GERÊNCIA DE CONFIGURAÇÃO E EVOLUÇÃO

AULA 4

Profª Adriana Bastos da Costa

CONVERSA INICIAL

Temos discutido ao longo dos nossos estudos que mudanças é algo inerente a qualquer software, pois as necessidades dos clientes mudam, as tecnologias evoluem e as tendências dos mercados precisam ser acompanhadas, para que a competitividade cresça e resulte em ganhos financeiros melhores. Todo software automatiza processos de uma empresa, processos esses que fazem girar o negócio e atingir os objetivos estratégicos de crescimento.

Seguindo essa premissa, é importante contar com uma ferramenta automatizada para controlar o versionamento da evolução e as mudanças dos itens de configuração, de forma a inserir mais organização e segurança na construção e manutenção dos softwares.

Nesta etapa vamos explorar mais detalhadamente o que são artefatos gerados ao longo de um projeto de software, além de compreender como ocorre o controle de versão desses artefatos. Vamos nos aprofundar no controle da configuração, das baselines e dos releases de um software, apoiado em uma ferramenta de controle de versão.

Esta etapa está dividida em cinco tópicos principais, sendo eles:

- Artefatos de projeto;
- Versionamento dos itens de configuração;
- Sistemas de controle de versão;
- Configuração de ambientes, baselines e release;
- Gerência de mudança de arquivos.

Vamos explorar os temas e entender como o processo de gerência de configuração automatizado melhora a produtividade da equipe de desenvolvimento e a qualidade do software construído.

TEMA 1 – ARTEFATOS DE PROJETO

A construção de um software envolve o entendimento do que precisa ser feito e a tradução das necessidades em uma solução técnica. É preciso analisar e documentar esse entendimento sobre as necessidades dos usuários, além de documentar a solução técnica projetada, em todas as suas partes. Como a

construção de um software é algo bastante complexo, quanto mais documentado a solução estiver, mais fácil será o entendimento de todos os envolvidos.

Mas a documentação de um software não deve ser feita sem um motivo claro e plausível. Documentar apenas por documentar não agrega valor, muito pelo contrário, gera prejuízo financeiro e de tempo.

Mas, afinal, o que é um artefato de software, que é o foco principal deste tópico? Bom, artefato é um dos vários tipos de subprodutos produzido durante o desenvolvimento de um software. Ou seja, todos os produtos de trabalho gerados ao longo do projeto, e mesmo depois dele, podem ser entendidos também como artefatos do software.

Por exemplo, em um determinado projeto podemos gerar os seguintes artefatos: casos de uso, diagramas de classes, diagrama de casos de uso, detalhamento dos requisitos, modelo da dados, entre outros. Esses artefatos têm o objetivo de ajudar a descrever a solução técnica, a arquitetura a ser utilizada e o design previsto para o software. Muitas vezes são criados, até mesmo para auxiliar a equipe técnica a pensar, a partir dos requisitos, na melhor solução para o software em questão.

Além dos artefatos técnicos, outros podem ser gerados durante o desenvolvimento, como planos de projetos, processos de negócios e avaliações de risco. São artefatos gerenciais que ajudam a planejar a construção do software, partindo do entendimento do que precisa ser feito e da complexidade dos requisitos.

Portanto, artefato é todo produto de trabalho gerado ao longo do desenvolvimento do projeto, tanto como apoio para a equipe que vai construir o software como para documentar o que está sendo construído.

Os artefatos gerados para o projeto são amplamente utilizados na manutenção do software, como uma fonte de entendimento sobre como o software foi criado, facilitando, assim, a manutenção e inserção de novos requisitos ou correção de defeitos.

Algumas vezes, os artefatos que precisam ser entregues ao cliente com o software no final do projeto são formalizados no contrato de desenvolvimento do software. Por isso mesmo, é preciso garantir que os artefatos evoluam junto com a evolução do software, para que estejam atualizados e úteis para uso pela equipe que vai fazer a manutenção do software, após ele entrar em produção.

Todos os artefatos do software precisam ser criados e gerenciados, seja manual ou automaticamente. Alguns deles, por conta de sua criticidade para o projeto, deverão ter um controle mais rígido tanto de criação, quanto de evolução ou manutenção. Em outros casos, não precisarão de tanta rigidez no controle. Essa é a diferença entre os artefatos considerados como itens de configuração e os que não são itens de configuração.

Vamos analisar o quadro a seguir, com artefatos gerados para um projeto.

Quadro 1 – Quadro de artefatos e itens de configuração

Artefato	Item de Configuração	Justificativa
Plano de Projeto	Não	Artefato alterado constantemente, sem impacto para os demais artefatos do projeto.
Planilha de Risco	Sim	Artefato que serve de insumo para o planejamento de custo, prazo e tarefas do projeto.
Cronograma	Não	Artefato alterado constantemente, sem impacto para os demais artefatos do projeto.
Requisitos	Sim	Artefato base para todo o projeto. Todos os demais artefatos dependem da integridade dele.
Casos de Uso	Sim	Artefato base para todos os artefatos relacionados com o modelo físico do software.
Modelo de Classe	Sim	Artefato base para a construção do software, alterações afetam diretamente na construção no código.
Modelo de Banco de Dados	Sim	Artefato base para a construção do software, alterações afetam diretamente na construção no código.
Código	Sim	Artefato base para o funcionamento do software. Precisa ser controlado para evitar defeitos e retrabalho.
Casos de Teste	Não	Artefato que não impacta outros artefatos do software.

Fonte: Costa, 2022.

O objetivo do Quadro 1 é organizar a decisão de quais artefatos serão tratados como itens de configuração, sendo regidos pelo processo de gerência de configuração. Essa é uma decisão bastante própria de projeto para projeto, portanto, o quadro apresentado é apenas um exemplo. As justificativas poderiam

alterar dependendo da criticidade do projeto e da estratégia de gerenciamento e controle definido pela equipe.

Esse quadro serve de insumo para as tarefas de auditoria de gerência de configuração, indicada pelo MPS.Br como uma boa prática para garantir que o processo de gerência de configuração está sendo seguido da forma como deveria. Ou seja, em momento de auditoria, é preciso consultar o quadro para verificar se todos os artefatos definidos como itens de configuração estão seguindo o processo de gerência de configuração de maneira completa e adequada.

No próximo tópico, vamos detalhar mais como os artefatos que também são itens de configuração devem passar por versionamento e devem ser controlados seguindo o processo de gerência de configuração.

TEMA 2 – VERSIONAMENTO DOS ITENS DE CONFIGURAÇÃO

Os artefatos críticos de um projeto são fortes candidatos a serem tratados como itens de configuração, tanto na sua construção quanto na sua manutenção. Todo item de configuração é controlado por meio do processo de gerência de configuração. Lembrando que o segundo resultado esperado do MPS.Br indica que é preciso definir critérios para selecionar os itens de configuração, conforme vimos quadro exemplo no tópico anterior.

