

(NOME DOS INTEGRANTES)

**CONTROLE DE VERSIONAMENTO**

NOME DOS INTEGRANTES

**CONTROLE DE VERSIONAMENTO**

Trabalho de fixação de conteúdo apresentado á Proz Educação, como requisito parcial para aprovação em módulo do curso de Técnico em Desenvolvimento de Sistemas

Professor: Juliano **(sobrenome)**

**Resumo**

O desenvolvimento de softwares é uma tarefa complexa, geralmente demorada, e comumente conta com o envolvimento de varias pessoas. Neste contexto, podemos citar a criação de novos softwares desde o zero e correções ou melhorias em softwares já desenvolvidos. A medida que um projeto evolui, é preciso ter o controle de tudo que foi feito dentro do código-fonte para que se possa regredir o código em caso de falhas, ou ter o controle da eficiência das mudanças feitas. Em caso de projetos que contam com colaboração múltipla, esse controle se faz necessário para que se possa rastrear quem mudou o quê, por quê e pra quê. Assim sendo, este trabalho busca explorar as possibilidades de controle de versionamento e seus impactos no desenvolvimento de softwares, através de diferentes ferramentas e metodologias, mas com foco na apresentação da ferramenta de maior popularidade para versionamento de software: O Git.

**PALAVRAS-CHAVE:** Versionamento; Github; Commit; SemVer; Repositório.

**Abstract**

Software development is a complex and often time-consuming task that frequently involves the collaboration of multiple individuals. In this context, we can consider both the creation of new software from scratch and the maintenance, bug fixes, or enhancements of existing software. As a project evolves, it becomes essential to maintain control over all changes made within the source code. This control allows for code rollback in case of failures and ensures an understanding of the effectiveness of the modifications. For projects with multiple collaborators, this control becomes crucial to track who changed what, why, and for what purpose. Consequently, this work aims to explore the possibilities of version control and its impact on software development. We will examine various tools and methodologies, with a particular focus on introducing the most popular version control tool: Git.  
  
**KEY-WORDS:** Version control; GitHub; Commit; SemVer; Repository

1. **Introdução**

O desenvolvimento de softwares dificilmente é uma tarefa solitaria. Projetos robustos e multifuncionais pedem a colaboração de equipes multidisciplinares com profissionais como desenvolvedores front-end, back-end, full-stack, mobile, desenvolvedores de banco de dados, Dev Ops, Ethical Hackers dentre outros. E comumente essas equipes precisam trabalhar simultaneamente dentro do mesmo código fonte, seja no mesmo bloco de programação, mesma função do código ou blocos e funções diferentes.

Além da necessidade de atuação simultanea em diferentes parte do código fonte, é necessário que haja um controle do que é feito dentro do código, principalmente em cenários em que o código já esteja estruturado e operacional, e precisa-se garantir que as alterações não impossibilitarão o software de funcionar corretamente. Em outras palavras, é como se precisassem de uma máquina do tempo para que pudessem alterar o código-fonte mas com uma medida de segurança para retornar o código ao seu estado original em caso de problemas com as alterações.

Existem ferramentas que funcionam como essas máquinas do tempo. Elas registram o código em sua forma original, aponta as alterações que foram feitas em relação ao estado anterior do código, indica quem as fez em caso de trabalho coletivo, e possibilita o desenvolvimento paralelo do mesmo bloco de código, permitindo a fusão dessas linhas paralelas de desenvolvimento posteriormente. E, se tudo der errado, basta voltar ao estado inicial do código. Desde a década de 90 existe esse tipo de ferramenta como CVS, ClearCase, Source-Safe e SVN.

Mas foi somente em meados de 2000 que surgiram os sistemas de controle de versão mais modernos como Mercurial, Bazaar e o que possui maior abrangência no mercado hoje: O Git.

