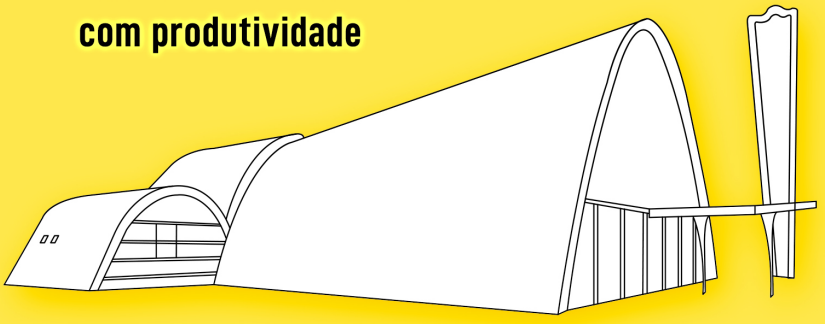
ENGENHARIA *DE* SOFTWARE MODERNA

Princípios e práticas para desenvolvimento de software com produtividade



MARCO TULIO VALENTE

Engenharia de Software Moderna

Princípios e Práticas para Desenvolvimento de Software com Produtividade

Marco Tulio Valente

Versão 2020.1.0 - LeanPub

Para Cynthia, Daniel e Mariana.

Conteúdo

Capítulo 1

DevOps

> *Imagine a world where product owners, development, QA, IT Operations, and Infosec work together, not only to help each other, but also to ensure that the overall organization succeeds.* – G. Kim, J. Humble, P. Debois, J. Willes

Este capítulo inicia discutindo o conceito de DevOps e seus benefícios (Seção 10.1). Apesar de ser um termo novo, existe uma tendência em ver DevOps como um movimento que visa introduzir práticas ágeis "na última milha" de um projeto de software, isto é, quando o sistema vai entrar em produção. Além de discutir o conceito, tratamos de três práticas importantes quando adota-se DevOps. São elas: Controle de Versões (Seção 10.2), Integração Contínua (Seção 10.3) e Deployment Contínuo (Seção 10.4).

1.1 Introdução

Até agora, neste livro, estudamos um conjunto de práticas para desenvolvimento de software com qualidade e agilidade. Por meio de métodos ágeis — como Scrum, XP ou Kanban —, vimos que o cliente deve participar desde o primeiro dia da construção de um sistema. Também estudamos práticas importantes para produção de software com qualidade, como testes de unidade e refactoring. Estudamos ainda princípios e padrões de projeto e também padrões arquiteturais.

Logo, após aplicar o que vimos, o sistema — ou um incremento dele, resultante de um sprint — está pronto para entrar em produção. Essa tarefa é conhecida pelos nomes de **implantação (deploy)**, **liberação (release)** ou **entrega (delivery)**

do sistema. Independentemente do nome, ela não é tão simples e rápida como pode parecer.

Historicamente, em organizações tradicionais, a área de Tecnologia da Informação era dividida em dois departamentos:

- Departamento de Sistemas (ou Desenvolvimento), formado por desenvolvedores, programadores, analistas, arquitetos, etc.
- Departamento de Suporte (ou Operações), no qual ficavam alocados os administradores de rede, administradores de bancos de dados, técnicos de suporte, técnicos de infraestrutura, etc.

Hoje em dia, é fácil imaginar os problemas causados por essa divisão. Na maioria das vezes, a área de suporte tomava conhecimento de um sistema na véspera da sua implantação. Consequentemente, a implantação poderia atrasar por meses, devido a uma variedade de problemas que não tinham sido identificados. Dentre eles, podemos citar a falta de hardware para executar o novo sistema ou a nova funcionalidade, problemas de desempenho, incompatibilidades com o banco de dados de produção, vulnerabilidades de segurança, etc. No limite, esses problemas poderiam resultar no cancelamento da implantação e no abandono do sistema.

Resumindo, nesse modelo tradicional, existia um stakeholder importante — os administradores de sistemas ou *sysadmins* — que tomava conhecimento das características e requisitos não-funcionais de um novo software na véspera da implantação. Esse problema era agravado pelo fato de os sistemas serem monolitos, cuja implantação gerava todo tipo de preocupação, como mencionado no final do parágrafo anterior.

Então, para facilitar a implantação e entrega de sistemas, foi proposto o conceito de **DevOps**. Por ser um termo recente, ele ainda não possui uma definição consolidada. Mas seus proponentes gostam de descrever DevOps como um movimento que visa unificar as culturas de desenvolvimento (Dev) e de operação (Ops), visando permitir a implantação mais rápida e ágil de um sistema. Esse objetivo está refletido na frase que abre esse capítulo, de autoria de Gene Kim, Jez Humble, Patrick Debois e John Willes, todos eles membros de um grupo de desenvolvedores que ajudou a difundir os princípios de DevOps. Segundo eles, DevOps implica em uma disrupção na cultura tradicional de implantação de sistemas (link):

> "Em vez de iniciar as implantações à meia-noite de sexta-feira e passar o fim de semana trabalhando para concluí-las, as implantações ocorrem em qualquer dia útil, quando todos estão na empresa e sem que os clientes percebam — exceto quando encontram novas funcionalidades e correções de bugs."

No entanto, DevOps não advoga a criação de um profissional novo, que fique responsável tanto pelo desenvolvimento como pela implantação de sistemas. Em vez disso, defende-se uma aproximação entre o pessoal de desenvolvimento e o pessoal de operações e vice-versa, visando fazer com que a implantação de sistemas seja mais ágil e menos traumática. Tentando explicar com outras palavras, a ideia é evitar dois silos independentes: desenvolvedores e operadores, com pouca ou nenhuma iteração eles, como ilustrado na figura a seguir.

