrão.

- ***Build* de Integração:** ao realizar o trabalho local é necessário incorporá-lo ao resto do sistema em um *build* de integração. A seção 2.4.4 apresenta os detalhes desse padrão.

- **Teste de Fumaça:** permite verificar se as mudanças não quebram a funcionalidade do sistema. Idealmente, deve ser executado depois de cada *build* (*Build* Privado de Sistema e *Build* de Integração), antes do *check-in* para o repositório de controle de versão. A seção 2.4.5 apresenta os detalhes desse padrão.

- **Teste de Unidade:** verifica se os módulos que estão sendo alterados ainda funcionam de forma adequada. Deve ser executado antes de realizar o *check-in* da mudança e depois de atualizar o código com a versão atual, ou ao tentar encontrar um problema em um teste de fumaça, de regressão ou em resposta a um problema reportado pelo usuário. A seção 2.4.6 apresenta os detalhes desse padrão.

- **Teste de Regressão:** são executados no sistema quando se quer garantir a estabilidade da linha de desenvolvimento, podendo ocorrer antes da realização do *build* de um *release* ou de uma mudança arriscada. A seção 2.4.7 apresenta os detalhes desse padrão.

O *build* de integração deve ser integrado com frequência. Isto é, deve-se ter um processo de *build* periódico de modo que as pessoas possam ver o resultado. Quanto mais se adia a integração, mais os problemas se tornam difíceis de resolver.

O teste de fumaça é provavelmente o teste mais importante quando já se tem o teste de unidade funcionando de fato. O teste de fumaça dá a confiança de que o software realmente funciona. Se o software não funcionar, ele deve dar um diagnóstico básico (HUMBLE; FARLEY, 2010).

Se existirem muitos processos manuais é recomendado o uso de ferramentas, pois as pessoas podem cometer erros ou ignorar algum passo.

4.3 Lista de Verificação para a Implantação da Gerência de Configuração

O critério utilizado para a escolha dos itens que compõem a lista de verificação para a implantação da GCS foi pragmático: os principais padrões da GCS foram enumerados na seção anterior e descritos com detalhes no capítulo 2. Como justificado anteriormente, não se pode obter sucesso na implantação da IC se não houver uma sólida base de GCS.

No presente trabalho, não se analisam as prioridades dos itens, ou dos padrões e, portanto, do encadeamento das condições para uma implantação incremental. Esse aspecto deverá ser estudado futuramente. Apesar disso, é evidente por tudo o que foi dito até aqui, que o controle de versão é uma pré-condição primordial para a implantação de qualquer outro padrão da GCS.

Na lista, a coluna “Itens” indica os itens relativos a um determinado padrão que devem ser verificados. Optou-se por uma verificação binária simples: a marcação na coluna “Sim” significa a existência do item, e na coluna “Não” o caso contrário. A coluna “Comentários” é um espaço para detalhar o item em questão.

Para a definição dos itens da lista de verificação dos padrões da GCS necessários para a implantação posterior da integração contínua. Os requisitos de cada um dos padrões (BERCZUK; APPLETON, 2002) (DUVALL; MATYAS; GLOVER, 2007) (HUMBLE; FARLEY, 2010):

- Controle de Versão: para gerenciar as mudanças no código fonte e outros artefatos é necessário um sistema de controle de versão e um repositório de controle de versão com acesso controlado. Exemplos de sistemas de Controle de versão: *CVS*, *Subversion, Perforce, StarTeam, ClearCase, AccuRev*. Os desenvolvedores devem submeter frequentemente as novas funcionalidades na linha principal do repositório. Uma solução é o uso da estratégia de controle de versão TBD, apresentada em detalhes na seção 2.5.3.

- Espaço de Trabalho Privado: o desenvolvedor deve criar um espaço de trabalho com os artefatos, como, por exemplo, código fonte, componentes de terceiros e bibliotecas. O mecanismo do espaço de trabalho deve identificar se existe uma nova versão de um elemento ou um novo componente.

- *Build* Privado de Sistema: a equipe deve executar o *build* privado antes do envio do código para o repositório. O *script* de *build* e envio das mudanças para o repositório pode ser executado através de linha de comando ou de uma ferramenta de *build*, por exemplo, *Ant, NAnt, MSBuild* e *Rake*.

- *Build* de Integração: quando o desenvolvimento local está pronto, é necessário incorporá-lo com o resto do sistema por um *build* de integração. O *build* de integração deve ser automatizado, ser executado em um servidor separado que terá a responsabilidade de integrar o software e ter um mecanismo de notificação configurado que divulgue os resultados da integração.

- Teste de Fumaça: é recomendado executar um script automatizado para certificar que a aplicação está funcionando. O teste de fumaça deve verificar se quaisquer serviços que a sua aplicação depende estão funcionando, tais como banco de dados ou serviço externo. Pode ser executado na máquina do desenvolvedor antes do envio das mudanças ou antes do teste de aceitação.

- Teste de Unidade: deve ser automatizado, executado por uma ferramenta gráfica ou de linha de comando. Pode ser executado no espaço de trabalho como parte do *build* privado, com o uso, por exemplo, do *framework* de teste *xUnit*.

- Teste de Regressão: deve ser automatizado e executado antes de cada envio do código para o repositório.

A ideia básica, portanto, da lista de verificação é se ter um quadro do estado atual da GCS numa empresa de desenvolvimento de software.

Algum nível de teste automatizado deve ser realizado para se certificar que o produto funciona. Este teste pode variar de superficial a completo, de acordo com as necessidades e possibilidades (HASS, 2002).

A Tabela 4.1 apresenta a lista de verificação proposta para a implantação da GCS.

Tabela 4.1: Lista de Verificação para a Implantação da GCS.

Padrões	Itens	Sim	Não	Comentários
1. Controle de Versão	Sistema de controle de versão instalado no ambiente compartilhado			
	Repositório de controle de versão criado			
	Uso da estratégia Desenvolvimento Baseado em Tronco			
2. Espaço de Trabalho Privado	Espaço de trabalho privado criado no ambiente de desenvolvimento			
3. *Build* Privado de Sistema	Uso de linha de comando ou ferramenta de *build* instalada no ambiente de desenvolvimento			
	Scripts de *build* criados			
4. *Build* de Integração	Ferramenta de *build* instalada no servidor de integração			
	Scripts de *build* criados			
	Mecanismo de notificação configurado			
5. Teste de Fumaça	Teste de fumaça automatizado			
	Scripts de testes escritos			
6. Teste de Unidade	Uso de linha de comando ou ferramenta.			
	Suíte de testes criado			
7. Teste de Regressão	Teste de regressão automatizado.			
	Suíte de testes criado			

4.4 Condições Necessárias para Implantar a Integração Contínua

Como foi apresentado e descrito no capítulo 3, a implantação da IC com sucesso requer requisitos adicionais dos materializados na lista de verificação da GCS. Para que a IC seja eficaz algumas práticas ser implantadas previamente, como visto, a IC depende de a equipe seguir algumas práticas essenciais (HUMBLE; FARLEY, 2010). Conforme descrito na seção 3.6, as práticas essenciais são:

- Controle de Versão;
- *Build* Automatizado;
- Acordo da Equipe;

- Testes automatizados;
- Espaço de Trabalho Privado;
- Submeter Frequentemente;
- Manter o *Build* e Testes Rápido.