1. **Sistemas de versionamento e a evolução das ferramentas de versionamento**

Apesar da história da programação se dar inicio na primeira geração de computadores em 1940 com o surgimento do ENIAC, que funcionava com válvulas e cartões perfurados e consistia numa programação de baixo nível com instriuções escritas manualmente, o surgimento dos sistemas de controle de versão aconteceu em meados de 1972. Nessa década surgiu o SCCS (Source Code Control System), desenvolvida na Bell Labs por Marc Rochkind, para computadores IBM System/370 com o sistema operacional OS/360. Entretanto o sistema tinha algumas limitações notáveis: O SCCS naõ permitia trabalhar com um projeto inteiro, de forma que tudo fosse trabalhado de forma individualizada, o que dificultava o controle de versão dado o grande volume de dados; Era um SCV (Sistema de controle de versionamento) centralizado; Mantinha apenas o estado atual do codigo fonte e registrava as alterações feitas, mas não permitia o resgate de versoes anteriores; A sintaxe das versões era complicada, dito que ainda não existia o SemVer (Semantic Versioning); Não havia como trabalhar com *branchs* (Ramificações no projeto original); Tinha capacidade de trabalho reduzida ao trabalhar com arquivos binários; Não possuia mecanismos de defesa contra manipulações diretas nos arquivos de controle de versão.

Cerca de 10 anos depois, em 1982, surge a primeira resposta direta ao SCCS: O RCS (Revision Control System), desenvolvido por Walter F. Tichy enquanto ele estava na Universidade de Purdue. O RCS ainda era um sistema centralizado, mas com uma interface mais fácil e com resposta de armazenamento e leitura mais eficientes por salvar as alterações localmente e só depois as sincronizar com açoes de *push* e *pull no servidor central*. Trouxe também a criação de *branchs* e a possibilidade de recuperação de versões anteriores dos códigos.   
  
Nos anos 90, surge o CVS (Concurrent Version System), que permitia a colaboração remota, superando as limitações do RCS. E no inicio dos anos 2000 surge o SVN (Apache Subversion), com capacidade de rastrear arquivos renomeados, agrupar mudanças em um único commit, e maior eficiencia na comunicação entre cliente e servidor, dentre outras vantagens diante do CVS. Mas até então, todos eram centralizados.

Somente em 2005 que tivemos o surgimento do GIT, que resolveu a maioria esmagadora das limitações dos concorrentes mais velhos, e trouxe novos conceitos que iriam revolucionar os sistemas de versionamento. O Git foi lançado em 2005 por Linus Torvalds e outros colaboradores Linux, como um software de código aberto, o que tornava o GIT um SCV muito flexível e dinâmico. O GIT apresentava velocidade de operação, tornando ações como *commit*, *branch* e *merge* tarefas rápidas e simples; Possuia *commits* atomicos, o que possibilitava o rastreamento preciso dos *commits* e histórico completo de alterações, inclusive renomeio e movimentação de arquivos, facilitando as ações de *rollback* em casos de falhas; Prestava bom suporte a links simbólicos e outros tipos de arquivos, e o principal: Era o primeiro SCV distribuído. Ou seja: As alterações eram feitas localmente na máquina do desenvolvedor, podendo serem feitas até mesmo em ambiente offline, e posteriormente eram sincronizadas com o código fonte que estava hospedado em um serviço centralizado, o que possibilitava o trabalho simultaneo de desenvolvedores de qualquer lugar, em qualquer área do código. Desde então, o GIT é a ferramenta de versionamento mais versátil e mais usada até nos dias de hoje.

Mas em 2008 o GIT passou por uma integração que consolidou sua liderança no mercado. Chris Wanstrath, J. Hyett, Tom Preston-Werner e Scott Chacon lançaram o GitHub.

O GitHub é uma plataforma que hospeda os códigos utilizados pelo GIT para versionamento e oferece alguns recursos adicionais como controle de acesso, rastreamento de problemas, integração contínua e outras funcionalidades. Ele facilitou o emprego do SCV distribuído, permitindo que desenvolvedores colaborem em projetos de código aberto e criem *pull requests* como sugestões de melhoras ou correções em projetos públicos, trouxe wikis criando informação centralizada e acessível para todos, e hospeda gratuitamente sites estáticos, o que o torna numa ótima opção para portfólios e documentações.   
  
Apesar de existir concorrentes como GitLab, BitBucket, SourceForge, Gitea e Mercurial, o GitHub ainda é o mais utilizado para pequenos e grandes projetos e/ou empresas.

1. **Beneficios do versionamento**

Partindo da premissa que raramente o desenvolvimento de softwares é uma tarefa individual, o controle de versionamento se torna indispensável para garantir a segurança, a celeridade e eficiencia no desenvolvimento do projeto. Segundo Dias (2016), 4 perguntas devem ser feitas para identificar a necessidade de uso de um SCV. Se a resposta de 1 delas for “sim”, então um SCV deve ser implementado. São elas:

1. “Alguém já subscreveu o código de outra pessoa por acidente e acabou perdendo as alterações ?”
2. “Têm dificuldades em saber quais as alterações efetuadas em um programa, quando foram feitas e quem fez ?”
3. “Tem dificuldade em recuperar o código de uma versão anterior da que está em produção ?”
4. “Têm problemas em manter variações do sistema ao mesmo tempo ?”