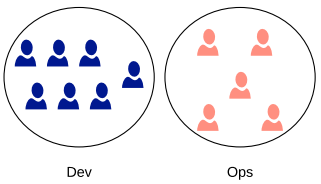


Figura 1.1: Organização que **não** é baseada em DevOps, pois existe pouca comunicação entre Dev e Ops.

Em vez disso, defende-se que esses profissionais atuem em conjunto desde os primeiros sprints de um projeto, como ilustrado na figura a seguir. Para o cliente, o benefício deve ser a entrada em produção mais cedo do sistema que ele contratou.

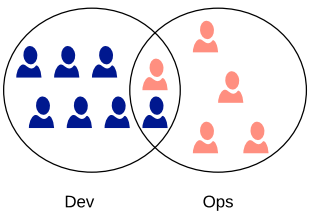


Figura 1.2: Organização baseada em DevOps. Frequentemente, alguns Dev e alguns Ops sentam juntos para discutir questões sobre a entrega do sistema.

Quando migra-se para uma cultura de DevOps, os times ágeis podem incluir um profissional de operações, que participará dos trabalhos em tempo parcial ou mesmo em tempo integral. Sempre em função da demanda, esse profissional pode também participar de mais de um time. A ideia é que ele antecipe problemas de desempenho, segurança, incompatibilidades com outros sistemas, etc. Ele pode também, enquanto o código está sendo implementado, começar a trabalhar nos scripts de instalação, administração e monitoramento do sistema em produção.

De forma não menos importante, DevOps advoga a automatização de todos os passos necessários para colocar um sistema em produção e monitorar o seu correto funcionamento. Isso implica na adoção de práticas que já vimos neste livro, notadamente testes automatizados. Mas também implica no emprego de novas práticas e ferramentas, tais como Integração Contínua (*Continuous Integration*) e Deployment Contínuo (*Continuous Deployment*), que iremos estudar no presente capítulo.

> **Mundo Real**: O termo DevOps começou a ser usado no final dos anos 2000 por profissionais frustrados com os atritos constantes entre as equipes de desenvolvimento e de operações. Então, eles convenceram-se de que uma solução seria a adoção de princípios ágeis não apenas na fase de desenvolvimento, mas também na fase de implantação de sistemas. Para citar uma data precisa, em Novembro de 2009 foi realizada, na Bélgica, a primeira conferência da indústria sobre o tema, chamada DevOpsDay. Considera-se que foi nesta conferência, organizada por Patrick Dubois, que a palavra DevOps foi cunhada (link).

Para finalizar, vamos discutir um conjunto de princípios para entrega de software, enunciados por Jez Humble e David Harley (link). Apesar de propostos antes da ideia de DevOps ganhar tração, eles estão completamente alinhados com essa ideia. Alguns desses princípios são os seguintes:

- **Crie um processo repetível e confiável para entrega de software**. Esse princípio é o mais importante deles. A ideia é que a entrega de software não pode ser um evento traumático, com passos manuais e sujeitos a surpresas. Em vez disso, colocar um software em produção deve ser tão simples como apertar um botão.
- **Automatize tudo que for possível**. Na verdade, esse princípio é um pré-requisito do princípio anterior. Advoga-se que todos os passos para entrega de um software devem ser automáticos, incluindo seu *build*, a execução dos

testes, a configuração e ativação dos servidores e da rede, a carga do banco de dados, etc. De novo, idealmente, queremos apertar um botão e, em seguida, ver o sistema em produção.

- **Mantenha tudo em um sistema de controle de versões**. "Tudo" no enunciado do princípio refere-se não apenas a todo o código fonte, mas também arquivos e scripts de administração do sistema, documentação, páginas Web, arquivos de dados, etc. Consequentemente, deve ser simples restaurar e voltar o sistema para um estado anterior. Neste capítulo, iniciaremos estudando alguns conceitos básicos de **Controle de Versões**, na Seção 10.2. Além disso, no Apêndice A apresentamos o uso dos principais comandos do sistema Git, que é o sistema de controle de versões mais usado atualmente.
- **Se um passo causa dor, execute-o com mais frequência e o quanto antes**. Esse princípio não tem uma inspiração masoquista. Em vez disso, a ideia é antecipar os problemas, antes que eles se acumulem e as soluções fiquem complicadas. O exemplo clássico é o de **Integração Contínua**. Se um desenvolvedor passa muito tempo trabalhando de forma isolada, ele e o seu time podem depois ter uma grande dor de cabeça para integrar o código. Logo, como integração pode causar dor, a recomendação consiste em integrar código novo com mais frequência e o quanto antes, se possível, diariamente. Iremos estudar mais sobre integração contínua na Seção 10.3.
- **"Concluído" significa pronto para entrega**. Com frequência, desenvolvedores dizem que uma nova história está pronta (*done*). Porém, ao serem questionados se ela pode entrar em produção, começam a surgir "pequenas" pendências, tais como: a implementação ainda não foi testada com dados reais, ela ainda não foi documentada, ela ainda não foi integrada com o sistema X, etc. Esse princípio defende então que "concluído", em projetos de software, deve ter uma semântica clara, isto é: 100% pronto para entrar em produção.
- **Todos são responsáveis pela entrega do software**. Esse último princípio alinha-se perfeitamente com a cultura de DevOps, que discutimos no início desta Introdução. Ou seja, não admite-se mais que os times de desenvolvimento e de operações trabalham em silos independentes e troquem informações apenas na véspera de uma implantação.

1.2 Controle de Versões

Como mencionamos algumas vezes neste livro, software é desenvolvido em equipe. Por isso, precisamos de um servidor para armazenar o código fonte do sistema

que está sendo implementado por um grupo de desenvolvedores. A existência desse servidor é fundamental para que esses desenvolvedores possam colaborar e para que os operadores saibam precisamente qual versão do sistema deve ser colocada em produção. Além disso, sempre é útil manter o histórico das versões mais importantes de cada arquivo. Isso permite, se necessário, realizar uma espécie de "undo" no tempo, isto é, recuperar o código de um arquivo como ele estava há anos atrás, por exemplo.