4.5 Lista de Verificação para a Implantação da Integração Contínua

A lista de verificação para a implantação da IC define os itens que devem ser observados. Na lista de verificação, os itens referem-se as práticas da IC acima enumeradas. A lista de verificação para a implantação da IC deve ser aplicada após a implantação dos itens da lista de verificação para a implantação da GCS, uma vez que existe uma dependência.

Para a definição dos itens da lista para a implantação da IC foram identificados os requisitos que devem ser cumpridos para cada uma das práticas. Os requisitos de cada uma das práticas (BERCZUK; APPLETON, 2002) (DUVALL; MATYAS; GLOVER, 2007) (HUMBLE; FARLEY, 2010):

- Submeter Frequentemente: a equipe deve realizar o *check-in* na linha principal com frequência.
- Acordo da Equipe: a equipe deve ter disciplina, comprometimento e seguir as práticas da IC, tais como, submeter pequenos incrementos de mudanças frequentemente para a linha principal e concordar que a tarefa de maior prioridade no projeto é corrigir qualquer alteração que quebre o software.
- Manter o *Build* e Testes Rápidos: a equipe deve analisar regularmente quanto tempo o *build* e os testes estão levando e ver se há uma maneira otimiza-los. Algumas ferramentas fornecem essas informações, assim como existem técnicas para reduzir o tempo de *build*.
- Teste de Componente: deve ser automatizado. Pode ser feito com o uso de *frameworks* de teste *xUnit, DbUnit* e *NDbUnit*, se envolver um banco de dados.

- Teste de Aceitação: deve ser automatizado com o uso de ferramentas, como, *Selenium* (para aplicações Web) e *Abbot* e executado quando o sistema está em ambiente de teste similar ao de produção.

Os requisitos das práticas: controle de versão, espaço de trabalho privado, *build* de integração, teste de fumaça, teste de unidade e teste de regressão, não foram apresentadas uma vez que foram definidos na seção 4.3.

A Tabela 4.2 apresenta a lista de verificação proposta para a implantação da IC.

Tabela 4.2: Lista de Verificação para a Implantação da IC.

Práticas	**Itens**	**Sim**	**Não**	**Comentários**
1. Controle de Versão	Sistema de controle de versão instalado no ambiente compartilhado			
	Repositório de controle de versão criado			
	Uso da estratégia Desenvolvimento Baseado em Tronco			
2. Espaço de Trabalho Privado	Espaço de trabalho privado criado no ambiente de desenvolvimento			
3. *Build* de Integração	Ferramenta de *build* instalada no servidor de integração			
	Scripts de *build* criados			
	Mecanismo de notificação configurado			
4. Teste de Fumaça	Teste de fumaça automatizado			
	Scripts de testes escritos			
5. Teste de Unidade	Uso de linha de comando ou ferramenta.			
	Suíte de testes criado			
6. Submeter Frequentemente	A equipe realiza o *check-in* na linha principal com frequência.			
7. Acordo da Equipe	A equipe está de acordo com as práticas da Integração Contínua			
8. Manter o *Build* e Testes Rápido	A equipe verifica regularmente quanto tempo o *build* e os testes demoram			
	A equipe otimiza o *build* e os testes quando necessário			
9. Teste de Componente	Teste de componente automatizado			
	Suíte de testes criado			

Práticas	Itens	Sim	Não	Comentários
10. Teste de Regressão	Teste de regressão automatizado.			
	Suíte de testes criado			
11. Teste de Aceitação	Teste de aceitação automatizado em um ambiente de teste			
	Suíte de testes criado			

Os padrões da lista de verificação para a implantação da GCS são interpretados como a obtenção de uma prática, uma vez que os padrões são na realidade decorrentes das melhores práticas. A Figura 4.1 apresenta as dependências entre as práticas da lista de verificação para a implantação da IC e os padrões da lista de verificação para a implantação da GCS.

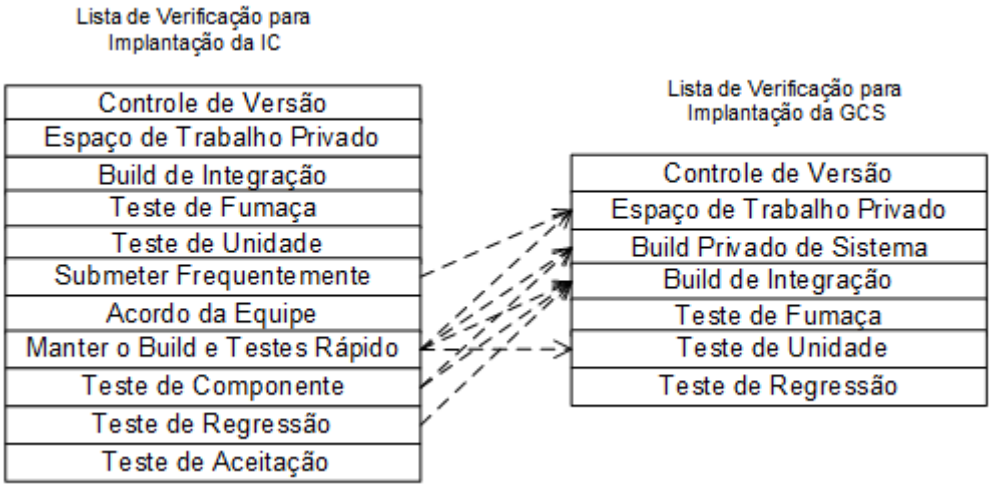


Figura 4.1: Dependências entre os itens das listas de verificação para a implantação da IC e os da GCS.

O Controle de Versão é um pré-requisito para todos os itens, por isso deve ser implantado inicialmente. Não foram representadas as dependências entre os itens em comuns das duas listas, uma vez que se implantados seguindo a lista para a implantação da GCS, não precisarão de nenhuma ação ao aplicar a lista para a implantação da IC.

O item Submeter Frequentemente depende da Estratégia de Controle de Versão e do Espaço de Trabalho Privado. Para submeter frequentemente, as tarefas devem ser menores, tornando menos provável a quebra do *build.* O *check-in* deve ser feito

na linha principal, pois não é aconselhável trabalhar em ramos. Por definição, se o desenvolvedor estiver trabalhando em ramos de seu código, ele não estará sendo integrado com o de outros desenvolvedores impossibilitando o uso da IC. Os desenvolvedores devem submeter as mudanças de seus espaços de trabalho privado à linha principal do repositório de controle de versão.

O item Acordo da Equipe não depende de nenhum dos itens da lista para a implantação da GCS, uma vez que exige atitudes (compromisso e disciplina) dos membros da equipe especifica da IC.

O item Manter o *Build* e Testes Rápidos depende do Espaço de Trabalho Privado, *Build* Privado de Sistema, *Build* de Integração e Teste de Unidade. Se o *Build* Privado de Sistema ou Teste de Unidade demorar, a equipe deixará de fazer esta atividade em seu espaço de trabalho privado ou fará *check-in* com menos frequência e terá mais *builds* quebrados.

O Teste de Componente pode depender do *Build* Privado de Sistema ou do *Build* de Integração. O Teste de Componente pode ser executado periodicamente ou como parte secundária do *Build de Integração*, independentemente, deve ser executado antes do *commit* do código para o repositório (no *Build* Privado do Sistema) (DUVALL; MATYAS; GLOVER, 2007).

O Teste de Regressão depende do *Build* de Integração. O Teste de Regressão faz parte do *build* e pode ser executado a cada *check-in* no repositório de controle de versão (DUVALL; MATYAS; GLOVER, 2007).