Portanto, uma das tarefas de um processo de gerência de configuração envolve fazer o versionamento dos itens de configuração de maneira adequada. Mas além do versionamento, a gerência de configuração envolve outros subprocessos, vamos relembrar quais são:

- **Registro da evolução do projeto, também conhecido como controle de versão** – é o subprocesso que controla cada solicitação de mudança que é solicitada e implementada, para registrar o histórico do incremento do software. O incremento corresponde a uma configuração, que é o estado do conjunto e itens de configuração que foram o sistema em um determinado momento do tempo, por isso, não é algo estático. É a fotografia do momento. Mas as funcionalidades do subprocesso de controle de versão não se restringem ao registro do histórico das configurações, ele também possibilita a edição de maneira concorrente e em paralelo dos arquivos e a criação de variações no projeto, controlando

ramos de versionamento que não interferem entre si. Essas funcionalidades são essenciais para o trabalho em equipe, pois mais de um profissional pode trabalhar de maneira concorrente, sem que um afete o outro. Por exemplo, imagina que uma parte da equipe esteja finalizando em uma iteração e outra parte da equipe já esteja iniciando a próxima iteração, que evolui alguns requisitos da iteração anterior. Dessa forma, é possível trabalhar em paralelo, de maneira organizada, sem que uma equipe afete ou atrapalhe a outra. O controle de versão é a parte principal da gerência de configuração, sendo o elo comum entre o controle de mudança e a integração contínua do projeto;

- **Controle e acompanhamento de mudanças, também conhecido como controle de mudança** – já discutimos anteriormente que mudanças aparecem durante todo o desenvolvimento do software e devem ser registradas, avaliadas e organizadas de acordo com sua prioridade e necessidade para o negócio. Com base em tudo isso, é possível planejar o escopo, o prazo e o custo de cada iteração. Dessa forma, pode-se acompanhar o estado da solicitação da mudança, desde seu entendimento e construção, até sua implementação e o lançamento de uma versão em produção. O importante é gerenciar de forma adequada os riscos inerentes a uma mudança no software, analisando a rastreabilidade e identificando os possíveis impactos. Os riscos e impactos no software precisam ser eficientemente gerenciados, para minimizar os impactos nos negócios;

- **Estabelecimento da integridade do sistema, também conhecido como integração contínua** – a integração contínua tem como objetivo ir verificando, conforme o software vai sendo construído, se os itens de configuração estão corretos e estão se integrando de maneira adequada aos demais itens de configuração, para gerar um executável que atenda às necessidades dos usuários. A integração pressupõe uma boa estratégia técnica, que organize a ordem de construção dos componentes, para que estes sejam construídos e integrados de maneira contínua, conforme definição da equipe de desenvolvimento.

Vamos nos aprofundar um pouco mais no subprocesso de integração contínua. O subprocesso de controle de mudança já foi bastante discutido em

conteúdo anterior, e o subprocesso de controle de versão será explicado em profundidade no próximo tópico.

2.1 Integração contínua

Bom, já vimos que a integração contínua é a ação de integrar os componentes criados em um software constantemente, continuamente, para que o processo ocorra de maneira gradual, facilitando a organização do projeto e a identificação e correção de defeitos.

Portanto, integração contínua é uma prática de desenvolvimento de software em que os desenvolvedores, em espaços de tempo curtos e constantes, juntam suas alterações de código em um repositório central. Normalmente, a integração contínua se refere ao estágio de criação dos componentes de software. A frequência de integração deve ser definida pela equipe de desenvolvimento, dependendo da criticidade, tamanho e complexidade do software.

A integração contínua é uma prática ágil, da metodologia XP – *Xtreme Programing* (ou programação extrema, em português). O XP está baseado em boas práticas ágeis para a engenharia de software, em que uma delas é a construção colaborativa do software com a integração contínua.

Os principais objetivos da integração contínua envolvem encontrar e investigar defeitos de forma mais rápida, melhorar a qualidade do software e reduzir o tempo que leva para testar e disponibilizar novas versões do software.

Em termos práticos, a integração contínua é feita por *scripts* que automatizam o processo de integração dos componentes que estão sendo construídos e disponibilizados para a fase de teste do ciclo de vida de desenvolvimento, pela criação de uma versão compacta do software.

As ferramentas utilizadas para realizar a integração contínua acompanham as ferramentas de controle de versão e disparam os *scripts* a cada vez que uma nova configuração é registrada, ou em horários programados, dependendo da definição dada pela equipe de desenvolvimento. Perceba, então, que além de um processo definido e organizado, é preciso ter disciplina para manter a integração dos componentes que vão sendo construídos de maneira contínua e em intervalos regulares.

TEMA 3 – SISTEMA DE CONTROLE DE VERSÃO

O controle de versão é uma forma de controlar os componentes de software, quando são criados e quando são alterados, mantendo o histórico de tudo o que ocorre ao longo do tempo. Muitos problemas de desenvolvimento de software são causados por falta de controle de versão, exatamente porque sem o controle do que está sendo alterado é muito comum inserir defeitos no código ou mesmo gerar retrabalho, seja por perda de código, seja por alteração mal planejada, que precisa ser desfeita ou refeita, para atender às necessidades do negócio.

A falta de um sistema de controle de versão em um projeto de construção ou manutenção de software pode gerar problemas como: um profissional pode sobrescrever o código de outro profissional por acidente, perdendo as alterações já realizadas; perder o controle de quais alterações já foram efetuadas em um programa, quando foram feitas e quem fez; ter dificuldade para recuperar o código de uma versão anterior do componente ou componentes que estão em produção; e ter problemas para manter variações do sistema ou manutenção em paralelo.

Portanto, para evitar problemas como esses e melhorar a qualidade geral da construção do software, é preciso definir, implementar e utilizar um processo de controle de versão, de preferência, automatizado por meio de uma ferramenta, melhorando também a produtividade da equipe de desenvolvimento.

De uma forma resumida, o subprocesso de controle de versão apoia o desenvolvimento do software de diversas maneiras, como:

- **registro do histórico** – processo para registrar toda a evolução do projeto, cada alteração sobre os componentes e arquivos que compõe o software. Com essas informações, sabe-se quem fez o que, quando e onde, facilitando o gerenciamento da construção e da manutenção dos requisitos;
- **colaboração concorrente** – o processo de controle de versão possibilita que vários desenvolvedores trabalhem em paralelo, inclusive, nos mesmos arquivos sem que um sobrescreva o código de outro. Essa capacidade evita o reaparecimento ou surgimento de defeitos e perda de funcionalidades;

- **variações no projeto** – processo que mantém linhas diferentes de evolução do mesmo projeto, permitindo dividir a equipe em trabalhos concorrentes, por exemplo, sendo possível manter uma versão 1.0 enquanto a equipe prepara uma versão 2.0 do software.