Um SCV apropriado as necessidades do projeto e da equipe, podem impactar positivamente atraves de:

* **Histórico de alterações:** O SCVpermite criar um histórico com todas as alterações feitas dentro do código fonte. Desde a inclusão de linhas de comando, exclusão e modificações até a movimentação de seções de código, arquivos ou um simples renomeamento de algum elemento.
* **Controle de versão e segurança/confiabilidade:** Alguns SCV permitem retornar rapidamente o código a um estado anterior em caso de falhas, insatisfações com as mudanças, ou em caso de alterações inadvertidas/não autorizadas, garantindo confiabilidade ao código. No GitHub, há o uso do *Pull request,* onde uma alteração só é feita caso um moderador autorize a alteração no código fonte raiz.
* **Teste e experimentação:** Com versões controladas, podendo escolher qual versão atual ou passado deve ser executada, testes de novas funcionais, correções de bugs ou quaisquer outras demandas ficam mais fáceis e eficientes de serem executadas no projeto.
* **Ramificação e mesclagem** : Os SCV mais modernos permitem o uso de *branchs,* o que permite o desenvolvimento simultaneo em áreas diferentes do código, ou na mesma área, sem que isso impacte negativamente no processo, com uma gestão inteligente em caso de conflitos após a mescla das alterações, como alterações na mesma linha de código, por exemplo.
* **Colaboração eficiente:** Especialmente nos SCV com ambiente distribuído, a colaboração em equipe fica muito mais fácil, uma vez que o trabalho pode ser feito por diversas pessoas de varidas localidades, de forma online ou offline e com eficiência e tempo de resposta curto, uma vez que as mudanças são feitas inicialmente em cópias locais do código e somente após ações de *pull* são submetidas ao servidor para *merge.*

Mas, de acordo com Paracelso (Século XVI), “todas as substâncias são venenos. Não existe nada que não seja veneno. Somente a dose correta diferencia o veneno do remédio”. Então, um SCV inadequado as necessidades do projeto e da equipe, uma equipe sem entendimento comum da ferramenta, ou uma gestão inadequada do SCV pode ter efeito reverso, e dificultar muito o desenvolvimento do projeto.

1. **Controle de versão**

O controle de versão é um sistema que registra alterações em arquivos ao longo do tempo, permitindo revisitar versões específicas.

**4.1 Conceitos Básicos:**

* Repositório: Armazena todos os arquivos e suas versões.
* Commit: Registro de uma versão específica do projeto.
* Branch: Linha de desenvolvimento paralela.
* Merge: Junta mudanças de diferentes branches.
* Tag: Marca versões importantes do projeto.

**4.2 Sistemas de Controle de Versão:**

* Git: Sistema distribuído, popular pela eficiência.
* Subversion (SVN): Sistema centralizado.
* Mercurial: Sistema distribuído semelhante ao Git.

**4.3 Fluxo de Trabalho Comum em Git:**

1. Clonar repositório.
2. Criar branch.
3. Fazer alterações e commit.
4. Puxar alterações remotas.
5. Fundir branches.
6. Enviar alterações.

**4.4 Melhores Práticas:**

* Fazer commits frequentes.
* Usar branches de funcionalidades.
* Realizar revisões de código.
* Utilizar integração contínua.

**4.5 Resolução de Conflitos:**

* Editar manualmente arquivos conflitantes e fazer commit.

**4.5 Ferramentas Populares:**

* GitHub: Plataforma para hospedagem e revisão de código.
* GitLab: Similar ao GitHub com integração contínua.
* Bitbucket: Suporta Git e Mercurial.

1. **Práticas de Versionamento**
   1. **Fluxos de trabalho com Git**

**GitFlow**

Diagrama

Descrição gerada automaticamenteO **Gitflow** é um modelo alternativo de ramificação do Git que consiste no uso de ramificações de recursos e várias ramificações primárias. Este fluxo de trabalho não adiciona novos conceitos ou comandos além do necessário para o fluxo de trabalho de ramificação de recurso, o que ele faz é atribuir funções bem específicas para diferentes ramificações e definir quando elas devem interagir. Além das ramificações de recurso, ele utiliza ramificações individuais para preparar, manter e registrar lançamentos.