Um **Sistema de Controle de Versões** (VCS, na sigla em inglês) oferece os dois serviços mencionados no parágrafo anterior. Primeiro, ele oferece um **repositório** para armazenar a versão mais recente do código fonte de um sistema, bem como de arquivos relacionados, como arquivos de documentação, configuração, páginas Web, manuais, etc. Em segundo lugar, ele permite que se recupere versões mais antigas de qualquer arquivo, caso isso seja necessário. Como enunciamos na Introdução, modernamente é inconcebível desenvolver qualquer sistema, mesmo que simples, sem um VCS.

Os primeiros sistemas de controle de versões surgiram no início da década de 70, como o sistema SCCS, desenvolvido para o sistema operacional Unix. Em seguida, surgiram outros sistemas, como o CVS, em meados da década de 80, e depois o sistema Subversion, também conhecido pela sigla svn, no início dos anos 2000. Todos são sistemas centralizados e baseados em uma arquitetura cliente/servidor (veja figura a seguir). Nessa arquitetura, existe um único servidor, que armazena o repositório e o sistema de controle de versões. Os clientes acessam esse servidor para obter a versão mais recente de um arquivo. Feito isso, eles podem modificar o arquivo, por exemplo, para corrigir um bug ou implementar uma nova funcionalidade. Por fim, eles atualizam o arquivo no servidor, realizando uma operação chamada **commit**, que torna o arquivo visível para outros desenvolvedores.

No início dos anos 2000, começaram a surgir **Sistemas de Controle de Versões Distribuídos** (DVCS). Dentre eles, podemos citar o sistema BitKeeper, cujo primeiro release é de 2000, e os sistemas Mercurial e git, ambos lançados em 2005. Em vez de uma arquitetura cliente/servidor, um DVCS adota uma arquitetura peer-to-peer. Na prática, isso significa que cada desenvolvedor possui em sua máquina um servidor completo de controle de versões, que pode se comunicar com os servidores de outras máquinas, como ilustrado na próxima figura.

Em teoria, quando se usa um DVCS, os clientes (ou *peers*) são funcionalmente equivalentes. Porém, na prática, costuma existir uma máquina principal, que armazena a versão de referência do código fonte. Na nossa figura, chamamos esse

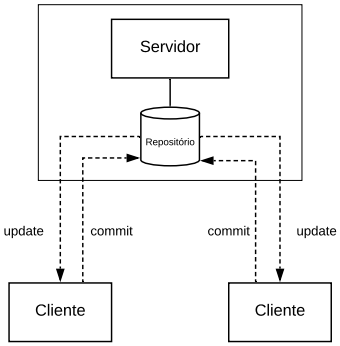


Figura 1.3: VCS Centralizado. Existe um único repositório, no nodo servidor.

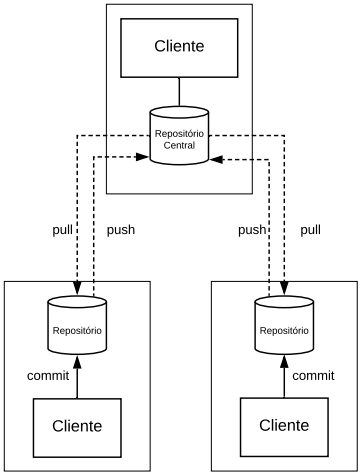


Figura 1.4: VCS Distribuído (DVCS). Cada cliente possui um servidor. Logo, a arquitetura é peer-to-peer.

repositório de **repositório central**. Cada desenvolvedor pode trabalhar de forma independente e até mesmo offline em sua máquina cliente, realizando commits no seu repositório. De tempos em tempos, ele deve sincronizar esse repositório com o central, por meio de duas operações: **pull** e **push**. Um pull atualiza o repositório local com novos commits disponíveis no repositório central. Por sua vez, um push faz a operação contrária, isto é, ele envia para o repositório central os commits mais recentes realizados pelo desenvolvedor em seu repositório local.

Quando comparado com VCS centralizados, um DVCS tem as seguintes vantagens:

- Pode-se trabalhar e gerenciar versões de forma offline, sem estar conectado a uma rede, pois os commits são realizados primeiro no repositório local.
- Pode-se realizar commits com mais frequência, incluindo commits com implementações parciais, pois eles não vão chegar imediatamente até o repositório central.
- Commits são executados em menos tempo, isto é, eles são operações mais rápidas e leves. O motivo é que eles são realizados no repositório local de cada máquina.
- A sincronização não precisa ser sempre com o repositório central. Em vez disso, dois nodos podem também sincronizar os seus repositórios. Por exemplo, pode-se ter uma estrutura hierárquica dos repositórios. Nesses casos, os commits "nascem" nos repositórios que representam as folhas da hierarquia e vão subindo até chegar ao repositório central.

Git é um sistema de controle de versões distribuído cujo desenvolvimento foi liderado por Linus Torvalds, também responsável pela criação do sistema operacional Linux. Nos anos iniciais, o desenvolvimento do kernel do Linux usava um sistema de controle de versões comercial, chamado BitKeeper, que também possui uma arquitetura distribuída. No entanto, em 2005, a empresa proprietária do BitKeeper resolveu revogar as licenças gratuitas que eram usadas no desenvolvimento do Linux. Os desenvolvedores do sistema operacional, liderados por Torvalds, decidiram então iniciar a implementação de um DVCS próprio, ao qual deram o nome de Git. Assim como o Linux, o Git é um sistema de código aberto, que pode ser gratuitamente instalado em qualquer máquina. O Git é também um sistema de linha de comando. Porém, existem clientes com interfaces gráficas, desenvolvidos por terceiros, que permitem usar o sistema sem ter que digitar comandos.