O Teste de Aceitação se inicia no fim do teste de componentes, quando os componentes individuais do software já foram testados e corrigidos, o software está organizado como um pacote e os erros de interface já foram descobertos e corrigidos.

4.6 Ramo de Funcionalidade e Integração Contínua

O uso de ramo de funcionalidade descrito na seção 2.5.2 é incompatível com a IC. Mesmo com a realização de práticas da IC em todos os ramos, de fato os ramos não

serão integrados. O mais próximo que se pode chegar do conceito de IC é se ter um sistema de IC que mescla todos os ramos na linha principal e executa testes automatizados na linha principal. Mas a mesclagem pode falhar na maioria das vezes, o que demonstra bem o problema do uso de ramo de funcionalidade na IC (HUMBLE; FARLEY, 2010).

Um dos princípios da IC é que todos devem submeter na linha principal a cada dia. Portanto, somente se um ramo durasse menos de um dia seria compatível com a IC (FOWLER, 2009). A IC diz que toda mudança deve ser submetida o mais rápido possível na linha principal. A linha principal é sempre o mais completo e atualizado estado de um sistema, uma vez que ele será implantado a partir dela. Quanto mais tempo as mudanças são mantidas separadas da linha principal, mais trabalhosa será a mesclagem. Quando se cria um ramo, assume-se que existe um custo associado a ele. Esse custo pode aumentar o risco de conflitos na mesclagem, e a única forma de minimizar este risco é garantir que o ramo criado seja mesclado com a linha principal diariamente ou mais frequentemente. Sem isso, o processo não pode ser considerado baseado em IC (HUMBLE; FARLEY, 2010).

No desenvolvimento baseado em tronco (TBD) apresentado na seção 2.5.3, os desenvolvedores sempre realizam o *check-in* na linha principal, quase nunca criam ramos e realizam o *check-in* algumas vezes ao dia (NAIK, 2015). Segundo Fowler (2009), os desenvolvedores submetem na linha principal de modo que as funcionalidades crescem na linha principal. Com a IC a linha principal deve ser sempre consistente, possibilitando liberar a versão seguramente.

Com todos trabalhando na linha principal, o TBD aumenta a visibilidade sobre o que todos estão fazendo, aumentando a colaboração e reduzindo o esforço duplicado. A prática de realizar o *check-in* com frequência menciona que se houver conflitos de mesclagem eles serão poucos. A resolução de problemas se torna mais fácil quando a alteração é menor (NAIK, 2015).

Em resumo, o TBD também implica a implantação de código a partir da linha principal, uma vez que ela deve sempre estar em um estado que possibilite a liberação para produção (NAIK, 2015).

5 EXEMPLO DE APLICAÇÃO DAS LISTAS DE VERIFICAÇÃO

Este capítulo apresenta a empresa XPTO, o seu processo de produção de *release* e o controle de versões de código e os testes funcionais realizados. Apresenta um exemplo da aplicação das listas de verificação propostas para a preparação do ambiente para a implantação da IC. O exemplo ajuda na análise de viabilidade das listas e mostra de forma prática sua aplicação, servindo de base para outras empresas que pretendam implantar a IC. Além disso, é apresentada a análise dos resultados, possibilitando verificar o estado atual da GCS da empresa e as ações necessárias para que os objetivos planejados sejam cumpridos.

5.1 Empresa XPTO

A XPTO é uma empresa de tecnologia da informação, especializada em desenvolvimento de software que oferece serviços de gestão de processos e documentos. O departamento de Tecnologia da Informação (TI) é dividido da seguinte maneira: 1 Diretor, 1 Líder de desenvolvimento, 9 Desenvolvedores, 2 Arquitetos de Informação (1 User Experience Design (UX) e 1 User Interface Design (UI)), 1 Líder de Infraestrutura, 2 Analistas de Infraestrutura, 1 Analista de Banco de Dados e 1 Analista de Segurança da Informação (SI). A estrutura hierárquica do departamento de TI é apresentada na Figura 5.1.

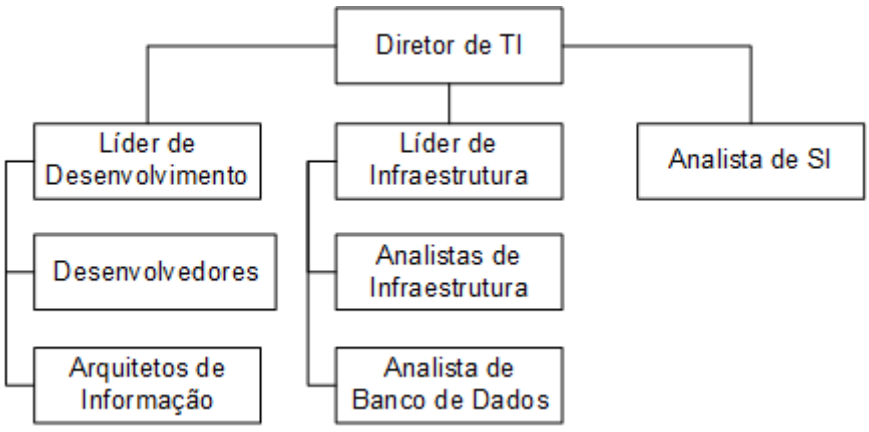


Figura 5.1: Organograma do Departamento de Tecnologia da Informação.

Os desenvolvedores possuem experiência nas linguagens de programação: ASP.NET e C#, assim como em banco de dados SQL Server.

As três ferramentas tecnológicas da empresa XPTO são X, Y e Z. X é uma aplicação *web* que tem como principal função a gestão de processos, consulta e armazenamento de documentos. Y é uma aplicação *desktop* que possibilita digitalizar documentos, anexar arquivos e realizar suas indexações. Z é um aplicativo para celular que possibilita fotografar documentos e realizar a indexação.

5.2 Processo de Produção de *Release*

Descreve-se, a seguir, a situação inicial com relação ao sistema X, que é desenvolvido e mantido pela equipe de Tecnologia da Informação. A Figura 5.2 apresenta o processo para produzir um *release* do sistema. Todos os requisitos são expressos como histórias do usuário, que são implementados como uma série de tarefas (SOMMERVILLE, 2007).

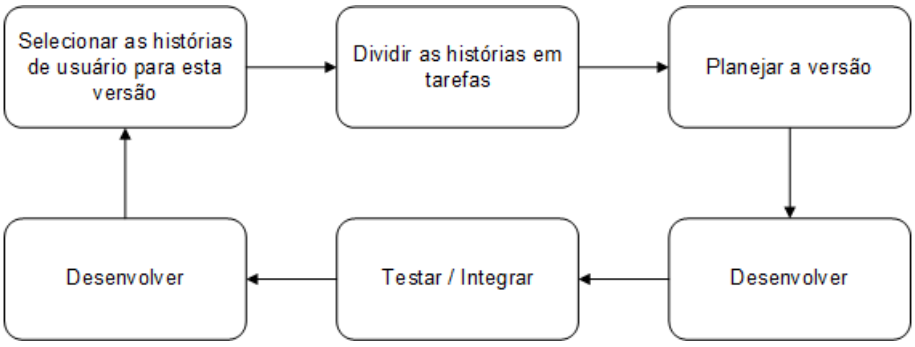


Figura 5.2: Ciclo de um *release*. Adaptado de Sommerville (2007).