Imagem 1: Fluxo de trabalho no GitFlow

**GitHub Flow**

O **fluxo de GitHub** é um fluxo de trabalho leve e baseado no branch. Em Sistema de controle de versão, a base do código ganha o nome de tronco, linha de base, mestre ou linha principal. As ramificações (branching) — originadas direta ou indiretamente da linha principal –– permitem que os desenvolvedores testem novas funções isoladamente, mantendo o programa estável. Assim que as modificações estão prontas, basta juntá-las no branch principal e levá-las para o código estável.

Diagrama

Descrição gerada automaticamente com confiança baixaImagem 2: Fluxo de trabalho no GitHub Flow

**GitLab Flow**

O GitLab Flow é um fluxo de trabalho criado pelo GitLab em 2014. A diferença do **GitLab Flow** para o Git Flow é que ele possui três branchs principais ao invés de dois.

Branch principal:

* Main (Dev);
* Staging (Homologação);
* Productions (Produção);

Branches de suporte (Temporárias):

* ***Feature*** – Branch com novas funcionalidades. Quando finalizada, ela é removida após realizar o merge com a Branch Main.
* ***Bug Fix***: Uma branch criada a partir da Staging para realizar correções e no final ela faz o merge na Staging e na Main. A *branch* é removida após realizar o *merge*.
* ***Hotfix*** – Uma branch criada a partir da *Production* para realizar correções e no final ela faz o *merge* diretamente na *Staging* e *Production*. A branch é removida após realizar o *merge*.

Diagrama

Descrição gerada automaticamenteImagem 3: Fluxo de trabalho no GitLab Flow

# Comparando fluxos de trabalho do Git: o que você deve saber

Git é o sistema de controle de versões mais usado hoje em dia. Um fluxo de trabalho do Git é uma receita ou recomendação sobre como usar o Git para realizar o trabalho de maneira consistente e produtiva. Os fluxos de trabalho do Git incentivam os desenvolvedores e as equipes de [DevOps](https://www.atlassian.com/br/devops/what-is-devops) a aproveitar o Git com eficácia e estabilidade. O Git oferece muita flexibilidade em como os usuários gerenciam mudanças. Dado o foco do Git em flexibilidade, não há nenhum processo padronizado de como interagir com o Git. Ao trabalhar com uma equipe em um projeto gerenciado pelo Git, é importante ter certeza de que a equipe toda esteja de acordo sobre como o fluxo de mudanças vai ser aplicado. Para garantir que a equipe esteja alinhada, deve ser desenvolvido ou selecionado um acordo sobre o fluxo de trabalho do Git. Há vários fluxos de trabalho do Git divulgados que podem ser uma boa opção para a equipe. Aqui, vamos discutir algumas dessas opções de fluxo de trabalho do Git.

A matriz de possíveis fluxos de trabalho pode dificultar saber onde começar ao implementar o Git no local de trabalho. Esta página fornece um ponto de partida ao pesquisar os fluxos de trabalho mais comuns do Git para equipes de software.

Como você lê, lembre-se de que esses fluxos de trabalho são projetados para serem orientações e não regras concretas. Queremos mostrar o que é possível, então, você pode misturar e combinar aspectos de fluxos de trabalho diferentes para atender às suas necessidades individuais.

* 5.3 **O que é um fluxo de trabalho bem-sucedido do Git?**

Ao avaliar um fluxo de trabalho para sua equipe, o mais importante é considerar a cultura da equipe. Você quer que o fluxo de trabalho melhore a eficácia da equipe e não seja uma carga que limita a produtividade. Algumas coisas a considerar ao avaliar um fluxo de trabalho do Git são:

* Este fluxo de trabalho tem escalabilidade para diferentes tamanhos de equipe?
* É fácil de desfazer erros e enganos com este fluxo de trabalho?
* Este fluxo de trabalho adiciona alguma carga cognitiva desnecessária à equipe?