GitHub é um serviço de hospedagem de código que usa o sistema Git para prover controle de versões. O GitHub oferece repositórios públicos e gratuitos, para projetos de código aberto, e repositórios fechados e pagos, para uso por empresas. Assim, em vez de manter internamente um DVCS, uma empresa desenvolvedora de software pode alugar esse serviço do GitHub. Uma comparação pode ser feita com serviços de mail. Em vez de instalar um servidor de mail em uma máquina própria, uma empresa pode contratar esse serviço de terceiros, como do Google, via GMail. Apesar de o GitHub ser o mais popular, existem serviços semelhantes providos por outras empresas, como GitLab e BitBucket.

No Apêndice A, apresentamos e ilustramos os principais comandos do sistema Git. São explicados também os conceitos de forks e pull requests, os quais são específicos do GitHub.

Multirepos vs Monorepos

Um VCS gerencia repositórios. Assim, uma organização precisa decidir os repositórios que vai criar em seu VCS. Uma decisão tradicional consiste em criar um repositório para cada projeto ou sistema da organização. Porém, soluções baseadas em um único repositório estão sendo adotadas com mais frequência, principalmente por grandes empresas, como Google, Facebook e Microsoft. Essas duas alternativas — chamadas, respectivamente, de **multirepos** e **monorepo** — são ilustradas nas próximas duas figuras.

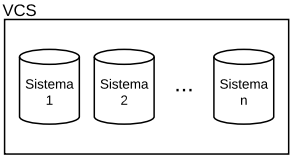


Figura 1.5: Multirepos: um VCS gerencia vários repositórios. Normalmente, um repositório por projeto ou sistema.

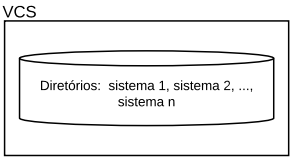


Figura 1.6: Monorepo: VCS gerencia um único repositório. Projetos são diretórios desse repositório.

Se pensarmos em contas do GitHub, podemos exemplificar da seguinte forma:

- Se optar por multirepos, uma organização terá vários repositórios, tais como `aserg-ufmg/sistema1`, `aserg-ufmg/sistema2`, `aserg-ufmg/sistema3`, etc.

- Se optar por monorepos, ela terá um único repositório — digamos, `aserg-ufmg/aserg-ufmg`. No diretório raiz desse repositório, teremos os subdiretórios `sistema1`, `sistema2`, `sistema3`, etc.

Dentre as vantagens de monorepos podemos citar:

- Como existe um único repositório, não há dúvida sobre qual repositório possui a versão mais atualizada de um arquivo. Isto é, com monorepos, existe uma única fonte de "verdade" sobre versões do código fonte.
- Monorepos incentivam o reuso e compartilhamento de código, pois os desenvolvedores têm acesso mais rápido a qualquer arquivo, de qualquer sistema.
- Mudanças são sempre atômicas. Com multirepos, dois commits podem ser necessários para implementar uma única mudança, caso ela afete dois sistemas. Com monorepos, a mesma mudança pode ser realizada por meio de um único commit.
- Facilita a execução de refactorings em larga escala. Por exemplo, suponha a renomeação de uma função utilitária que é usada em todos os sistemas da organização. Com monorepos, essa renomeação pode ser realizada com um único commit.

Por outro lado, monorepos requerem ferramentas para navegar em grandes bases de código. O motivo é que cada desenvolvedor terá em seu repositório local todos os arquivos de todos os sistemas da organização. Por isso, os responsáveis pelo monorepo do Google comentam que foram obrigados a implementar internamente um plug-in para a IDE Eclipse, que facilita o trabalho com uma base de código muito grande, como a que eles possuem na empresa (link).

1.3 Integração Contínua

Para explicar o conceito de Integração Contínua (CI), iniciamos com uma subseção de motivação. Em seguida, apresentamos o conceito propriamente dito. Feito isso, discutimos outras práticas que uma organização deve adotar junto com CI. Terminamos com uma breve discussão sobre cenários que podem desmotivar o emprego de CI em uma organização.

1.3.1 Motivação

Antes de definir o que é integração contínua, vamos descrever o problema que levou à proposta dessa prática de integração de código. Tradicionalmente, era comum o uso de branches durante a implementação de novas funcionalidades. Branches podem ser entendidos como um sub-diretório interno e virtual, gerenciado

pelo sistema de controle de versões. Nesses sistemas, existe um branch principal, conhecido pelo nome de **master** (quando usa-se Git) ou **trunk** (quando usa-se outros sistemas, como svn). Além do branch principal, os usuários podem criar seus próprios branches.

Por exemplo, antes de implementar uma nova funcionalidade, pode ser comum criar um branch para conter o seu código. Tais branches são chamados de **branches de funcionalidades (feature branches)** e, dependendo da complexidade da funcionalidade, eles podem levar meses para serem integrados de volta à linha principal de desenvolvimento. Logo, em sistemas maiores e mais complexos podem existir dezenas de branches ativos.

Quando a implementação da nova funcionalidade terminar, o código do branch deve ser "copiado" de volta para o master, por meio de um comando do sistema de controle de versões chamado **merge**. Nesse momento, uma variedade de conflitos pode ocorrer, os quais são conhecidos como **conflitos de integração** ou **conflitos de merge**.

Para ilustrar esse cenário, suponha que Alice criou um branch para implementar uma nova funcionalidade X em seu sistema. Como essa funcionalidade era complexa, Alice trabalhou de forma isolada no seu branch por 40 dias, conforme ilustrado na figura a seguir (cada nodo desse grafo é um commit). Observe que enquanto Alice trabalhava — realizando commits em seu branch — também ocorriam commits no branch principal.