Após o desenvolvimento de cada funcionalidade. O membro da equipe que desenvolveu a nova funcionalidade submete a funcionalidade ao repositório de controle de versão.

A Figura 5.3 apresenta o processo para entrega de um *release*, que ocorre a cada trimestre. A equipe publica a nova versão do sistema em um ambiente de homologação. Os testes funcionais de cada funcionalidade devem ser realizados e evidenciados pelos membros da equipe com a captura de imagens do passo a passo das telas. Se a nova funcionalidade não tiver erros é disponibilizada para os

clientes que solicitaram alguma das funcionalidades implementadas na versão. Após uma semana da versão nos clientes solicitantes, se nenhum erro for detectado a nova versão é disponibilizada para os demais clientes.

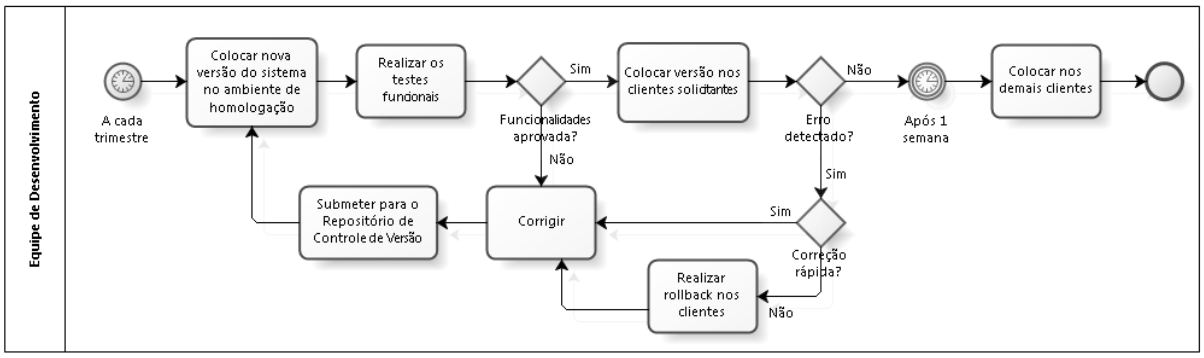


Figura 5.3: Processo de entrega de um release.

5.3 Controle de Versão

O controle de versão é realizado através do sistema de controle de versão TortoiseHg. O TortoiseHg é uma ferramenta gráfica e uma extensão de *shell* para o sistema de controle de versão distribuído Mercurial. É possível fazer o uso do TortoiseHg através da ferramenta ou de linha de comando (TORTOISEHG, 2015).

5.3.1 Repositório

Para adicionar um projeto no TortoiseHg é necessário criar um novo repositório. Para criação de um novo repositório informar o diretório de destino do projeto. O diretório do repositório que contém os arquivos é chamado de diretório de trabalho (TORTOISEHG, 2015).

Para incluir ou modificar algo no repositório é necessário fazer uma cópia do repositório no computador do desenvolvedor. O Mercurial invoca a opção de cópia de *clone*. Para realizar a cópia é necessário informar o diretório de origem (repositório central) e o de destino (TORTOISEHG, 2015).

A Figura 5.4 apresenta o repositório central que fica em um servidor e os repositórios dos desenvolvedores em seus respectivos computadores. O João incluiu um arquivo em seu diretório de trabalho e o Charles excluiu um arquivo. João e Charles são nomes fictícios de desenvolvedores.

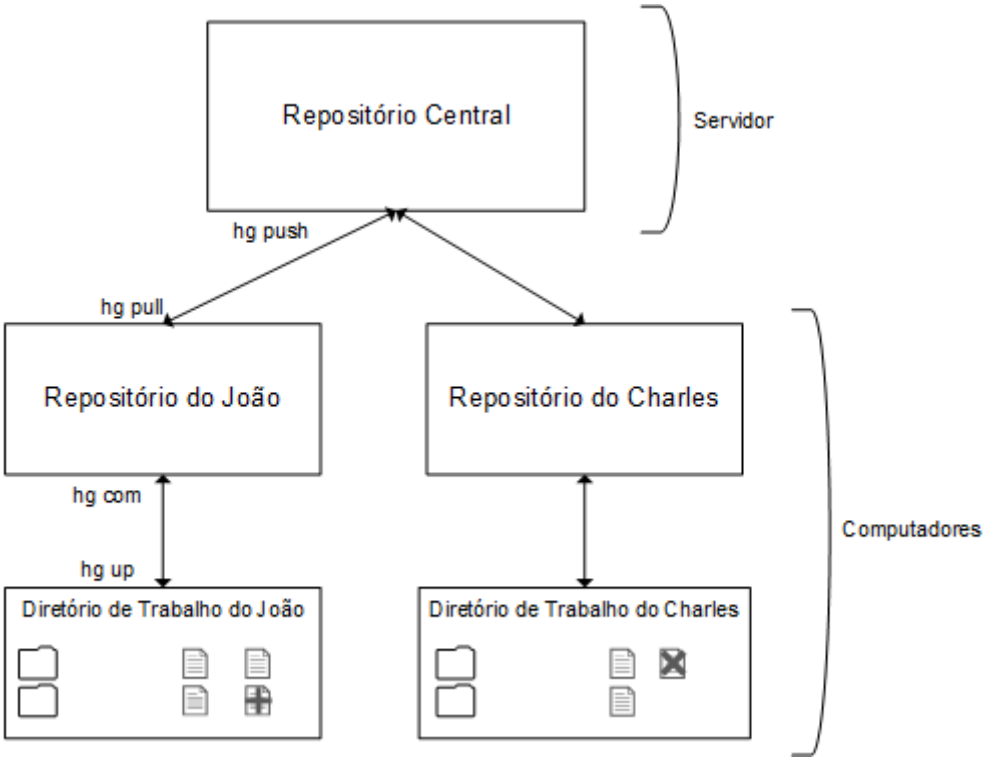


Figura 5.4: Repositórios no controle de versão do TortoiseHg.

Opções do TortoiseHg exibidas na Figura 5.4 (TORTOISEHG, 2015):

- **hg push:** envia as alterações do repositório local para o repositório central.
- **hg pull:** obtém as alterações mais recentes no repositório central para o repositório local.
- **hg com (commit):** salva localmente as alterações realizadas no repositório local.
- **hg up (update):** atualiza o diretório de trabalho do repositório com as alterações especificadas.

5.3.2 Fluxo de Trabalho

O fluxo de trabalho no TortoiseHg funciona da seguinte maneira:

1. Realizar a cópia (*hg clone*) do repositório central.
2. Realizar o *update* (*hg update*) na revisão do ramo de trabalho (se existir).
3. Realizar as mudanças.
4. Realizar o *commit* (*hg commit*) para persistir as mudanças localmente. Se for o primeiro *commit* de um novo projeto, cria-se o ramo (*hg branch*).
5. Repetir os passos 3 e 4 até obter um código aceitável.
6. Quando estiver pronto:
    a) Obter as mudanças feitas pelos outros (*hg pull*).
    b) Realizar a mesclagem (*hg merge*) e *update* (*hg update*).
    c) Testar para ter certeza de que não estragou nada. Se necessário, refaça os passos a e b.
    d) Realizar o *commit* (*hg commit*) da mesclagem.
    e) Enviar as mudanças para o repositório central (*hg push*).