**Fluxo de trabalho centralizado**

O Fluxo de trabalho centralizado é um grande fluxo de trabalho do Git para equipes em transição do SVN. Como o Subversão, o Fluxo de trabalho centralizado usa um repositório central para servir como único ponto de entrada para todas as mudanças no projeto. Em vez de tronco, a ramificação de desenvolvimento padrão é chamada de branch main e todas as alterações recebem commit nesta ramificação. Este fluxo de trabalho não requer nenhum outro branch além do branch main.

Fazer a transição para um sistema de controle de versão distribuído pode parecer uma tarefa assustadora, mas você não tem que mudar seu fluxo de trabalho existente para aproveitar o Git. Sua equipe pode desenvolver projetos exatamente da mesma maneira que faz com o Subversão.

No entanto, usar o Git para impulsionar o fluxo de trabalho de desenvolvimento apresenta algumas vantagens sobre o SVN. Em primeiro lugar, dá a cada desenvolvedor sua própria cópia local do projeto inteiro. Este ambiente isolado permite que cada desenvolvedor trabalhe de modo independente de todas as outras mudanças em um projeto — ele pode adicionar confirmações ao repositório local e esquecer completamente sobre desenvolvimentos de upstream até que seja conveniente para ele.

Em segundo lugar, dá a você acesso ao robusto modelo de ramificação e mesclagem do Git. Ao contrário do SVN, as ramificações do Git foram projetadas para serem um mecanismo à prova de falhas para integrar o código e compartilhar alterações entre repositórios. O Fluxo de trabalho centralizado é semelhante a outros fluxos de trabalho na utilização de um repositório hospedado do lado do servidor remoto para que desenvolvedores enviem e extraiam alterações. Comparado a outros fluxos de trabalho, o Fluxo de trabalho centralizado não tem nenhum padrão definido de pull request ou ramificação. Um Fluxo de trabalho centralizado em geral é mais adequado para equipes menores e aquelas que estão migrando do SVN para o Git.

1. **Ferramentas de versionamento**
   1. **Ferramentas e plataformas populares**

**GitHub:**

GitHub e uma plataforma de hospedagem de código-fonte, cada projeto com código-fonte no GitHub e considerado um repositório, GIT é um sistema de controle de versão de código aberto (VCS) que permite salvar “snapshots” de um projeto de software. Ele registra as mudanças feitas em um programa e permite trabalhar em diferentes versões simultaneamente.

O GitHub oferece armazenamento ilimitado baseado em nuvem para repositórios. Além das funcionalidades básicas do GIT, o GitHub inclui gerenciamento de projetos, colaboração ferramentas de implantação para melhorar o fluxo de trabalho de desenvolvimento.

**GitLab:**

GitLab é uma plataforma de gerenciamento e operações de software de ponta a ponta.

* Controla e Problemas E Gerenciar Códigos-Fonte.
* Monitorar O Andamento De Tarefas Por Meio De Listas E Cards
* Automatizar Processos Como Testes E Publicação De Aplicações
* Criar Repositórios De Códigos Privados

**Bitbucket:**

A funcionalidade Bitbucket permite que as equipes de software colaborem com seu código-fonte, protejam sua base de códigos, reforcem os fluxos de trabalho de desenvolvimento e escalem conforme sua equipe cresce. Ele se integra ao JIRA Software para rastreabilidade, desde a concepção de recursos até a implantação. O Bitbucket fornece uma plataforma para os desenvolvedores discutirem alterações de código, uma janela para o progresso do desenvolvimento para gerentes de desenvolvimento e um sistema de controle de versão com desempenho para administradores do sistema.

**Recursos principais do BitBucket:**

* **Implementação/integração contínuas:** O Bitbucket oferece suporte a implementação e integração contínuas, permitindo que as equipes liberem software com mais frequência e confiabilidade.
* **Segurança na nuvem:** O Bitbucket é uma plataforma segura na nuvem, garantindo a proteção do seu código-fonte e dados.
* **Revisão de Código:** O Bitbucket facilita a revisão de código, permitindo que as equipes colaborem e melhorem a qualidade do código.
* Texto

  Descrição gerada automaticamente**Integrações:** O Bitbucket se integra a uma variedade de ferramentas e serviços, incluindo Jira, Jenkins e Slack.

Imagem 4: Comparativo entre GitHub, GitLab e BitBucket

1. **Desafios e Melhores Práticas**

7.1 **Principais Desafios no Versionamento de Código**

Gerenciar grandes repositórios em Git pode apresentar vários desafios.