Enfim, o processo de controle de versão é fundamental para o desenvolvimento de software, por isso, vários ambientes de desenvolvimento já possuem integração com alguns sistemas de controle de versão, automatizando o desenvolvimento integrado com a gerência de configuração.

Vamos entender em detalhes como funciona o controle de versão.

3.1 Como funciona o controle de versão

Para entender o funcionamento, vamos primeiro entender como o controle de versão é composto. Existem duas partes básicas que se conectam e fazem o processo funcionar adequadamente, são elas: o **repositório** e a **área de trabalho de cada integrante da equipe de desenvolvimento**.

O **repositório** armazena o histórico de evolução do projeto, registrando toda e qualquer alteração feita em cada um dos itens de configuração versionados. Ele deve ser único e centralizado para todo o projeto, com a equipe de desenvolvimento tendo acesso para manter o controle de versão contínuo, de acordo com a periodicidade acordada entre a equipe.

O desenvolvedor não trabalha diretamente nos arquivos do repositório, eles são utilizados para integrar o código construído. Para manter a organização e a integridade do código, o desenvolvedor usa uma **área de trabalho** que contém a cópia dos arquivos do projeto e que é monitorada para identificar as mudanças realizadas. Essa área é individual e isolada das demais áreas de trabalho, com cada desenvolvedor tendo a sua.

O relacionamento entre a área de trabalho e o repositório do projeto ocorre com comandos de *COMMIT* (acesso) ou *UPDATE* (atualização), que são executados pelo desenvolvedor, tanto para buscar um arquivo do repositório e alterá-lo na sua área de trabalho local, quanto para salvar sua atualização no repositório. A área de trabalho fica localizada, normalmente, na máquina de trabalho do desenvolvedor.

Portanto, a sincronização entre a área de trabalho e o repositório central é feita pelos comandos *commit* e *update*. O *commit* tem como objetivo enviar um

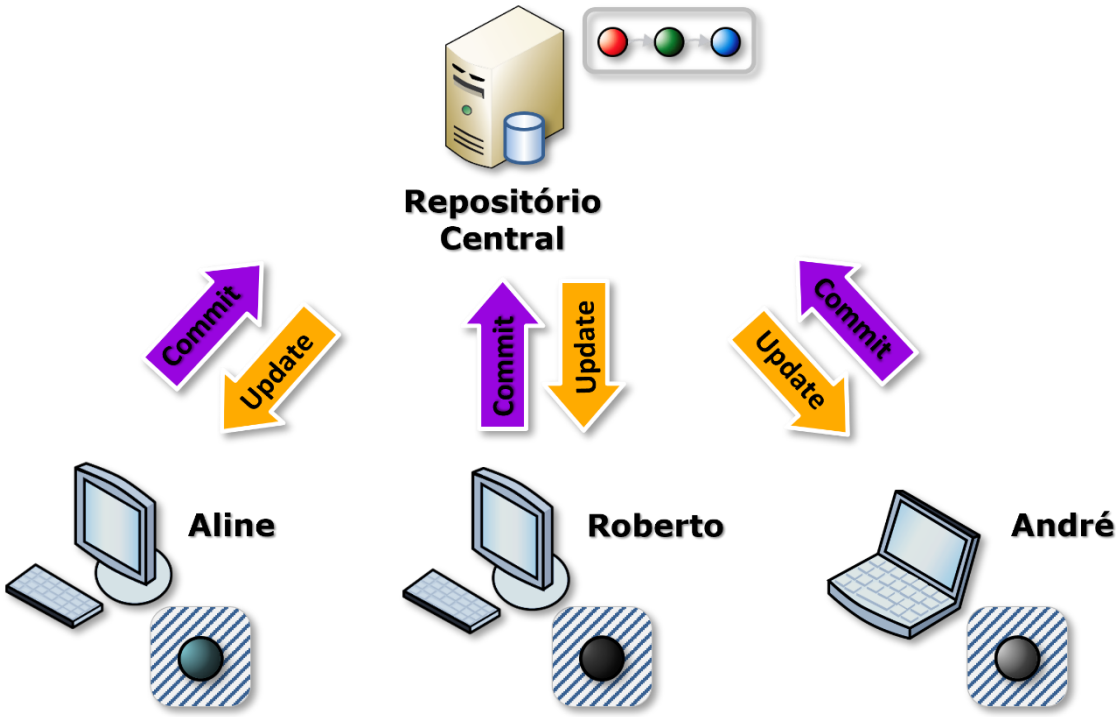
pacote contendo uma ou mais modificações feitas na área de trabalho (origem) ao repositório (destino). O update faz o caminho inverso, ou seja, envia as modificações contidas no repositório (origem) para a área de trabalho (destino). Dessa forma, é possível manter o histórico de todas as alterações realizadas, assim como gerenciar quem alterou e quando foi alterado. Cada *commit/update* mantém o histórico do usuário que acessou o repositório e da data e horário do acesso, gerando uma nova revisão no repositório, ou uma nova versão dos arquivos manipulados.

Existem dois tipos de controle de versão, o centralizado e o distribuído. Ambos possuem repositórios e áreas de trabalho, mas são diferentes na organização de cada uma dessas partes. Vamos entender como cada tipo de controle de versão funciona.

3.2 Controle de versão centralizado

O controle de versão **centralizado** segue o que chamamos de topologia em estrela, ou seja, é organizado apenas em um **único repositório central** e com **várias cópias de trabalho**, uma para cada desenvolvedor, em suas áreas de trabalho ou máquinas próprias. A comunicação entre uma área de trabalho e outra passa obrigatoriamente pelo repositório central, por meio dos comandos de *commit* e *update*.

Figura 1 – Exemplo de controle de versão centralizado



Fonte: Dias, 2016a.

Nesse exemplo é possível perceber a dinâmica de trabalho da equipe utilizando o controle de versão centralizado. Todos os desenvolvedores, a Aline, o Roberto e o André trabalham localmente, em suas áreas de trabalho. E, de maneira disciplinada, a constante alimenta o repositório central, tanto para buscar arquivos para serem alterados (*commit*) quanto para salvar arquivos que foram alterados ou criados (*update*). Dessa forma, todo o código do projeto fica centralizado, e o histórico de versões é mantido de forma automatizada pela ferramenta utilizada pelo projeto para esse fim.