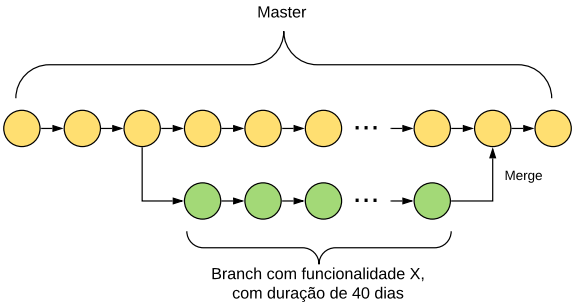


Figura 1.7: Desenvolvimento usando branches de funcionalidades.

Então, após 40 dias, quando Alice integrou seu código no master, surgiram diversos conflitos. Alguns deles são descritos a seguir:

- Para implementar a funcionalidade X, o código desenvolvido por Alice chamava uma função `f1`, que existia no master no momento da criação do branch. Porém, no intervalo de 40 dias, a assinatura dessa função foi modificada no master por outros desenvolvedores. Por exemplo, a função pode ter sido renomeada ou ter ganho um novo parâmetro. Ou ainda, em um cenário mais radical, `f1` pode ter sido removida da linha principal de desenvolvimento.
- Para implementar a funcionalidade X, Alice mudou o comportamento de uma função `f2` do master. Por exemplo, `f2` retornava seu resultado em milhas e Alice alterou o seu código para que o resultado fosse retornado em quilômetros. Evidentemente, Alice atualizou todo o código que chamava `f2` no seu branch, para considerar resultados em quilômetros. Porém, no período de 40 dias, surgiram novas chamadas de `f2`, que foram integradas no master, mas supondo um resultado ainda em milhas.

Em sistemas grandes, com milhares de arquivos, dezenas de desenvolvedores e de branches de funcionalidades, os problemas causados por conflitos podem assumir proporções consideráveis e atrasar a entrada em produção de novas funcionalidades. Veja que a resolução de conflitos é uma tarefa manual, que requer análise e consenso entre os desenvolvedores envolvidos. Por isso, os termos **integration hell** ou **merge hell** são usados para descrever os problemas que ocorrem durante a integração de branches de funcionalidades.

Adicionalmente, branches de funcionalidades, principalmente aqueles com duração longa, ajudam a criar silos de conhecimento. Isto é, cada nova funcionalidade passa a ter um dono, pois um desenvolvedor ficou dedicado a ela por semanas. Por isso, esse desenvolvedor pode sentir-se confortável para adotar padrões diferentes do restante do time, incluindo padrões para leiaute do código, para organização de janelas e telas, para acesso a dados, etc.

1.3.2 O que é Integração Contínua?

Integração Contínua (*continuous integration* ou CI) é uma prática de desenvolvimento proposta por Extreme Programming (XP), conforme estudamos no Capítulo 2. O princípio motivador da prática já foi enunciado na Introdução deste capítulo: se uma tarefa causa "dor", não podemos deixar que ela acumule. Em vez disso, devemos quebrá-la em subtarefas que possam ser realizadas de forma frequente. Como essas subtarefas são pequenas e simples, a "dor" decorrente da sua realização será menor.

Adaptando para o contexto de integração de código, sabemos que grandes integrações são uma fonte de "dor" para os desenvolvedores, pois eles têm que resolver de forma manual diversos conflitos. Assim, CI recomenda integrar o código de forma frequente, isto é, contínua. Como isso, as integrações serão pequenas e irão gerar menos conflitos.

Kent Beck, em seu livro de XP, defende o uso de CI da seguinte forma (link):

> "Você deve integrar e testar o seu código em intervalos menores do que algumas horas. Programação em times não é um problema do tipo dividir-e-conquistar. Na verdade, é um problema que requer dividir, conquistar e integrar. A duração de uma tarefa de integração é imprevisível e pode facilmente levar mais tempo do que a tarefa original de codificação. Assim, quanto mais tempo você demorar para integrar, maiores e mais imprevisíveis serão os custos."

Nessa citação, Beck defende várias integrações ao longo de um dia de trabalho de um desenvolvedor. No entanto, essa recomendação não é consensual. Outros autores, como Martin Fowler, mencionam pelo menos uma integração por dia por desenvolvedor (link), o que parece ser um limite mínimo para um time argumentar que está usando CI.

1.3.3 Boas Práticas para Uso de CI

Quando usa-se CI, o master é constantemente atualizado com código novo. Para garantir que ele não seja quebrado — isto é, deixe de compilar ou possua bugs —, recomenda-se o uso de algumas práticas em conjunto com CI, as quais vamos discutir a seguir.

Build Automatizado

Build é o nome usado para designar a compilação de todo os arquivos de um sistema, até a geração de uma versão executável. Quando se usa CI, o build deve ser automatizado, isto é, não incluir nenhum passo manual. Além disso, é importante que ele seja o mais rápido possível, pois com integração contínua ele será sempre executado. Alguns autores, por exemplo, chegam a recomendar um limite de 10 minutos para execução de um build (link).

Testes Automatizados

Além de garantir que o sistema compila sem erros após cada novo commit, é importante garantir também que ele continua com o comportamento esperado. Por isso, ao usar CI, deve-se ter uma boa cobertura de testes, principalmente testes de unidade, conforme estudamos no Capítulo 8.

Servidores de Integração Contínua

Por fim, os builds e testes automatizados devem ser executados com frequência, se possível após cada novo commit realizado no master. Para isso, existem **Servidores de CI**, que funcionam da seguinte forma (acompanhe também pela próxima figura):

- Após um novo commit, o sistema de controle de versões avisa o servidor de CI, que clona o repositório e executa um build completo do sistema, bem como roda todos os testes.
- Após a execução do build e dos testes, o servidor de CI notifica o usuário.