**Desempenho e lentidão:**

Repositórios grandes podem ser lentos para clonar e fazer pull, especialmente se tiverem muitos arquivos grandes ou um histórico extenso.

Uma alternativa para solucionar esse problema é utilizar o método de clonagem superficial (shallow cloning), onde o desenvolvedor pode escolher a partir de qual versão do projeto ele quer clonar, em vez de clonar todo o repositório. Isso melhora significativamente o desempenho ao reduzir a quantidade de dados transferidos.

**Gerenciamento de histórico:**

Manter um histórico limpo e compreensível pode ser difícil.

Um histórico com muitos commits irrelevantes ou mal descritos pode ser confuso.  
Para resolver esse problema, é importante adotar algumas práticas essenciais. Primeiramente, incentive a criação de commits significativos e bem descritos, onde cada commit represente uma mudança lógica e independente no código, acompanhada de uma mensagem clara que explique a alteração.

**Conflitos de merge:**

Em equipes grandes, os conflitos de merge podem ser comuns, especialmente se muitas pessoas estiverem trabalhando nos mesmos arquivos ou áreas do código.  
Quando duas ou mais pessoas modificam o mesmo arquivo simultaneamente em diferentes partes do código, pode ocorrer um conflito de fusão durante a integração dessas alterações no repositório principal.

Para minimizar conflitos de merge, incentive a prática de integração contínua, onde os desenvolvedores frequentemente fazem pull das mudanças da branch principal e fazem merge em suas próprias branches. Isso ajuda a identificar e resolver conflitos menores antes que se tornem problemas maiores.

**Gerenciamento de branches:**

Manter muitas branches ativas pode ser complicado, especialmente se não houver uma estratégia clara para criação, uso e exclusão de branches.  
Estratégias como GitFlow ou GitHub Flow ajudam a padronizar o uso de ramificações. Ferramentas de integração contínua (CI) também podem ser usadas para automatizar a integração e testes em diversas ramificações.  
  
Um bom gerenciamento de branches é crucial para equipes de desenvolvimento que trabalham em projetos complexos e dinâmicos. Ele não apenas facilita o desenvolvimento paralelo de diferentes funcionalidades, mas também promove a estabilidade e a qualidade do código através de práticas como revisão de código e testes automatizados.

**Gerenciamento de permissões:**

Em grandes equipes, pode ser necessário implementar um controle detalhado de permissões para prevenir alterações não autorizadas.

O líder técnico ao estabelecer um controle detalhado de permissões, cada membro sabe exatamente quais são suas responsabilidades e onde podem atuar. Isso evita alterações não autorizadas e ajuda a manter o projeto organizado e seguro. Dessa forma, todos podem se concentrar em suas tarefas sabendo que o trabalho está protegido e sendo feito da maneira correta.

**7.2 Melhores práticas**

* **Commits pequenos e frequentes.**
* **Mensagens de commit claras e descritivas.**
* **Revisões de código e pull requests**

Para usar o Git de forma eficiente, é importante seguir algumas boas práticas que ajudam a organizar o trabalho e a colaboração na equipe. Uma delas é fazer commits pequenos e frequentes. Isso significa dividir as mudanças no código em partes menores, o que torna mais fácil acompanhar o que foi feito e resolver problemas rapidamente, se necessário.

Além disso, é essencial escrever mensagens de commit claras e descritivas. Essas mensagens explicam não apenas o que foi alterado no código, mas também por que essas mudanças foram feitas.

Isso ajuda todos na equipe a entenderem as alterações de maneira rápida e precisa, melhorando a comunicação geral.

Outra prática importante é usar revisões de código e pull requests. Isso significa que antes de incorporar as mudanças principais no projeto, outros membros da equipe revisam o código. Isso não só ajuda a identificar erros mais cedo, mas também promove boas práticas de programação e aumenta a colaboração entre os desenvolvedores, compartilhando conhecimentos e melhorando a qualidade do código ao longo do tempo.

Adotando essas práticas, as equipes podem não apenas melhorar a qualidade do código que produzem, mas também tornar o processo de desenvolvimento mais eficiente e colaborativo.

1. **Conclusão**
   1. **RECAPITULAÇÃO DA IMPORTÂNCIA DO VERSIONAMENTO**.

Como vimos, o Versionamento de Código ou Controle de Versão surgiu com o intuito de facilitar a dinâmica do trabalho entre programadores, possibilitando compartilhar e controlar as diversas alterações de um código raiz visando diminuir problemas e facilitar a execução de um programa ou aplicação.