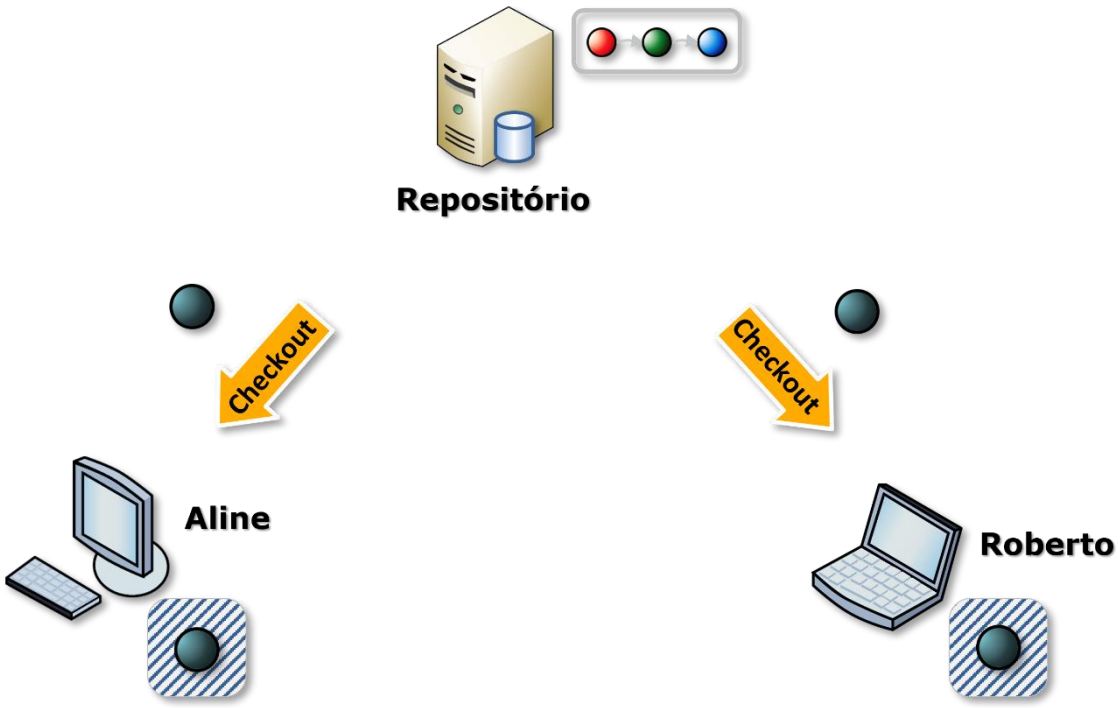
Cada versão recebe uma identificação única, que no controle de versão **centralizado** será um número inteiro sequencial: 1, 2, 3, e assim por diante. Como só existe um repositório, a numeração de revisão é a mesma para todos os desenvolvedores.

Já discutimos que uma das responsabilidades do controle de versão é permitir o trabalho paralelo e concorrente de vários desenvolvedores da equipe sobre os mesmos arquivos, evitando que um sobrescreva o código de outro, minimizando o reaparecimento de defeitos, perda de funcionalidades, e, por consequência, retrabalho. O que permite isso é, em parte, a área de trabalho, que dá a impressão de que o desenvolvedor é o único dono do projeto, dando autonomia para trabalhar sem se preocupar com o restante do software. Mas,

além da área de trabalho, ainda é necessária uma forma de sincronizar os esforços de todos os membros da equipe. Essa forma é a sincronização feita combinando-se versões concorrentes em uma única versão resultante, utilizando o comando *merge* (mesclagem). O trabalho de merge é realizado pela ferramenta de controle de versão, mas deve ser comandado por um desenvolvedor experiente, para manter a ordem certa dos arquivos que precisam ser mesclados. De forma bem planejada, o merge ocorre sem conflitos, gerando um código único, que consegue unir o trabalho de todos os desenvolvedores.

Vamos exemplificar como ocorre a sincronização das alterações quando é utilizado o controle de versão centralizado.

Figura 2 – Exemplo de sincronização das alterações quando é utilizado o controle de versão centralizado



Fonte: Dias, 2016a.

Vamos então entender o funcionamento do merge, por meio de um exemplo hipotético.

Vamos imaginar que Aline e Roberto comecem a trabalhar ao mesmo tempo, no mesmo arquivo. Cada um utiliza o comando checkout para criar uma cópia de trabalho. Dessa forma, duas cópias de trabalho são criadas a partir do comando checkout, cada uma na área de trabalho de cada desenvolvedor. As duas cópias iniciam no mesmo estado e com o mesmo conteúdo.

De forma independente, cada uma em sua área de trabalho, os dois desenvolvedores executam modificações nas suas cópias dos arquivos, mas Aline finaliza sua alteração primeiro e publica a sua versão antes de Roberto, no repositório central.

Em seguida, Roberto finaliza seu trabalho e tenta publicar suas alterações, mas o controle de versão recusa a atualização justificando que as alterações foram baseadas em arquivos desatualizados. Veja que a ferramenta já atualizou a versão da Aline e se aceitar a versão do Roberto, vai sobrescrever o trabalho realizado por Aline, gerando perda de código. Para resolver isso e evitar perda de código, na atualização da cópia de trabalho do Roberto, o controle de versão mescla automaticamente as revisões. Ou seja, a ferramenta executa o merge juntando o código do Roberto com o código da Aline, que já estava no repositório.

Após conferir se a atualização e o processo de merge produziram o resultado desejado, Roberto envia as mudanças ao repositório para serem efetivadas. Enquanto isso, Aline já trabalha em outra tarefa, executando novas alterações. Veja que temos a facilidade da ferramenta com a atenção e disciplina dos desenvolvedores, de forma a garantir um trabalho limpo e efetivo, sem perda de código e aumentando a produtividade da equipe como um todo, que pode trabalhar de maneira ágil e segura.