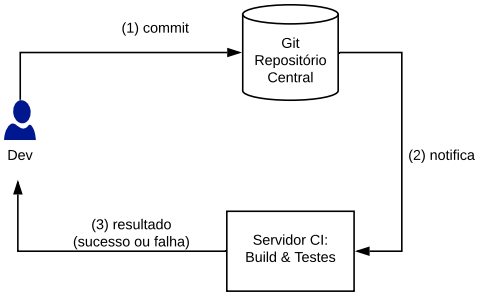


Figura 1.8: Servidor de Integração Contínua

O objetivo principal de um servidor de integração contínua é evitar a integração de código com problemas, sejam eles de build ou de comportamento. Quando o build falha, costuma-se dizer que ele "quebrou". Com frequência, o build na máquina do desenvolvedor pode ter sido concluído com sucesso. Mas ao ser executado no servidor de CI, ele pode falhar. Isso ocorre, por exemplo, quando o desenvolvedor esquece de realizar o commit de algum arquivo. Dependências incorretas são um outro motivo para quebra de builds. Por exemplo, o código pode ter sido compilado e testado na máquina do desenvolvedor usando a versão 2.0 de uma determinada biblioteca, mas o servidor de CI realiza o build usando a versão 1.0.

Se o servidor de CI notificar o desenvolvedor de que seu código não passou nos testes ou quebrou o build, ele deve parar tudo o que está fazendo e providenciar a correção. Isso é importante porque um build quebrado impacta o trabalho dos outros desenvolvedores, pois eles não vão conseguir mais compilar ou executar o sistema. Costuma-se dizer que nada em uma empresa de software tem maior prioridade do que a correção de um build quebrado. No entanto, a solução pode ser simplesmente reverter o código para a versão anterior ao commit com problemas.

Ainda nesta linha de raciocínio, um desenvolvedor somente deve avançar para uma próxima tarefa de programação após receber o resultado do servidor de CI. Por exemplo, ele não deve começar a escrever código novo, antes de ter certeza de que seu último commit passou pelo serviço de integração contínua. Ele também não deve iniciar outras tarefas importantes, como entrar em uma reunião, sair para almoçar ou ir para a casa, antes do resultado do servidor de CI.

Existem diversos servidores de integração contínua no mercado. Alguns deles são oferecidos como um serviço independente, normalmente gratuito para repositórios de código aberto, mas pago para repositórios privados de empresas. Assim, se você possui um repositório aberto no GitHub, existe mais de uma opção gratuita para ativar um serviço de CI no mesmo.

Uma dúvida comum é se CI é compatível com o uso de branches. Mantendo coerência com a definição de CI, a melhor resposta é a seguinte: sim, desde que os branches sejam integrados de forma frequente no master, via de regra, todo dia. Dizendo de outra forma, CI não é incompatível com branches, mas apenas com branches com um tempo de vida elevado. Por exemplo, Martin Fowler tem a seguinte observação sobre o uso de branches, especificamente branches de funcionalidades, junto com CI (link):

> "Na maioria das vezes, branches de funcionalidades constituem uma abordagem incompatível com CI. Um dos princípios de CI é que todos devem enviar commits para a linha de desenvolvimento principal diariamente. Então, a não ser que os branches de funcionalidades durem menos do que um dia, eles são um animal diferente de CI. É comum ouvir desenvolvedores dizendo que eles estão usando CI porque eles rodam builds automáticos, talvez usando um servidor de CI, após cada commit. Isso pode ser chamado de building contínuo e pode ser uma coisa boa. Porém, como não há integração, não podemos chamar essa prática de CI."

Desenvolvimento Baseado no Trunk

Como vimos, ao adotar CI, branches devem durar no máximo um dia de trabalho. Logo, o custo/benefício de criá-los pode não compensar. Por isso, quando migram para CI, é comum que as organizações usem também **desenvolvimento baseado no trunk** (*trunk based development* ou TBD). Com TBD, não existem mais branches para implementação de novas funcionalidades ou para correção de bugs. Em vez disso, todo desenvolvimento ocorre no branch principal, também conhecido com trunk ou master.

Mundo Real: TBD é usado por grandes empresas desenvolvedoras de software, incluindo Google e Facebook:

- No Google, "quase todo desenvolvimento ocorre no HEAD do repositório [isto é, no master]. Isso ajuda a identificar problemas de integração mais cedo e minimiza o esforço para realização de merges." (link)
- No Facebook, "todos engenheiros de front-end trabalham em um único branch que é mantido sempre estável, o que também torna o desenvolvimento mais rápido, pois não dispende-se esforço na integração de branches de longa duração no trunk." (link)

Programação em Pares

Programação em Pares (*Pair Programming*) pode ser considerada uma forma contínua de revisão de código. Quando adota-se essa prática, qualquer novo trecho de código é revisado por um outro desenvolvedor, que encontra-se sentado ao lado do desenvolvedor líder da sessão de programação. Portanto, assim como builds e testes contínuos, recomenda-se usar programação em pares com CI. Porém, esse uso também não é obrigatório. Por exemplo, o código pode ser revisado após o commit ser realizado no master. No entanto, nesse caso, como o código já foi integrado, os custos de aplicar a revisão podem ser maiores.

1.3.4 Quando não usar CI?

Os proponentes de CI definem um limite rígido para integrações no master: pelo menos uma integração por dia por desenvolvedor. No entanto, dependendo da organização, do domínio do sistema (que pode ser um sistema crítico) e do perfil dos desenvolvedores (que podem ser iniciantes), pode ser difícil seguir esse limite.

Por outro lado, é bom lembrar que esse limite não é uma lei da natureza. Por exemplo, talvez seja mais factível realizar uma integração a cada dois ou três dias. Na verdade, qualquer prática de Engenharia de Software — incluindo integração contínua — não deve ser considerada ao pé da letra, isto é, exatamente como está descrita no manual ou neste livro texto. Adaptações justificadas pelo contexto da organização são possíveis e devem ser consideradas. Experimentação com diferentes intervalos de integração pode também ajudar a definir a melhor configuração para uma organização.