5.3.3 Uso do Ramo de Funcionalidade

A estratégia de controle de versão utilizada pela empresa XPTO é a Ramo de Funcionalidade (definido na seção 2.5.2). As figuras 5.5 e 5.6 apresentam um exemplo do uso da estratégia na empresa XPTO.

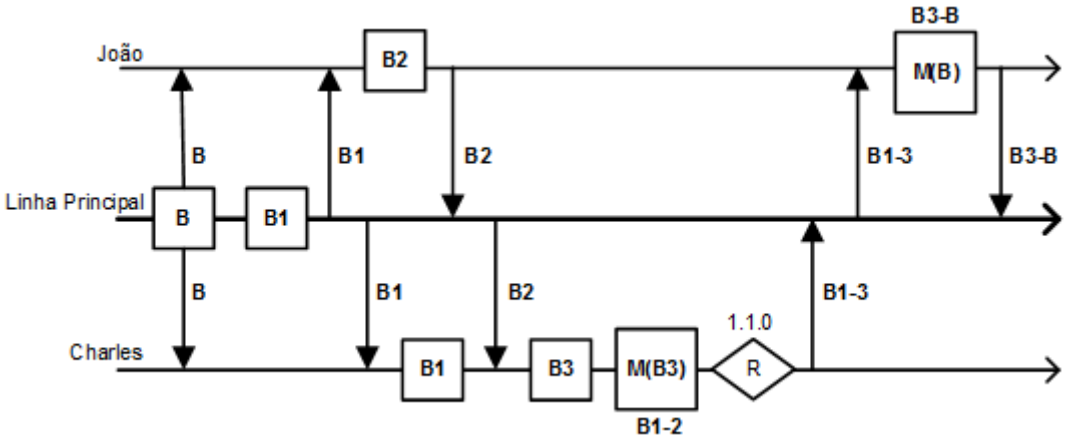


Figura 5.5: Cenário 1 - Desenvolvedores criando ramos de funcionalidades.

Descrição detalhada do cenário 1 apresentado na Figura 5.5:

- A linha central indica a linha principal do repositório central, a primeira linha o repositório do João e a última linha o repositório do Charles. O repositório do João e do Charles são cópias do repositório central.
- As setas podem representar o envio (*push*) ou a obtenção (*pull*) das revisões.
- As caixas (exemplo: B2) representam *commits* locais no ramo. As caixas (exemplo: M(B3)) indicam a mesclagem entre os ramos. A caixa B representa o ramo de revisão estável do sistema.
- O João e o Charles trazem as revisões da linha principal (ramo B1) para os respectivos repositórios.
- O João desenvolve uma funcionalidade e faz o *commit* no ramo B2. O João envia as revisões (ramo B2) para a linha principal.
- O Charles desenvolve no ramo B1. O Charles traz as revisões da linha principal (ramo B2) para o seu repositório. O Charles cria um novo ramo B3 para a mesclagem dos ramos B1 e B2, sendo que M(B3) indica a realização da mesclagem no ramo B3. O Charles envia as revisões para a linha principal.
- O novo release 1.1.0 (ramo B3) disponível na linha principal passa nos testes e é considerada estável, depois de disponibilizada em alguns clientes que não reportam nenhum problema. O João faz a mesclagem do ramo B3 com o ramo B (revisão estável) e envia a revisão para a linha principal.

- O João fecha os ramos com exceção do ramo B e envia a revisão para a linha principal. O ramo de funcionalidade deve ter vida curta. Os ramos fechados são extintos, não podendo receber novas revisões.

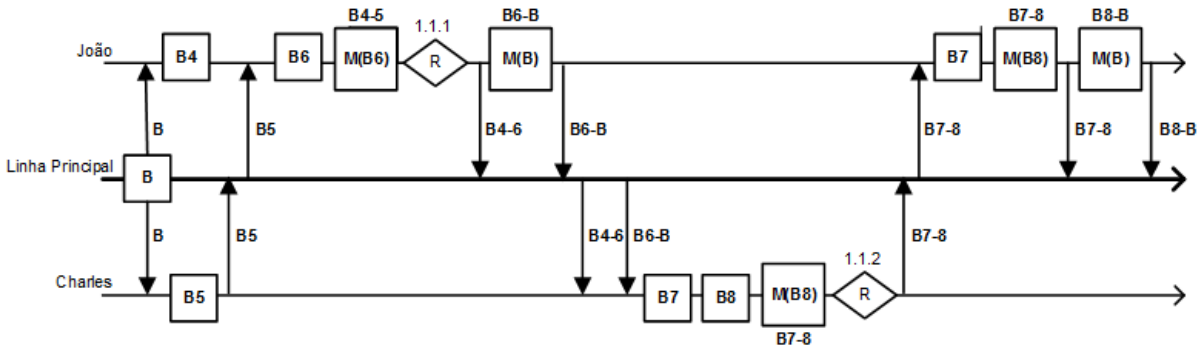


Figura 5.6: Cenário 2 - Conflitos de mesclagem nos ramos de funcionalidades.

Descrição detalhada do cenário 2 apresentado na Figura 5.6:

- O João faz o *commit* de uma funcionalidade no ramo B4.
- O Charles faz o *commit* de uma funcionalidade no ramo B5. O Charles envia as revisões (ramo B5) para a linha principal.
- O João traz as revisões da linha principal. Cria o ramo B6 para a mesclagem dos ramos B4 e B5. Ocorrem conflitos na mesclagem, uma vez o que o Charles e o João alteraram, por exemplo, a mesma classe. É necessário resolver os conflitos antes do envio para a linha principal. O João resolve os conflitos e envia as revisões para a linha principal.
- O novo release 1.1.1 (ramo B6) disponível na linha principal passa nos testes e é considerada estável, sendo depois de disponibilizada em alguns clientes que não reportam nenhum problema. O João faz a mesclagem do ramo B6 com o ramo B e envia a revisão para a linha principal.
- Ao encontrarem erros na versão em produção (release 1.1.1), o Charles traz as revisões da linha principal.

- O Charles realiza as correções no ramo B7. O Charles cria o ramo B8 para a mesclagem do ramo B7, gera o release 1.1.2 e envia as revisões para a linha principal.
- Nos testes realizados no ramo B8 da linha principal foram encontrados erros. O João traz as revisões da linha principal. O João realiza as correções no ramo B7, a mesclagem do ramo B7 no ramo B8 e envia as revisões para a linha principal.
- O release 1.1.2 (ramo B8) disponível na linha principal passa nos testes e é considerada estável. O João faz a mesclagem do ramo B8 com o ramo B, e envia a revisão para a linha principal.
- O João fecha os ramos, com exceção do ramo B e envia a revisão para a linha principal.

5.4 Testes Funcionais

Conforme apresentado na seção 5.2, os testes funcionais devem ser realizados e evidenciados pelos membros da equipe com a captura de imagens do passo a passo das telas. Os testes são realizados nas novas funcionalidades desenvolvidas, não existindo testes automatizados e testes de regressão. Apenas para exemplificar: se a funcionalidade “inclusão de validação de CPF no cadastro de clientes”, for disponibilizada pelo desenvolvedor João para testes no ambiente de homologação, o João prepara uma planilha com alguns testes que devem ser realizados, conforme apresentado na Tabela 5.1. Três membros da equipe devem realizar o teste guiado manualmente.

Tabela 5.1: Planilha de Testes.