3.3 Controle de versão distribuído

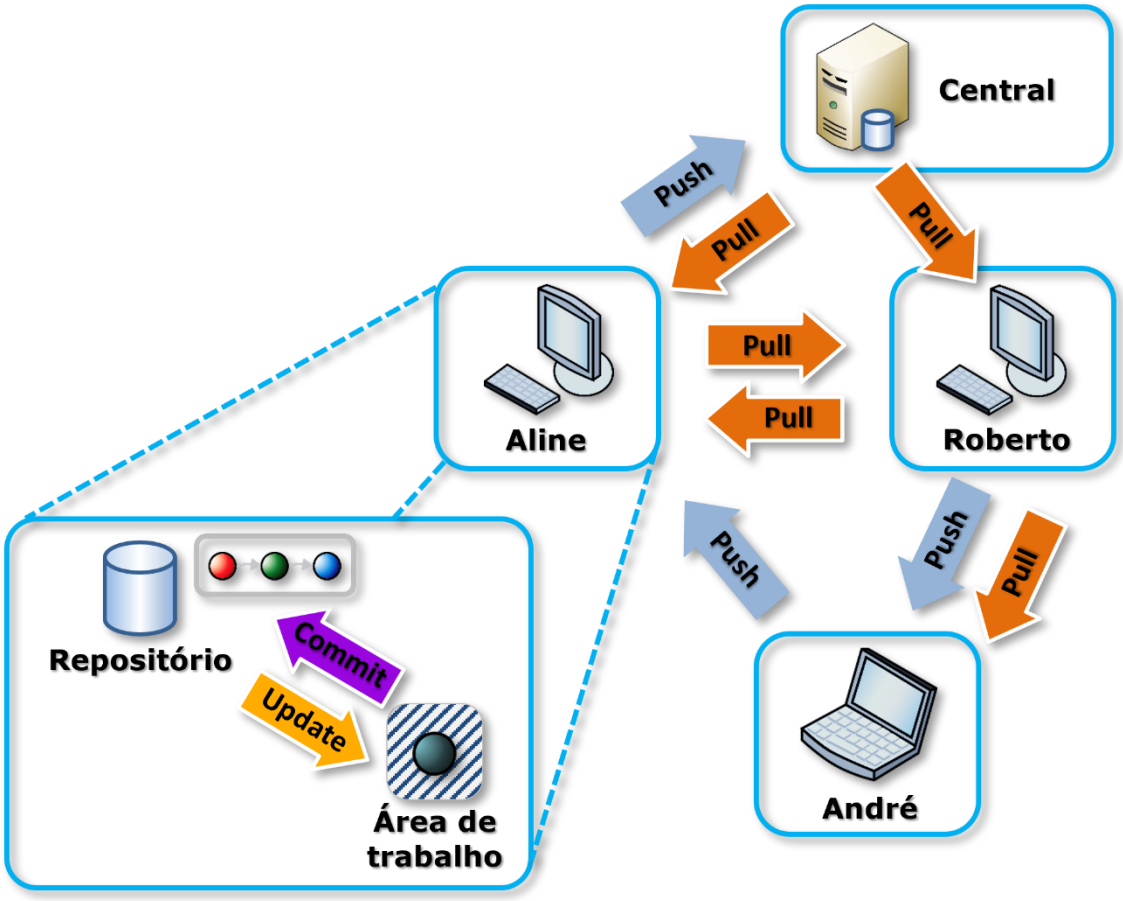
É um controle de versão um pouco mais complexo, porém, mais flexível. Ele é organizado por vários repositórios autônomos e independentes, um para cada desenvolvedor; além disso, cada repositório possui uma área de trabalho acoplada e as operações *commit* e *update* acontecem localmente entre os dois, repositório independente e área de trabalho acoplada.

Portanto, neste tipo de controle de versão teremos um conjunto de repositório independente e área de trabalho acoplada para cada um dos desenvolvedores da equipe. A comunicação entre os repositórios independentes e dos repositórios independentes com o repositório central ocorre por meio dos comandos de *pull* (puxar, ou ler) e *push* (empurrar, ou atualizar), em que:

- **Pull (puxar)** – atualiza o repositório local (destino) com todas as alterações feitas em outro repositório (origem);

- **Push (empurrar)** – envia as alterações do repositório local (origem) para um outro repositório (destino).

Figura 3 – Exemplo de controle de versão distribuído



Fonte: Dias, 2016a.

A organização do controle de versão distribuído envolve um repositório independente e uma área de trabalho para cada desenvolvedor, conforme mostrado para a Aline. E um repositório central do projeto, único, que centraliza o código de todos os desenvolvedores.

A comunicação entre o repositório independente e sua área de trabalho acoplada é realizada pelos comandos *commit* e *update*. Enquanto a comunicação entre os diversos repositórios independentes entre si e destes com o repositório central ocorrem por meio dos comandos *pull* e *push*.

No conceito do controle de versão distribuído, a sincronização entre os desenvolvedores acontece de repositório a repositório e não existe, em princípio, um repositório mais importante que o outro, embora o papel de um repositório

central possa ser usado para convencionar o fluxo de trabalho e fazer a centralização de todo o código do projeto de software.

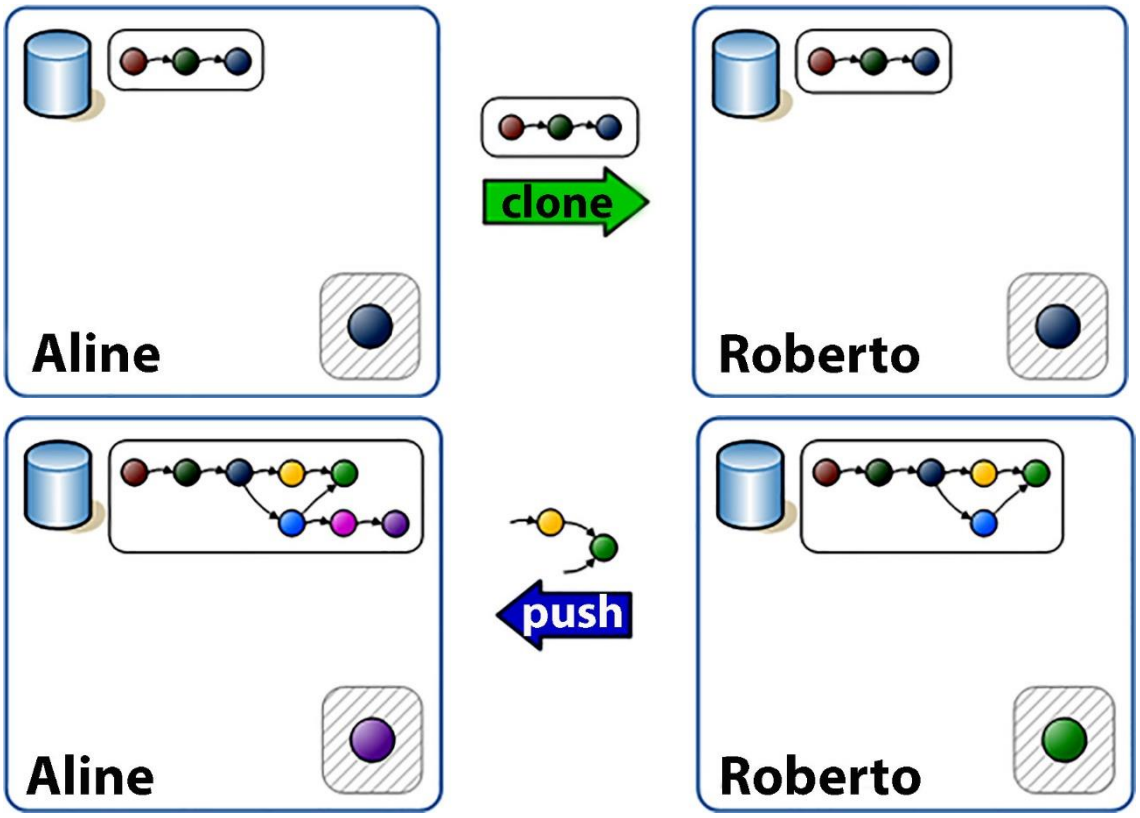
Como no sistema de controle de versão **distribuído** os repositórios são autônomos, para identificar a versão, não há como definir uma numeração sequencial compartilhada e única para todos. A solução é criar uma lógica e identificar cada revisão com uma numeração que nunca se repita em qualquer outro repositório, assim as versões serão únicas, mesmo integrando todos os repositórios.