CI também não é compatível com projetos de código livre. Na maioria das vezes, os desenvolvedores desses projetos são voluntários e não têm disponibilidade para trabalhar diariamente no seu código. Nesses casos, um modelo baseado em Pull Requests e Forks, conforme usado pelo GitHub, é mais adequado.

1.4 Deployment Contínuo

Com integração contínua, código novo é frequentemente integrado no branch principal. No entanto, esse código não precisa estar pronto para entrar em produção. Ou seja, ele pode ser uma versão preliminar, que foi integrado para que os outros desenvolvedores tomem ciência da sua existência e, consequentemente, evitem conflitos de integração futuros. Por exemplo, você pode integrar uma versão preliminar de uma tela, com uma interface ainda ruim. Ou então, uma versão de uma função com problemas de desempenho.

Porém, existe mais um passo da cadeia de automação proposta por DevOps, chamado de **Deployment Contínuo (Continuous Deployment ou CD)**. A diferença entre CI e CD é simples, mas seus impactos são profundos: quando usa-se CD, todo novo commit que chega no master pode entrar rapidamente em produção, em questões de horas, por exemplo. O fluxo de trabalho quando se usa CD é o seguinte:

- O desenvolvedor desenvolve e testa na sua máquina local.
- Ele realiza um commit e o servidor de CI executa novamente um build e os testes de unidade.
- Algumas vezes no dia, o servidor de CI realiza testes mais exaustivos com os novos commits que ainda não entraram em produção. Esses testes incluem, por exemplo, testes de integração, testes de interface e testes de desempenho.
- Se todos os testes passarem, os commits entram imediatamente em produção. Isto é, os usuários já vão interagir com a nova versão do código.

Dentre as vantagens de CD, podemos citar:

- CD reduz o tempo de entrega de novas funcionalidades. Por exemplo, suponha que as funcionalidades F1, F2,..., Fn estão previstas para uma nova release de um sistema. No modo tradicional, todas elas devem ser implementadas e testadas, antes da liberação da nova release. Por outro lado, com CD, as funcionalidades são liberadas assim que ficam prontas. Ou seja, CD diminui o intervalo entre releases. Passa-se a ter mais releases, mas com um menor número de funcionalidades.
- CD torna novas releases (ou implantações) um "não-evento". Explicando melhor, não existe mais um dia D ou um deadline para entrega de novas releases. Deadlines são uma fonte de stress para desenvolvedores e operadores de sistemas de software. A perda de um deadline, por exemplo, pode fazer com que uma funcionalidade somente entre em produção meses depois.
- Além de reduzir o stress causado por deadlines, CD ajuda a manter os desenvolvedores motivados, pois eles não ficam meses trabalhando sem receber feedback. Em vez disso, os desenvolvedores rapidamente recebem retorno — vindo de usuários reais — sobre o sucesso ou não de suas tarefas.
- Em linha com o item anterior, CD favore experimentação e um estilo de desenvolvimento orientado por dados e feedback dos usuários. Novas funcionalidades entram rapidamente em produção. Com isso, recebe-se retorno dos usuários, que podem recomendar mudanças na implementação ou, no limite, o cancelamento de alguma funcionalidade.

Mundo Real: Diversas empresas que desenvolvem sistemas Web usam CD. Por exemplo, Savor e colegas reportam que no Facebook cada desenvolvedor coloca em produção, na média, 3.5 atualizações de software por semana (link). Em cada atualização, na média, 92 linhas de código são adicionadas ou modificadas. Esses números revelam que, para funcionar bem, CD requer que as atualizações de código sejam pequenas. Portanto, os desenvolvedores têm que desenvolver a habilidade de quebrar qualquer tarefa de programação (por exemplo, uma nova funcionalidade, mesmo que complexa) em partes pequenas, que possam ser implementadas, testadas, integradas e entregues rapidamente.

1.4.1 Entrega Contínua (Continuous Delivery)

Deployment Contínuo (CD) não é recomendável para certos tipos de sistemas, incluindo sistemas desktop (como uma IDE, um navegador Web, etc), aplicações móveis e aplicações embutidas em hardware. Provavelmente, você não gostaria de ser notificado diariamente de que existe uma nova versão do navegador que usa em seu desktop, ou do sistema de rede social que usa em seu celular ou ainda de que um novo driver está disponível para sua impressora. Esses sistemas demandam um processo de instalação que não é transparente para seus usuários, como é a atualização de um sistema Web.

No entanto, mesmo nos sistemas mencionados no parágrafo anterior, pode-se usar um versão mais *fraca* de CD, chamada de **Entrega Contínua (Continuous Delivery)**. A ideia é simples: todo commit *pode* entrar em produção imediatamente. Ou seja, os desenvolvedores devem programar como se isso fosse acontecer. No entanto, existe uma autoridade externa — um gerente de projetos ou de releases, por exemplo — que toma a decisão sobre quando os commits, de fato, serão liberados para os usuários finais. Inclusive forças de mercado ou de estratégia da empresa podem influenciar nessa decisão.

Uma outra maneira de explicar esses conceitos é por meio da seguinte diferença:

- **Deployment to production** é o processo de liberar uma nova versão de um sistema para seus usuários.
- **Delivery** é o processo de liberar uma nova versão de um sistema para ser objeto de deployment.

Quando adota-se Deployment Contínuo, ambos os processos são automáticos e contínuos. Porém, com Entrega Contínua, a entrega é realizada com frequência, mas o deployment depende de uma autorização manual.

Mundo Real: Vamos citar alguns dados sobre a frequência de deployments em sistemas não-Web. Por exemplo, o Google libera novas releases do navegador Chrome para o público a cada seis semanas. Até 2019, a IDE Eclipse tinha uma única nova release por ano. A partir de 2019, o sistema passou a ter uma nova release a cada 13 semanas. Um dos motivos foi "permitir que os desenvolvedores liberem novas funcionalidades de forma rápida". Como um terceiro exemplo, a versão para Android do Facebook sofria uma atualização a cada oito semanas. Mais recentemente, o Facebook encurtou esse tempo para uma semana (link). Ou seja, as empresas estão lançando releases de

forma mais rápida, para agradar os usuários, receber feedback, manter os desenvolvedores motivados e continuarem competitivas no mercado.