Funcionalidade	Desenvolvedor	Responsável	Teste	OK (S/N)	Observação
Validação do CPF	João		1. Entre no cadastro de clientes		
			2. Preencha os campos obrigatórios e um CPF inválido. O sistema deverá notificar.		
			3. Digite um CPF válido e clique em confirmar.		
			4. Consulte o cliente cadastrado e veja se o CPF é o digitado.		

Os membros da equipe devem executar cada um dos passos e preencher a planilha informando o nome do responsável pelo teste, se o teste está ok (Sim ou Não). Se algum erro for encontrado, ele deve ser informado na observação. No caso de erro, a correção deve ser imediata e os testes realizados novamente por três membros, caso contrário é considerado que o teste passou.

5.5 Processo de Integração

Atualmente na empresa XPTO a integração é manual, não acontecendo com frequência e a entrega do produto é demorada. Existe uma falta de conhecimento dos membros da equipe do que está sendo integrado. Não existe um processo de *build*. Existem riscos de quebrar o *build* e tornar o código instável, comprometendo a qualidade do produto.

Não é possível prever quanto tempo a integração irá demorar, o que pode gerar atraso no projeto e insatisfação dos *stakeholders*. Como visto, a equipe usa o ramo de funcionalidades na estratégia de controle de versão, que é incompatível com a IC (conforme descrito na seção 4.6). Cria-se um ramo por funcionalidade e os ramos não são submetidos para a linha principal do repositório de controle de versão com frequência, podendo gerar mesclagens trabalhosas.

Os testes são realizados somente manualmente. Em algumas funcionalidades e nas novas funcionalidades desenvolvidas no *release*, testes incompletos podem deixar defeitos remanescentes. Além disso, como os testes são guiados manualmente, os desenvolvedores frequentemente realizam testes que repetem condições já

testadas, o que muitas vezes não contribui para o descobrimento de novos defeitos. Ao realizar correções, são realizados testes apenas nas funcionalidades corrigidas, a consequência geral é que a detecção de defeitos pode demorar e serem descobertos pelo cliente.

5.6 Aplicação das Listas de Verificação

A fim de evidenciar o uso das listas de verificação com a intenção de futuramente implantar a IC, as listas de verificação para a implantação da GCS (apresentada na seção 4.3) e IC (apresentada na seção 4.5) foram aplicadas na empresa XPTO pela própria autora do presente trabalho.

A princípio foi aplicada a lista de verificação para a implantação da GCS, uma vez que a implantação dos padrões é um pré-requisito para aplicação da lista de verificação para a implantação da IC, que depende dos itens da lista para a implantação da GCS.

A Tabela 5.2 apresenta a aplicação da lista de verificação para a implantação da GCS na empresa XPTO.

Tabela 5.2: Aplicação da Lista de Verificação para a Implantação da GCS.

Padrões	**Itens**	**Sim**	**Não**	**Comentários**
1. Controle de Versão	Sistema de controle de versão instalado no ambiente compartilhado	X		Usam o TortoiseHg
	Repositório de controle de versão criado	X		
	Uso da estratégia Desenvolvimento Baseado em Tronco		X	
2. Espaço de Trabalho Privado	Espaço de trabalho privado criado no ambiente de desenvolvimento	X		O desenvolvedor cria um repositório no seu computador, similar ao espaço de trabalho privado.
3. *Build* Privado de Sistema	Uso de linha de comando ou ferramenta de *build* instalada no ambiente de desenvolvimento		X	
	Scripts de *build* criados		X	
4. *Build* de Integração	Ferramenta de *build* instalada no servidor de integração		X	
	Scripts de *build* criados		X	
	Mecanismo de notificação configurado		X	
5. Teste de Fumaça	Teste de fumaça automatizado		X	
	Scripts de testes escritos		X	
6. Teste de Unidade	Uso de linha de comando ou ferramenta.		X	
	Suíte de testes criado		X	
7. Teste de Regressão	Teste de regressão automatizado.		X	
	Suíte de testes criado		X	

O ideal é aplicar a lista de verificação para a implantação da IC depois da implantação dos itens da lista para a implantação da GCS. A lista de verificação para a implantação da IC será aplicada na empresa XPTO apenas para exemplificar o uso da lista. Os itens em comum entre as duas listas têm o mesmo apontamento.

A Tabela 5.3 apresenta a aplicação da lista de verificação para a implantação da IC na empresa XPTO.

Tabela 5.3: Aplicação da Lista de Verificação para a Implantação da IC.

Práticas	**Itens**	**Sim**	**Não**	**Comentários**
1. Controle de Versão	Sistema de controle de versão instalado no ambiente compartilhado	X		Usam o TortoiseHg
	Repositório de controle de versão criado	X		
	Uso da estratégia Desenvolvimento Baseado em Tronco		X	
2. Espaço de Trabalho Privado	Espaço de trabalho privado criado no ambiente de desenvolvimento	X		O desenvolvedor cria um repositório no seu computador, similar ao espaço de trabalho privado.
3. *Build* de Integração	Ferramenta de *build* instalada no servidor de integração		X	
	Scripts de *build* criados		X	
	Mecanismo de notificação configurado		X	
4. Teste de Fumaça	Teste de fumaça automatizado		X	
	Scripts de testes escritos		X	
5. Teste de Unidade	Uso de linha de comando ou ferramenta.		X	
	Suíte de testes criado		X	
6. Submeter Frequentemente	A equipe realiza o *check-in* na linha principal com frequência.		X	
7. Acordo da Equipe	A equipe está de acordo com as práticas da Integração Contínua		X	
8. Manter o *Build* e Testes Rápido	A equipe verifica regularmente quanto tempo o *build* e os testes demoram		X	
	A equipe otimiza o *build* e os testes quando necessário		X	
9. Teste de Componente	Teste de componente automatizado		X	
	Suíte de testes criado		X	
10. Teste de Regressão	Teste de regressão automatizado.		X	
	Suíte de testes criado		X	
11. Teste de Aceitação	Teste de aceitação automatizado em um ambiente de teste		X	
	Suíte de testes criado		X	

5.7 Análise dos Resultados

Analisando os resultados apresentadas na aplicação da lista de verificação para a implantação da GCS (Tabela 5.2) na empresa XPTO, entende-se que existem várias práticas que não estão em uso para a implantação da IC e é possível identificar o que deve mudar para se possa implantar a IC.

Para se ter os padrões essenciais da GCS é necessário:

- Controle de Versão
- Espaço de Trabalho Privado
- Testes de Fumaça
- *Build* Privado de Sistema
- *Build* de Integração
- Testes de Unidade
- Testes de Regressão

A empresa XPTO já possui um sistema de controle de versão configurado e os desenvolvedores possuem repositórios locais em seus computadores (conforme apresentados na seção 5.3.1), similares aos espaços de trabalho privado. No entanto, haverá a necessidade de implantar uma série de ações para o atendimento dos padrões essenciais da GCS. As ações poderiam ser:

- Alterar a estratégia de controle de versão de Ramo de funcionalidade para o Desenvolvimento Baseado em Tronco, para que se tenha uma linha de desenvolvimento ativa.
- Implantar testes de unidade.
- Implantar um *build* privado de sistema e testes de fumaça no espaço de trabalho privado antes de submeter à mudança para a linha de desenvolvimento ativa do repositório de controle de versão.

- Implantar o *build* de integração e testes de regressão, o *build* deve passar nos testes de regressão.