A forma mais usada é por intermédio de uma função chamada *hash* SHA-1, que produz um número de 160 bits (40 dígitos na forma hexadecimal). Esse número é grande o suficiente e específico que torna extremamente improvável a repetição com um *hash* produzido por outro repositório.

Dessa forma, é possível ter versões únicas, de forma a manter a individualidade das alterações e ainda assim manter um trabalho colaborativo.

No controle de versão distribuídos, como estamos tratando de um mesmo software, é preciso tomar ainda mais cuidado com a sincronização do controle de versão. Vamos entender como funciona o sincronismo no controle de versão distribuído, lembrando dos comandos que são usados nesse tipo de controle de versão.

Figura 4 – Exemplo de sincronização das alterações quando é utilizado o controle de versão distribuído



Fonte: Dias, 2016a.

Vamos detalhar o funcionamento dessa modalidade de sincronismo por meio de um exemplo hipotético, como fizemos para o sincronismo utilizando a versão centralizada.

Vamos imaginar que Aline e Roberto precisem trabalhar no mesmo arquivo. Para iniciar o trabalho, é definido um repositório que será o principal da dupla, nesse exemplo, foi definido que será o repositório da Aline. Dessa forma, Roberto clona o repositório de Aline, para que ambos partam do mesmo ponto e do mesmo conteúdo de arquivos.

Aline e Roberto trabalham na área de trabalho acoplada e publicam suas alterações nos seus respectivos repositórios independentes, sem interferir no repositório um do outro, utilizando os comandos *commit* e *update*.

Aline finaliza seu trabalho antes, e Roberto sincroniza seu repositório com as versões publicadas por Aline, utilizando os comandos *pull* e *push*. Sua área de trabalho não é afetada pela sincronização realizada, pois elas foram feitas diretamente no seu repositório independente. Perceba que nessa situação, a

comunicação entre os profissionais da equipe de desenvolvimento é fundamental para o bom andamento do projeto, portanto, a disciplina é muito mais necessária do que no caso do controle de versão centralizado.

Como Aline finalizou suas alterações primeiro, o merge entre as revisões de Aline e Roberto deve ser feita explicitamente na área de trabalho de Roberto, pois ele é quem precisa unir suas alterações com as alterações já realizadas pela Aline. Enquanto isso, Aline já gera outra revisão no seu repositório, de outros arquivos ou, até mesmo, dos mesmos já em utilização por Roberto. Após conferir se a mesclagem produziu o resultado desejado, Roberto envia as mudanças ao seu repositório independente. Paralelamente, Aline publica mais uma vez no seu repositório independente, fazendo um trabalho colaborativo, porém de maneira autônoma.

Por fim, Roberto envia suas versões ao repositório de Aline, por meio do comando *push* e *pull*. Aline, então, combina as versões de Roberto com o histórico de revisões já existente, garantindo a integração de tudo antes de atualizar o repositório central, também utilizando os comandos *push* e *pull*.

Veja que o processo é um pouco mais complexo que o processo de controle de versão centralizado, e exige um controle maior dos desenvolvedores em relação à comunicação ativa durante o desenvolvimento do software.

É preciso definir qual o melhor modelo a ser adotado, o de controle de versão centralizado ou o de controle de versão distribuído, dependendo das características do projeto e do software, além da estratégia de organização da equipe do projeto. A complexidade do software também influencia na decisão de qual tipo de controle de versão utilizar.

Ultimamente, equipes muito grandes estão optando por utilizar o controle de versão distribuído. Mas não significa que o controle de versão centralizado não funcione nesses casos. Como vimos, o controle de versão centralizado se baseia na arquitetura cliente-servidor, com um repositório central (servidor) e áreas de trabalho para os desenvolvedores (clientes).

Para equipes de desenvolvimento com acesso ao repositório pela rede local, essa arquitetura funciona bem, a velocidade da rede não é problemática, o tempo de resposta do processamento é aceitável e todos os desenvolvedores da equipe têm permissão de leitura e escrita no repositório.

Mas o controle de versão distribuído foi pensado para resolver problemas de projetos que possuem o seguinte cenário:

- **equipe de desenvolvedores muito grande**, significando que mais processamento vai ser exigido do servidor central, piorando o tempo de resposta, prejudicando assim a produtividade;
- **equipe espalhada em diferentes filiais da empresa**. Acesso remoto ao repositório com limitações de conexão e de permissão de escrita, dificultando o trabalho em equipe.

O tipo de controle de versão a ser utilizado define qual é a melhor ferramenta de gerência de configuração para o projeto. Atualmente, existem várias ferramentas no mercado, como: *GitHub, SubVersion, BitBucket, Redmine, Mercurial*, entre outras.

TEMA 4 – CONFIGURAÇÃO DE AMBIENTES, *BASELINES* E *RELEASE*

Discutimos que o controle de versão pode ser dividido em centralizado ou distribuído, e isso nos leva a pensar que devemos organizar de maneira adequada o ambiente de desenvolvimento, para que a equipe consiga trabalhar de forma independente, porém, facilitando a junção do código, quando for necessário.

O controle de versão envolve recursos que são constantemente utilizados nos projetos de software, para organizar o desenvolvimento, minimizar riscos para o negócio e melhorar a produtividade da equipe de desenvolvimento. Pensando nesses pontos positivos, em determinados momentos do ciclo de vida de desenvolvimento e manutenção do software, é preciso agrupar e verificar os itens de configuração, para constituir configurações do software voltadas para propósitos específicos. É, nesse momento, que se cria conjuntos de componentes relacionados, criando o que chamamos de *baselines* ou *releases*.

Em outras palavras, *baseline* também pode ser entendida como sendo uma organização lógica dos artefatos que compõem o produto de software e que tem como objetivo caracterizar um conjunto estável dos componentes relacionados com um determinado momento do projeto, ou, ainda, pode ser definida como uma versão formalmente aprovada de um conjunto de itens de configuração.