Neste contexto, os benefícios do versionamento destacam-se pela facilidade de identificar as alterações e compartilhá-las aos colaboradores. Além de possibilitar a recuperação de versões anteriores dos códigos. Por isso, entender as nuances dos tipos de Sistemas de Controle de Versão (VCS ou SCV) se fez fundamental. Nisso vimos que os modelos ou tipos de VCS, como os do tipo Centralizados (CVCS), apresentam uma natureza hierárquica, controle rigoroso, e colaboração controlada, enquanto os do tipo Distribuídos (DVCS) apresentam modelo descentralizado, colaboração descentralizada e gestão de ramificações aprimoradas.

Dentre os Sistemas de Controle de Versão, o mais popular e conhecido entre os programadores como vimos, foi o Git, desenvolvido em 2005 por Linus Torvalds. Comparado a outros SCV, se destacou e ficou bastante conhecido a partir de 2010, por evitar o desgaste de armazenamento das diversas cópias de arquivos no sistema, e por impossibilitar a perda de versões que se queira reutilizar no futuro.

* 1. **PRINCIPAIS TÉCNICAS E FERRAMENTAS DISCUTIDAS.**

Vamos relembrar?

Falamos muito como o Git é uma ferramenta flexível e dinâmica, e para ilustrar isso iremos relembrar os conceitos ou comandos básicos do Git, e não bastando alguns conceitos que são comuns de se ouvir, iremos destacar o que é snapshot, commit, flags, e staging. Onde um snapshot, é como uma captura de algo em um determinado instante como uma foto. O commit, o comando que leva as mudanças para o repositório no Git, podendo ser instantâneos ou marcos ao longo do desenvolvimento de um projeto Git. O flags, um comando lógico de sinalização, para que finalize o processo de um programa. E o staging, a área que intermedia entre o diretório de trabalho e o repositório git.

Os conceitos ou comandos básicos do Git se dividem nos seguintes tópicos, e em cada tópico serão esclarecidos, sendo eles:

a. Inicialização e configuração

git init: inicia um novo repositório;

git config: configura as opções de instalação e/ou de usuário do git.

b. Enviar arquivos:

git add: antes de fazer o commit (projeto oficial);

git commit: submeter as mudanças;

git fetch: importar commits;

git pull: versão automatizada do git fetch;

git push: transfere commits;

arquivos.gitignore: para os arquivos que o Git deve ignorar na hora de fazer um commit;

git tag: cria etiquetas de estado relevantes.

c. Verificar informações:

git log: verificar as revisões passadas;

git diff: mostra alterações;

d. Receber arquivos:

git clone: cria cópia de um repositório já existente.

e. Alterar Branches:

git branch: gerencia as branches de um repositório;

git checkout: muda de Branch ou volta para algum estado do projeto;

git merge: faz mescla entre branchs;

git rebase: move as branches;

git stash: arquiva alterações não “commitadas”, volta para o estado do último commit, guardando as alterações adicionais.

Além dos comandos, no Git existem as tags, elas são como fotos de um determinado momento de um repositório, que também servem para ajudar a identificar a origem da falha em um código, e funcionam como apontadores fixos para um commit específico, e não avançam com novos commits.

Vimos também que a plataforma web mais utilizada quando se trata de Git, é a GitHub, devido a facilidade que ela permite ao trabalho em equipe, por qualquer integrante, tendo internet conseguir acessar os arquivos de um determinado projeto sem maiores problemas. E por ser uma espécie de rede social, é possível interagir com pessoas e os seus trabalhos. No GitHub, existe também os releases, que são as versões das aplicações disponíveis, e contribuem para a disponibilização do que seria na linguagem mais popular, a versão atualizada da aplicação.

Além do GitHub, existem outras plataformas como o GitLab e Bitbucker, que também facilitam a interação com os repositórios, pela adição de funcionalidades, e permitem identificar problemas, e rever códigos e também realizar integração continua e entrega continua. Por isso, não é de se estranhar que por essas e outras, o Git só tende a evoluir e revolucionar o mercado de desenvolvedores, pela facilidade de lidar com grandes projetos, por ser dinâmico e robusto.

1. **Referencia Bibliograficas**

3 Versionamento de código. Disponível em: <https://prdm0.github.io/aulas_computacional/versionamento-de-c%C3%B3digo.html>. Acesso em: 28 jun. 2024.