Para alterar a estratégia de controle de versão de Ramo de funcionalidade para o Desenvolvimento Baseado em Tronco, é necessário alterar as permissões dos desenvolvedores para que não realizem *commits* em ramos do repositório compartilhado e definir com a equipe que passem a acontecer na linha principal. Atualmente na empresa XPTO, as tarefas podem levar dias, os *commits* das funcionalidades são realizados em ramos e quando as funcionalidades estão prontas são mescladas na linha principal. Como consequência, os ramos podem ter vida longa. Para realizar a alteração da estratégia, as tarefas devem ser menores para que ocorram *commits* frequentes na linha principal e menos conflitos na mesclagem. Outro impacto da estratégia atual de ramo de funcionalidade é que não existem testes automatizados. Ao realizar os *commits* das novas funcionalidades os desenvolvedores podem quebrar o *build*. Por essa razão, alguns testes deviam ser incluídos antes de submeter as funcionalidades do espaço de trabalho privado no repositório de controle de versão.

Os testes de unidade devem evoluir até que todos os métodos, classes e funções sejam testados. O teste de fumaça não precisa ser exaustivo. Os testes de regressão podem conter testes para alguns problemas que já aconteceram e outros que não podem acontecer e à medida que novos problemas forem detectados devem ser adicionados.

À medida que novas funcionalidades forem desenvolvidas, é recomendado estender os testes para essas funcionalidades, para que o processo se torne sustentável. Com os testes de regressão é possível encontrar efeitos colaterais ou instabilidades introduzidas pelas novas funcionalidades.

No caso, se as ações apresentadas forem implantadas os padrões essenciais da GCS serão atendidos, sendo necessário posteriormente aplicar as seguintes práticas essenciais da IC, não consideradas dentre os padrões apresentados:

- Submeter Frequentemente
- Acordo da Equipe

- Manter o *Build* e Testes Rápidos
- Teste de Componente
- Teste de Aceitação

Para a adoção da IC as ações poderiam ser:

- Acordo da equipe e submeter frequentemente são práticas que embora pareçam simples, exigem um entendimento, disciplina e compromisso dos membros equipe. A equipe deve trabalhar em pequenos e frequentes releases, a integração e o *build* deve ser frequente (HASS, 2002).
- Manter o *build* e testes rápidos é uma prática que deve ser considerada. É importante analisar se os *build* e testes não estão levando muito tempo para executar, e se isso estiver acontecendo, verificar maneiras de otimizar.
- Implantar o teste de componente e executar periodicamente antes do commit do código para o repositório no *build* privado do sistema.
- Implantar os testes de aceitação quando o pacote do software estiver pronto e passado em todos os testes e *build*, a execução deve ser em um ambiente similar ao de produção.

Desse modo, os requisitos para a implantação da IC serão cumpridos, possibilitando sua implantação. Ressalta-se que as ações apresentadas para o atendimento da GCS podem ser implantadas incrementalmente. As listas de verificação apresentam os itens que detalham as atividades necessárias para aplicação de cada uma das práticas, sendo que a ordem proposta na lista é apenas uma sugestão, as ações decorrentes devem ser consideradas no plano de implantação seguindo vários critérios como: custo, prazo, escopo, qualidade, entre outros, que devem organizar a prioridade dos itens a serem atendidos. No presente trabalho, como foi dito, não haverá uma ênfase para se determinar a ordem de prioridade dos itens a serem atendidos.

6 CONCLUSÃO

Este trabalho apresentou duas listas de verificação para auxiliar as empresas na preparação do ambiente para a implantação da integração contínua (IC). A lista de verificação para a implantação da Gerência de Configuração de Software (GCS) deve ser aplicada primeiro, uma vez que as práticas da lista para a implantação da IC dependem da existência dos principais padrões da GCS. Para isso, foram apresentados as práticas e os principais padrões da GCS, duas estratégias bem difundidas de controle de versão (Ramo de Funcionalidade e Desenvolvimento Baseado em Tronco) e os pré-requisitos para a implantação da IC. As listas de verificação fornecem um conjunto de itens a serem considerados para uma posterior implantação da IC. Os itens estão agrupados por práticas, sendo que cada item da lista é um ponto importante que deve ser verificado e, caso não esteja em uso, deve ser implantado.

Inicialmente, o resultado da aplicação das listas na Empresa XPTO evidenciou que a decisão de implantação da IC não pode ser uma decisão circunscrita ao nível de liderança de projetos. Isso porque ela exigirá investimentos importantes em tecnologia, treinamento e infraestrutura, considerando o estado atual da gerência de configuração. Um ponto crucial é que a diretoria da empresa deve estar comprometida com o patrocínio da implantação da IC para que ela tenha sucesso. O custo deverá ser compensado com a minimização de erros, sobretudo os que influenciam diretamente na integridade dos artefatos de software produzidos e na agilidade do processo de integração e entrega do software.

Apesar de não ter sido um objetivo imediato a implantação dos itens indicados nas listas de verificação e sua posterior análise no ambiente real da Empresa XPTO, o exemplo da aplicação das listas nesta empresa permitiu, mesmo que de modo informal e inicial, concluir que elas são viáveis de serem aplicadas. Certamente que essa viabilidade implica o conhecimento, por parte do responsável, dos conceitos e práticas da GCS e da IC expressos nos itens que devem ser verificados e implantados, bem como das dependências entre a IC e a GCS.

Em suma, a aplicação das listas de verificação mostrou que a empresa XPTO usa a estratégia de controle de versão de ramo de funcionalidade que é incompatível com

a IC, bem como não possui um processo de *build* e testes automatizados. Através da aplicação da lista de verificação para a implantação da GCS foi constatado que a empresa possui poucos padrões da GCS, nem sempre implantados de modo satisfatório. Nesse caso, seria recomendada a implantação de todos os padrões da lista de verificação para a implantação da GCS de modo adequado e, posteriormente, a implantação dos itens constantes na lista de verificação para a implantação da IC. Ressalte-se que a lista para a implantação da IC foi aplicada apenas para exemplificar o seu uso.

Não é necessário apenas um servidor de *build* ou de ferramentas de suporte à IC para que se tenha efetivamente a IC. Deve existir um processo que defina como integrar o software. Os membros da equipe devem submeter os códigos modificados várias vezes ao dia à linha principal do repositório de controle de versão. As práticas da gerência de configuração devem acontecer no dia a dia do desenvolvedor. Parece simples, mas a dificuldade está em garantir na prática diária que as modificações locais integrem com a versão existente e com as modificações feitas em paralelo por outros desenvolvedores. Em caso de falhas no *build*, a equipe deve parar o que está sendo feito e realizar a correção imediatamente até que o *build* se torne estável. Outro ponto relevante é a automatização de tarefas manuais que facilitam bastante processo de integração. Muitas vezes a implantação do software não é feita frequentemente devido à complexidade dos processos que são executados manualmente, complexidade na maioria das vezes decorrente de uma gerência de configuração inadequada.

Com os resultados das aplicações das listas pode-se estabelecer um plano de implantação da IC. A relevância dos itens deve ser esclarecida e justificada para que a equipe utilize as listas. As listas propostas permitem que não se esqueça de nenhum item e ajudam na organização e distribuição das tarefas pelos membros da equipe, uma vez que elas deixam claro o que é necessário implantar para que os objetivos de cada item sejam atingidos.

6.1 Trabalhos Futuros

A integração contínua é um assunto bastante vasto e depende dos padrões essenciais da gerência de configuração de software, que embora pareçam óbvios ainda são, aparentemente, pouco difundidos e compreendidos na prática do dia-a-dia das equipes de desenvolvimento. A tendência é que as empresas implantem IC em projetos de software, almejando aproveitar as motivações (apresentadas na seção 3.2) e aumentar a confiança na produção de um software com qualidade.

