



# SCRUM e GitHub

# SUMÁRIO

<b>Versionamento</b> .....	4
Estratégia de versionamento .....	4
Versionamento de código .....	5
Versionamento de software .....	6
Versionamento semântico.....	7
<b>Git</b> .....	8
<b>Instalando o Git</b> .....	10
Windows .....	10
Linux/Unix .....	11
MacOs .....	12
<b>Comandos Git</b> .....	13
Criar um repositório .....	13
<i>git init</i> .....	13
<i>git clone</i> .....	14
<i>git config</i> .....	14
Analisando o repositório.....	15
<i>git status</i> .....	16
Salvar alterações .....	16
<i>git add</i> .....	17
<i>git commit</i> .....	17
<i>git diff</i> .....	18
Sincronizar repositório.....	19
<i>git remote</i> .....	19
<i>git push</i> .....	20
<i>git pull</i> .....	21
Trabalhando com branches (ramificações).....	22
<i>git branch</i> .....	22
<i>git checkout</i> .....	23
<i>git merge</i> .....	23

# SUMÁRIO

GitHub .....	24
Metodologias Ágeis .....	25
Tradicional x Ágil .....	26
Scrum .....	27
Papéis .....	28
Time Scrum .....	29
<i>Product Owner</i> .....	30
<i>Scrum Master</i> .....	31
Cerimônias .....	31
<i>Sprint Planning</i> .....	32
<i>Sprint</i> .....	32
<i>Daily Scrum</i> .....	33
<i>Review Meeting</i> .....	34
<i>Retrospective Sprint</i> .....	34
Artefatos .....	35
<i>Product Backlog</i> .....	35
<i>Sprint Backlog</i> .....	36
Entrega .....	36
Referências .....	37







Esse é um cenário em que podemos aplicar a estratégia de versionamento – a cada dia, vamos criando novas versões devido ao fato de que estaremos realizando adições, alterações ou remoções de informações no documento, nesse caso.

Essa mesma estratégia deve ser utilizada para a construção de software. Mas, antes de aprofundarmos ainda mais o assunto, vamos falar um pouco sobre os tipos de versionamento. Temos o versionamento de código, o versionamento semântico e o versionamento de software. A seguir, vamos falar um pouco mais sobre cada um deles.



## Versionamento de código

Conforme dito anteriormente, o desenvolvimento de software é realizado em fases, em que, a cada fase, realizamos um incremento ao nosso sistema, que possui um valor agregado para o cliente, por exemplo novas funcionalidades ou até mesmo melhorias a processos que já existem. Sendo assim, podemos imaginar que, a cada nova alteração realizada, teremos versões diferentes do nosso sistema, e isso é uma grande verdade.

Basicamente, o versionamento de código fonte é uma forma para que possamos lidar com essas várias alterações e, assim, consigamos administrar as mudanças de formas mais organizadas. A ideia principal é controlar as mudanças de uma forma que nos permita retornar a uma determinada versão caso exista alguma necessidade.

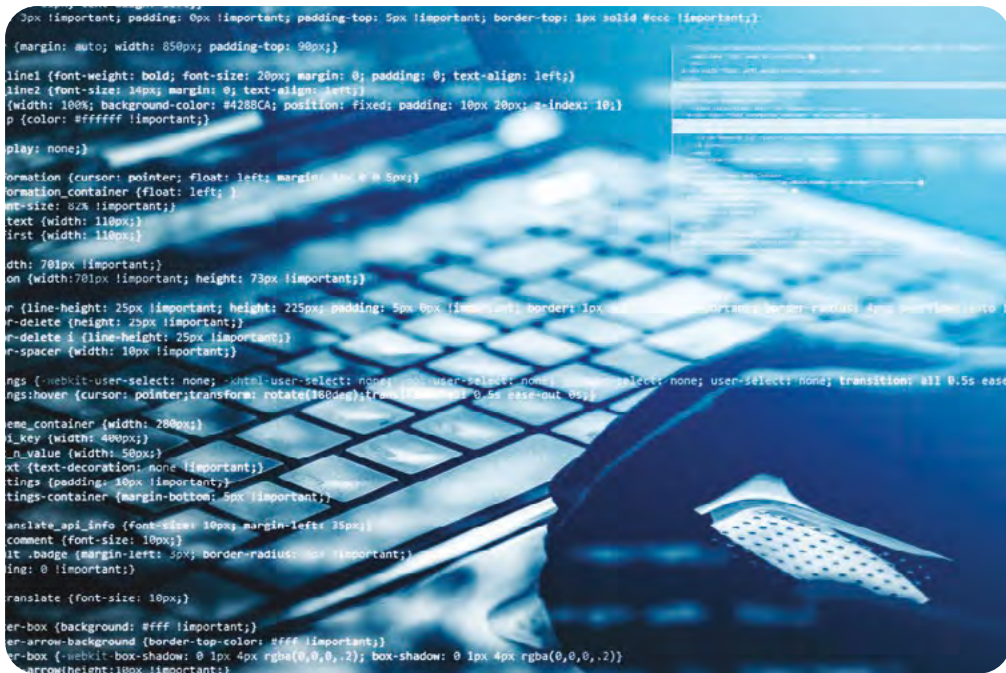
Além de tudo o que foi dito, pode ser considerada uma forma colaborativa de trabalho em equipe, pois, a partir do processo de versionamento de código, é possível identificar quem fez cada alteração, e onde e quando cada uma foi feita. Com isso, podemos tomar decisões baseadas no conjunto de alterações feitas em um determinado espaço de tempo. Por exemplo, caso o nosso sistema pare de funcionar por qualquer motivo, podemos identificar uma versão que estava estável e retornar a essa versão de forma simplificada.