ATLASSIAN. Git commit. Disponível em: <https://www.atlassian.com/br/git/tutorials/saving-changes/git-commit>. Acesso em: 28 jun. 2024a.

ATLASSIAN. O que é um sistema distribuído? Disponível em: <https://www.atlassian.com/br/microservices/microservices-architecture/distributed-architecture>. Acesso em: 28 jun. 2024b.

MARCELA. Versionamento de código: entenda o que é e porque é importante. Disponível em: <https://awari.com.br/versionamento-de-codigo/?utm_source=blog&utm_campaign=projeto+blog&utm_medium=Versionamento%20de%20c%C3%B3digo:%20entenda%20o%20que%20%C3%A9%20e%20porque%20%C3%A9%20importante>. Acesso em: 28 jun. 2024.

MARKETING ZUP. Versionamento Semântico.

Disponível em: <https://www.zup.com.br/blog/versionamento-semantico>. Acesso em: 28 jun. 2024.

ROSSI, T. GIT: História, Evolução e Aplicações.

Disponível em: <https://www.dio.me/articles/git-historia-evolucao-e-aplicacoes>. Acesso em: 28 jun. 2024.

SANTANA, R. Git: entenda conceitos básicos sobre o sistema. Disponível em: <https://www.lumis.com.br/a-lumis/blog/git-conceitos-basicos.htm>. Acesso em: 28 jun. 2024.

Disponível em: <https://mundododev.com.br/2024/01/02/c-tipos-de-sistemas-de-controle-de-versao-centralizado-vs-distribuido/>. Acesso em: 28 jun. 2024a.

Disponível em: <http://www2.ic.uff.br/~ilaim/repeticao1.pdf>. Acesso em: 28 jun. 2024b.

Disponível em: <https://napoleon.com.br/glossario/o-que-e-git-staging-area/#:~:text=A%20Git%20Staging%20Area%20%C3%A9,serem%20inclu%C3%ADdos%20no%20pr%C3%B3ximo%20commit.>. Acesso em: 28 jun. 2024c.

Disponível em: <https://www.telecom.uff.br/pet/petws/downloads/apostilas/GIT.pdf>. Acesso em: 28 jun. 2024d.

DAVI FERREIRA SANTIAGO, FELIPE MEIRELES LEONEL, JOÃO VITOR SAADE SIMÃO. GIT INIT - Uma introdução ao controle de versão com Git. Universidade Federal de Minas Gerais, Escola de Engenharia, Bloco 3, Sala 1050.: PETEE-MG, 2023.

Criando release no GitHub. Disponível em: <https://treinaweb.com.br/blog/criando-release-no-github>. Acesso em: 28 jun. 2024.

CI/CD. Disponível em: <https://www.redhat.com/pt-br/topics/devops/what-is-ci-cd>. Acesso em: 28 jun. 2024.

**https://www.atlassian.com/br/git/tutorials/what-is-version-control**

**https://remsoft.com.br/blog/tecnologias/a-evolucao-dos-sistemas-de-controle-de-versao-uma-breve-historia-e-visao-atual/**

**https://www.dio.me/articles/entendendo-versionamento-de-codigo-distribuido-e-centralizado**

**http://osr507doc.sco.com/en/tools/SCCS.html**

**https://www.ibm.com/docs/pt-br/aix/7.3?topic=concepts-source-code-control-system**

**https://remsoft.com.br/blog/tecnologias/a-evolucao-dos-sistemas-de-controle-de-versao-uma-breve-historia-e-visao-atual/**

**https://www.escoladnc.com.br/blog/git-vs-github-entenda-a-diferenca-e-potencialize-seu-desenvolvimento/**

**https://docs.github.com/pt/get-started**

**https://www.atlassian.com/br/git/tutorials/comparing-workflows/gitflow-workflow#:~:text=O%20que%20%C3%A9%20Gitflow%3F,por%20Vincent%20Driessen%20no%20nvie.**

**Fontes: https://docs.github.com/pt/get-started/using-github/github-flow**

**https://tecnoblog.net/responde/o-que-e-branch-em-programacao/**

**GitLab Flow**

**https://www.zup.com.br/blog/git-workflow#:~:text=O%20GitLab%20Flow%20%C3%A9%20um,utilizar%20os%20nomes%20do%20exemplo.**