Existem alguns trabalhos futuros que podem ser realizados a partir da presente pesquisa. Dentre as possibilidades, destacam-se os seguintes trabalhos considerados os mais importantes para a continuidade da pesquisa:

- Avaliar as dificuldades da implantação dos padrões das listas de verificação da gerência de configuração e posteriormente da integração contínua.
- Realizar um experimento implantando a integração contínua após cumprir os requisitos indicados nas listas de verificação propostas.

Para a realização de um experimento controlado em uma empresa será necessário o desenvolvimento de um plano de implantação da IC. Primeiramente, deve-se aplicar a lista de verificação para a implantação da GCS e posteriormente da IC. Cada item das listas que não tiver em uso deve ser implantado. As listas serão validadas e ajustadas. Após o cumprimento dos requisitos das listas pode-se iniciar a implantação da IC. O histórico do controle de integridade e de defeitos de um projeto de software realizado nas condições operacionais vigentes na empresa poderia desempenhar o papel de grupo de controle. No decorrer do experimento, melhorias e ajustes nos processos realizados seriam identificados e apontados.

REFERÊNCIAS

ACM Digital Library. Disponível em: <http://dl.acm.org/>. Acesso em: 15 jun. 2015.

ABRAN, A.; J.W. MOORE; P. BOURQUE and R. DUPUIS. **Guide to the Software Engineering Body of Knowledge (SWEBOK)**. 2004. Piscataway, NJ, USA: The Institute of Electrical and Electronic Engineers, Inc. (IEEE).

AIELLO, BOB. **Where Did Configuration Management Go?**, 2015. Disponível em: <http://www.cmcrossroads.com/article/where-did-configuration-management-go>. Acesso em: 29 jan. 2016.

BAKAL, R.M.; ALTHOUSE, J.; VERMA, P. **Continuous integration in agile development**. IBM Developer Works. 2012. Disponível em: <http://www.ibm.com/developerworks/rational/library/continuous-integration-agile-development/index.html?ca=drs>. Acesso em: 26 jun. 2015.

BERCZUK, STEPHEN P.; APPLETON, BRAD. **Software Configuration Management Patterns: Effective Teamwork, Practical Integration**. Addison Wesley Professional, 2002.

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

FOWLER, M. **Continuous Integration**, 2006a. Disponível em: <http://martinfowler.com/articles/continuousIntegration.html>. Acesso em: 08 jun. 2015.

FOWLER, M. **Feature Branch**, 2009. Disponível em: <http://martinfowler.com/bliki/FeatureBranch.html>. Acesso em: 29 out. 2015.

FOWLER, M. **XUnit**, 2006b. Disponível em: <http://www.martinfowler.com/bliki/Xunit.html>. Acesso em: 03 jan. 2016.

HAMMANT, P. **Shades of Trunk Based Development**, 2014. Disponível em: < http://paulhammant.com/2014/09/29/shades-of-trunk-based-development/>. Acesso em: 14 dez. 2015.

HAMMANT, P. **The origins of Trunk Based Development**, 2015. Disponível em: <http://paulhammant.com/2015/04/23/the-origins-of-trunk-based-development/>. Acesso em: 30 out. 2015.

HAMMANT, P. **What is Your Branching Model?**, 2013a. Disponível em: <http://paulhammant.com/2013/12/04/what_is_your_branching_model/>. Acesso em: 15 dez. 2015.

HAMMANT, P. **What is Trunk Based Development?**, 2013b. Disponível em: <http://paulhammant.com/2013/04/05/what-is-trunk-based-development/>. Acesso em: 28 out. 2015.

HASS, ANNE METTE JONASSEN. **Configuration Management Principles and Practice**. Addison Wesley, 2002.

HUMBLE, J.; FARLEY, D. **Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation**. Addison-Wesley, 2010.

IEEE - **Standard for Configuration Management in Systems and Software Engineering**. IEEE Std 828-2012 (Revision of IEEE Std 828-2005).

IEEE Xplore. Disponível em: <http://ieeexplore.ieee.org/>. Acesso em: 10 jun. 2015.

ISO - International Organization for Standardization. **Systems and software engineering - Vocabulary**. ISO/IEC/IEEE 24765, 2010.

MOLITOR, M. **Chapter 1. Software Configuration. Management and Continuous Integration**. 2012. Disponível em: <http://sewiki.iai.uni-bonn.de/_media/teaching/labs/xp/2012b/seminar/6-scm.pdf>. Acesso em: 25 jun. 2015.

NAIK, VISHAL. **Enabling Trunk Based Development with Deployment Pipelines**, 2015. Disponível em: <https://www.thoughtworks.com/pt/insights/blog/enabling-trunk-based-development-deployment-pipelines>. Acesso em: 01 dez. 2015.

OMG - Object Management Group. **Business Process Model and Notation (BPMN)**. Version 2.0.2, formal/2013-12-09, 2013.

SAMPAIO, C. **Guia de Campo do Bom Programador**. Brasport, 2012.

SOMMERVILLE, I. **Engenharia de Software**. 8ª edição. Pearson, 2007.

SCHOLAR GOOGLE. Disponível em:<scholar.google.com>. Acesso em: 28 jul. 2015.

SPARK SYSTEM. Disponível em: <http://www.sparxsystems.com/enterprise_architect_user_guide/9.2/standard_uml_models/deploymentdiagram.html>. Acesso em: 04 ago. 2015.

THOUGHTWORKS. **Continuous Integration**. Disponível em: <https://www.thoughtworks.com/pt/continuous-integration>. Acesso em: 16 nov. 2015.

TORTOISEHG. Disponível em: <http://tortoisehg.bitbucket.org/>. Acesso em: 07 dez. 2015.

WTHREEX. **Ambiente de Desenvolvimento**, 2003. Disponível em: <http://www.wthreex.com/rup/process/workflow/environm/co_deven.htm>. Acesso em: 21 dez. 2015.

GLOSSÁRIO

Ambiente de desenvolvimento - em um projeto de desenvolvimento de software é o termo usado para todos os itens de que o projeto precisa para desenvolver e implantar o sistema, por exemplo, ferramentas, diretrizes, processos, modelos e infraestrutura (WTHREEX, 2003).

Artefatos - são produtos de trabalhos finais ou intermediários produzidos e usados durante os projetos, por exemplo, documento de arquitetura de software, modelo de casos de uso ou o modelo de análise (WTHREEX, 2003).

Framework - um projeto reutilizável (modelos ou código) que pode ser refinado (especializado) e estendido para fornecer uma funcionalidade geral de muitas aplicações (ISO, 2010).

Integração - o processo de combinação de componentes de software, componentes de hardware ou ambos em um sistema (ISO, 2010).

Script - a descrição de uma sequencia especifica de ações (ISO, 2010).

Sistema de Software - um sistema intensivo de software é o único componente a ser desenvolvido ou alterado (ISO, 2010).

xUnit**:** é o nome dado aos *frameworks* de testes automáticos unitários que se tornam amplamente conhecidos entre os desenvolvedores de software. O nome é uma derivação do *JUnit* (para Java). O *JUnit* se tornou o mais popular, mas foi seguido pelo *CppUnit* para C++ e *NUnit* para C# (FOWLER, 2006b).