1.4.2 Feature Flags

Nem sempre todo commit estará pronto para entrar imediatamente em produção. Por exemplo, o desenvolvedor pode estar trabalhando em uma funcionalidade X, mas ainda falta implementar parte de seus requisitos. Portanto, um desenvolvedor pode se perguntar:

Se novas releases são liberadas quase todo dia, como evitar que minhas implementações parciais, que ainda não foram completamente testadas ou que têm problemas de desempenho, cheguem até os usuários finais?

Uma solução seria não integrá-las no branch principal de desenvolvimento. Porém, não queremos mais retroceder e usar essa prática, pois ela leva ao que chamamos de *integration (ou merge) hell*. Dizendo de outro modo, não queremos abrir mão de Integração Contínua (CI).

A solução para esse problema é simples: integre continuamente o código parcial da funcionalidade X, mas com ela desabilitada, isto é, qualquer código relativo a X estará "guardado" por uma variável booleana (um *flag*) que, enquanto a implementação de X não estiver concluída, vai avaliar como falso. Um exemplo hipotético é mostrado a seguir:

```
featureX = false;
...
if (featureX)
   "aqui tem codigo incompleto de X"

...

if (featureX)
   "aqui tem mais codigo incompleto de X"
```

No contexto de deployment contínuo, variáveis usadas para desativar a execução de implementações parciais de funcionalidades são chamadas de **Feature Flags** ou **Feature Toggles**.

Para mostrar um segundo exemplo, suponha que você está trabalhando em uma nova página de um certo sistema. Então, você pode declarar uma feature flag para desabilitar o carregamento da nova página, como mostrado a seguir:

```
nova_pag = false;
...
if (nova_pag)
   "carregue nova página"
else
   "carregue página antiga"
```

Esse código vai para produção enquanto a nova página não estiver pronta. Porém, durante a implementação, localmente, você pode habilitar a nova página, fazendo `nova_pag` receber `true`.

Observe que durante um certo intervalo de tempo vai existir duplicação de código entre as duas páginas. Porém, após a nova página ser aprovada, entrar em produção e receber feedback positivo dos clientes, o código da página antiga pode ser removido. Ou seja, a duplicação de código foi temporária.

Mundo Real: Pesquisadores de duas universidades canadenses, liderados pelos professores Peter Rigby e Bram Adams, realizaram um estudo sobre o uso de feature flags ao longo de 39 releases do navegador Chrome, relativas a cinco anos de desenvolvimento (link). Nesse período, eles encontraram mais de 2.400 feature flags distintas no código do navegador. Na primeira versão analisada, eles catalogaram 263 flags; na última versão, o número aumentou para 2.409 flags. Na média, uma nova release adicionava 73 novos flags e removia 43 flags. Por isso, o crescimento observado no estudo.

Bibliografia

Gene Kim, Jez Humble, John Willis, Patrick Debois. Manual de Devops. Como Obter Agilidade, Confiabilidade e Segurança em Organizações Tecnológicas. Alta Books, 2018.

Jez Humble, David Farley. Entrega Contínua: Como Entregar Software de Forma Rápida e Confiável. Bookman, 2014.

Steve Matyas, Andrew Glover, Paul Duvall. Continuous Integration: Improving Software Quality and Reducing Risk. Addison-Wesley, 2007.

Exercícios de Fixação

1. Defina e descreva os objetivos de DevOps.

2. Em sites de oferta de empregos na área de TI, é comum encontrar vagas para "Engenheiro DevOps", requerendo habilidades como as seguintes:

   - Ferramentas de controle de versão (Git, Bitbucket, SVN, etc)
   - Gerenciadores de dependência e build (Maven, Gradle e etc)
   - Ferramentas de integração contínua (Jenkins, Bamboo, VSTS)
   - Administração de servidores em Cloud (AWS e Azure)
   - Sistemas Operacionais (Ubuntu, CentOS e Red Hat)
   - Banco de dados (DynamoDB, Aurora Mysql)
   - Docker e orquestração de docker (Kubernetes, Mesos, Swarm)
   - Desenvolvimento com APIs REST, Java

   Considerando a definição de DevOps que usou como resposta no exercício anterior, você considera adequado que a função de um funcionário seja "Engenheiro DevOps"? Justifique a sua resposta.

3. Defina (e diferencie) os seguintes termos: integração contínua (*continuous integration*); entrega contínua (*continuous delivery*) e deployment contínuo (*continuous deployment*).

4. Por que integração contínua, entrega contínua e deployment contínuo são práticas importantes em DevOps? Na sua resposta, considere a definição de DevOps que deu no primeiro exercício.

5. Pesquise o significado da expressão "Teatro de CI" (*CI Theater*) e então descreva-o com suas próprias palavras.

6. Descreva um problema (ou dificuldade) que surge quando decide-se usar *feature flags* para delimitar código que ainda não está pronto para entrar em produção.

7. Linguagens como C possuem suporte a diretivas de compilação condicional do tipo `#ifdef` e `#endif`. Pesquise o funcionamento e o uso dessas diretivas. Qual a diferença entre elas e *feature flags*?

8. Quando uma empresa usa CI, normalmente ela não usa mais branches de funcionalidades (*feature branches*). Em vez disso, ela tem um único branch, que é compartilhado por todos os desenvolvedores. Essa prática é chamada *Trunk Based Development* ou TBD, conforme estudamos neste capítulo. No entanto, TBD não significa que branches deixam de ser usados em tais

empresas. Descreva então um outro uso para branches, que não seja como *feature branches*.