A diferença entre *baselines* e *releases* é sutil, mas precisa ser entendida para que os conceitos sejam usados corretamente. Dessa forma, as *baselines* representam conjuntos de itens de configuração formalmente aprovados que

servem de base para as etapas seguintes do ciclo de vida do desenvolvimento do software. Por outro lado, quando uma entrega formal é feita ao cliente, no final de uma iteração, por exemplo, denominamos essa entrega de *release*. Ou seja, a *baseline* é um conceito interno, utilizado para evoluir um conjunto de componentes ao longo das fases do desenvolvimento. Já a *release* é um conjunto de componentes integrados, testados e pronto para ser liberado para o cliente.

A *release* precisa ser bem gerenciada, para garantir que a entrega para o cliente está de acordo com o que foi testado no ambiente de testes. Ou seja, é preciso garantir a integridade da *release*, com as alterações necessárias para atender às necessidades do cliente de forma completa e correta.

Tanto as *baselines* quanto as *releases* são identificadas no repositório da ferramenta de controle de versão utilizada pelo projeto, por meio de etiquetas ou *tags*, ou ainda *labels*, que são criadas para identificar os conjuntos de componentes de um software.

Além de compreender o que é *baseline* e *release*, é precioso entender os conceitos relacionados à ramificação ou *branch.* Mas vamos entender mais detalhadamente o que realmente é ramificação no contexto da gerência de configuração.

4.1 O que é *branch* ou ramificação?

No contexto da gerência de configuração, um ramo é uma linha diferente na evolução do software, que forma uma variação isolada e controlada do projeto. Ou seja, um ramo é uma outra linha de desenvolvimento do código, que quando pronta, poderá ser integrada à linha principal do código do software. A utilização de ramificação em projetos de software ajuda a resolver vários problemas, como:

- permitem a execução simultânea de atividades diferentes, como codificação e testes, por exemplo;
- organizam o esforço de implementação por finalidade, equipe, risco, restrição, facilitando o trabalho em equipe;
- isolam implementações arriscadas das que são certas e rápidas, melhorando a qualidade e minimizando o risco no desenvolvimento do software.

A ramificação produz uma estrutura que consegue absorver todas as mudanças no projeto de maneira ordenada, dando tranquilidade para a equipe de desenvolvimento trabalhar. A ramificação pode ser utilizada em mudanças emergenciais, que precisam ser atendidas com urgências e postas em produção, até alterações incertas, que podem se mostrar viáveis ou não com o tempo.

Existe mais de um tipo de ramificação, que se diferencia pela finalidade de utilização, ou seja, são ramificações utilizadas para fins diferentes, como:

- **ramo principal** – é o ramo mais importante de um projeto, utilizado para armazenar e controlar o código principal e integrado do software. Ele concentra todo desenvolvimento, as correções, as funcionalidades e as melhorias nessas funcionalidades. O ramo principal serve para centralizar o código que fará parte da versão que será implantada em produção. Para que seja mantido sempre estável e confiável, apenas as implementações simples, rápidas e confirmadas podem ser feitas diretamente no ramo principal. Todas as demais devem ser feitas em outros ramos e apenas promovida para o ramo principal quando estiverem testadas e aprovadas. Dependendo da ferramenta de controle de versão, o nome dado ao ramo principal varia. Por exemplo, no *Subversion*, o ramo principal é chamado de *trunk*. No *Git*, o ramo principal é chamado de master. E no *Mercurial*, o ramo principal é chamado de default. Todo projeto deve possuir um ramo principal, que centraliza o código principal e pronto do projeto;
- **ramo dedicado** (também chamado de *feature branch* ou *topic branch*) – é o ramo usado para isolar as implementações complexas, demoradas ou incertas que não podem ser feitas diretamente no ramo principal, e que geram riscos para a estabilidade do software. Os casos que normalmente requerem um ramo dedicado são a implementação de um módulo/subsistema/componente ou alguma experimentação tecnológica/arquitetônica, que precisam ser completamente testadas antes de serem integradas ao software principal;
- **ramo de manutenção** (também chamado de *release branch*) – é o ramo que corresponde a uma versão lançada em produção que precisa ser corrigida, de forma pontual, sem impactar o conjunto das demais funcionalidades em desenvolvimento. As correções são primeiro feitas no

ramo de manutenção e depois repassadas aos demais ramos do projeto por meio do merge, que foi estudado no tópico anterior.

Portanto, os diferentes tipos de ramos inserem mais controle e estabilidade na construção e manutenção do software. É preciso também ter disciplina, para utilizar os ramos certos, garantindo assim a qualidade do software e minimizando o retrabalho.

TEMA 5 – GERÊNCIA DE MUDANÇA DE ARQUIVOS

Assim como estudamos nos tópicos anteriores, a importância da gerência de configuração e da gerência de mudança para os componentes de um software, é preciso ressaltar a importância do gerenciamento dos demais arquivos criados ao longo do desenvolvimento de um software, e mesmo depois do seu desenvolvimento, na fase de manutenção.

Quando falamos em arquivos, estamos falando de todos os artefatos que são criados para um projeto de software. Sejam eles artefatos gerenciais ou técnicos, criados como apoio para o entendimento do funcionamento e das necessidades de um software.

Existem algumas boas práticas que devem ser adotadas para garantir um gerenciamento adequado das mudanças de todos os arquivos de um software. Vamos analisar algumas delas:

- **identificar o requisitante da mudança** – saber quem requisitou a mudança é fundamental, porque quanto mais informado sobre o processo de mudança estiver a equipe, mais assertivas e com menor risco se tornam as alterações na operação. Identificar quem requisitou a mudança evidencia o alto nível de segurança de um processo, controlando de qual usuário partiu a alteração, trazendo rastreabilidade e segurança, além de ajudar a priorizar as mudanças que agregam mais valor para o negócio. É importante, ainda, classificar os níveis de mudança e configurar quem pode solicitá-las, além de garantir o acesso às informações da mudança apenas para os usuários envolvidos nos processos e autorizados para essa visualização;
- **identificar o motivo para realizar a mudança** – estabelecer as razões para a mudança é essencial, assim como informá-las às áreas impactadas. Para especificar e notificar qual é a razão da mudança, é

necessário conversar com todos os envolvidos com a mudança solicitada, de forma a garantir o entendimento completo dos requisitos;

- **identificar qual será o retorno esperado após a implantação da mudança** – é importante saber quais retornos a organização espera após a execução da mudança, sejam eles financeiros, de minimização de riscos ou para eliminação de desperdícios, por exemplo. É preciso medir também o custo para a realização da mudança, e comparar o custo X o benefício. Ou seja, o retorno esperado com a mudança deve ser maior que o custo com a sua realização;
- **identificar os riscos envolvidos na mudança** – durante todo o desenvolvimento do software é fundamental mapear todos os riscos envolvidos com o contexto do projeto. Da mesma forma e com o mesmo objetivo, durante a mudança, os riscos devem ser identificados e analisados. O gerenciamento adequado dos riscos envolve tecnologia e estratégia, o que permite menos instabilidade e uma gestão mais segura das mudanças. O principal é evitar ou minimizar o impacto dos riscos nos negócios, garantindo estabilidade de funcionamento adequado para o software;
- **identificar as necessidades de recursos para o desenvolvimento e a entrega da mudança** – é de vital importância para o desenvolvimento das mudanças identificar todos os recursos necessários para a sua realização, sejam recursos humanos, sejam recursos físicos. É muito improdutivo, durante a realização de uma mudança, identificar que a organização não tem determinados recursos, gerando atrasos no planejamento e ociosidade da equipe de desenvolvimento. Se os recursos necessários para a construção da mudança dependerem da contratação de um fornecedor externo, é preciso se atentar para os prazos de entrega, para garantir um planejamento assertivo;
- **identificar quem é o responsável pela construção, pelos testes e pela implementação da mudança** – o planejamento adequado de uma mudança requer identificar quais os perfis necessários para sua completa realização e a quantidade de profissionais, dependendo da criticidade e da complexidade da mudança. Toda mudança tem responsáveis dedicados pela execução das atividades, como construção, testes e

implementação. Automatizar um fluxo, definindo as pessoas envolvidas no processo da mudança é o caminho para garantir um planejamento e uma execução adequados. Uma mudança pode ser controlada por meio de um cronograma próprio, ou ser inserida no cronograma do projeto. Independentemente da forma, é importante garantir o acompanhamento constante, até a conclusão da mudança e entrega para o cliente;

- **identificar o impacto da mudança nos demais softwares e áreas da empresa** – é preciso utilizar a rastreabilidade para identificar todo o impacto causado pela mudança solicitada. O planejamento completo do custo e do prazo necessários para concluir uma mudança depende da complexidade e do impacto dela causado no software como um todo e até mesmo nos demais softwares utilizados pela empresa.

Dessa forma, fica clara a importância do gerenciamento de mudança de todos os artefatos de um projeto, para garantir que a documentação esteja atualizada e de acordo com o previsto e proposto pela mudança solicitada.

Não é fácil manter um software atualizado, mas é fundamental para garantir uma manutenção sustentável e segura para a evolução natural necessária a qualquer software.

O processo de gerência de configuração e a ferramenta de controle de versão e de mudança devem cobrir todos os artefatos gerados ao longo do desenvolvimento do software, garantindo a evolução controlada de todo o software, lembrando que a documentação gerada também faz parte do que será entregue para o cliente, portanto, faz parte do projeto de software.

FINALIZANDO

Nesta etapa, discutimos sobre a importância de automatizar o processo de controle de versão, para aumentar a produtividade da equipe de desenvolvimento e melhorar a qualidade do software como um todo.

Discutimos também sobre conceitos importantes como artefatos de software, integração contínua, *baseline* e *release*. Além de estudarmos o funcionamento de uma ferramenta como o GitHub, entendendo todas as facilidades e controles possíveis para o processo de gerência de configuração.

Compreendemos os conceitos relacionados ao versionamento dos artefatos de projetos, entendendo sobre check-in e checkout no contexto de projetos de software e gerência de configuração.

E estudamos sobre a importância de configurar de maneira adequada o ambiente de desenvolvimento de software, para facilitar o controle da construção e das alterações necessárias aos componentes de um software.

A cada etapa, percebemos a importância e a complexidade do processo de gerência de configuração, permeia todo o processo de desenvolvimento de software, pois, todos os artefatos críticos precisam estar sob gerência de configuração e controle de versão.

Vamos continuar evoluindo e aprofundando os conceitos que envolvem a Gerência de Configuração.

REFERÊNCIAS

ANDRADE JUNIOR, J. R. **Gerência de Configuração**. 1. ed. Pearson, 2014.

AWS. **O que significa integração contínua?** Disponível em: <https://aws.amazon.com/pt/devops/continuous-integration/#:~:text=Explica%C3%A7%C3%A3o%20da%20integra%C3%A7%C3%A3o%20cont%C3%ADnua,cria%C3%A7%C3%B5es%20e%20testes%20s%C3%A3o%20executados>. Acesso em: 17 jun. 2022.

DIAS, A. F. Conceitos básicos de controle de versão de software — centralizado e distribuído. **Blog**. 2016a. Disponível em: <https://blog.pronus.io/posts/controle-de-versao/conceitos-basicos-de-controle-de-versao-de-software-centralizado-e-distribuido/>. Acesso em: 21 abr. 2022.

_____. O que é Gerência de Configuração de Software? **Blog**. 2016b. Disponível em: <https://blog.pronus.io/posts/controle-de-versao/o-que-eh-gerencia-de-configuracao-de-software/>. Acesso em: 17 jun. 2022.

_____. Tipos de ramos do controle de versão. **Blog**. 2016c. Disponível em: <https://blog.pronus.io/posts/controle-de-versao/tipos-de-ramos-do-controle-de-versao/>. Acesso em: 17 jun. 2022.

MORAIS, I. S. de. **Engenharia de Software**. Porto Alegre: Sagah, 2017.

MUNIZ, A.; SANTOS, R. **Jornada Devops**: unindo cultura ágil, lean e tecnologia para entrega de software com qualidade. 1. ed. Brasport, 2019.

PAULA FILHO, W. de P. **Engenharia de softwares**: produtos. 4. ed. Rio de Janeiro: LTC, 2019.

PFLEEGER, S. L. **Engenharia de software**: teoria e prática. 2. ed. São Paulo: Prentice Hall, 2004.

SBROCCO, J. H. T. de C. **Metodologias ágeis**: Engenharia de Software sob medida. 1. ed. São Paulo: Érica, 2012.

SOARES, G. M. Gerência de Mudanças (GMUD) – controlando as intervenções no ambiente de produção. **Web artigos**. 2008. Disponível em: <https://www.webartigos.com/artigos/gerencia-de-mudancas-gmud-controlando-as-intervencoes-no-ambiente-de-producao/12779>. Acesso em: 17 jun. 2022.

SOFTEX. Associação para Promoção da Excelência do Software Brasileiro – SOFTEX. **MPS.BR** – Guia de Implementação – Parte 2: Fundamentação para Implementação do Nível F do MR-MPS:2011. 2011. Disponível em: <www.softex.br>. Acesso em: 17 jun. 2022.

SOFTEX. Associação para Promoção da Excelência do Software Brasileiro – SOFTEX. **MPS.BR** – Guia Geral MPS de Software: 2016. 2016. Disponível em: <www.softex.br>. Acesso em: 17 jun. 2022.

SOMMERVILLE, I. **Engenharia de software**. 10. ed. São Paulo: Pearson, 2019.

VETORAZZO, A. de S. **Engenharia de Software**. Porto Alegre: Sagah, 2018.