## Versionamento de software

O processo de versionamento de software é utilizado para controlar a versão do sistema que está sendo desenvolvida. A partir do momento que essa aplicação é lançada, ela passa a receber uma versão específica. Por exemplo, imagine que você esteja criando uma página web que possui um botão “Clique para ver a data” e lance essa página como versão 1, e que, depois de uma semana, os usuários da sua página comecem a reportar que queriam visualizar a hora também. Então, você desenvolve a visualização da hora e lança a página como versão 2.

Essa prática é muito comum, pois, assumindo que além da atualização temos que levar em consideração a manutenibilidade do projeto, devemos atribuir versões ao nosso software, assegurando que possamos sempre ter acesso a uma versão estável. Retomando o exemplo anterior, suponhamos que no lançamento da versão 2 a sua página apresentasse algum problema, você poderia “relançar” a versão 1 enquanto estivesse trabalhando na solução do problema. Dessa maneira, o versionamento de software nos garante uma maior segurança no processo de entrega do nosso sistema.



## Versionamento semântico

Um tipo de versionamento que é complementar ao versionamento de software é o versionamento semântico. Com ele, podemos utilizar um conjunto de padrões para determinar as versões do nosso sistema, mantendo uma estrutura que permita entender facilmente que tipo de mudança foi realizada e por qual motivo.

A estrutura utilizada para o versionamento de software utilizando o versionamento semântico possui um formato em que temos: *"MAJOR.MINOR.PATCH"*. O *MAJOR*, ou versão Maior, é quando você realiza uma mudança incompatível com a versão anterior. O *MINOR*, ou versão Menor, é quando você adiciona funcionalidades mantendo a compatibilidade. E, por fim, o *PATCH*, ou versão de Correção, é quando você realiza correções de falhas mantendo a compatibilidade. MAJOR, MINOR e PATCH são números inteiros que, quando atribuídos, formam a versão do software, e que, isolados, apontam para determinadas informações, conforme vimos anteriormente.

O versionamento semântico possui o intuito de comunicar as mudanças realizadas no sistema e as possíveis possibilidades de atualizações. Por exemplo, atualmente o seu sistema está na versão 2.4.3 e a versão mais estável disponível para download no momento é a versão 2.5.8; será mais fácil, para o usuário, identificar que existe uma versão mais recente com base na ordem crescente nos números.





## GIT

Utilizamos o Git como nosso aliado no processo de desenvolvimento, pois é uma das ferramentas mais modernas e mais utilizadas para controle de versão. O Git é um projeto de código aberto maduro e mantido ativamente pela comunidade *open source*. Ele foi desenvolvido em 2005 pelo Linus Torvalds, o famoso criador do *kernel* do sistema operacional Linux.

Você pode acessar o Git através do link ou do código QR a seguir:

<https://git-scm.com/>



Você pode acessar o repositório com o código fonte da comunidade *open source* através do link ou do código QR a seguir:

<https://github.com/git/git>





Atualmente, diversos projetos dependem do Git para realizar controle de versão, incluindo grandes projetos comerciais e de código aberto. É um sistema de arquitetura distribuída que, em vez de possuir um único local para o armazenamento histórico completo, realiza a cópia de trabalho para cada desenvolvedor, em que cada um pode possuir o seu próprio repositório local, com todo o histórico de *commits*, o que traz uma série de vantagens, como a possibilidade de trabalhar sem conexão com a internet, além de otimizar o desenvolvimento de software.

Além de ser um sistema distribuído, o Git foi desenvolvido para obter maior desempenho, segurança e flexibilidade no processo de controle de versão. Pode ser considerado uma das melhores opções no diz respeito à performance por ser um sistema otimizado, em que os algoritmos implementados aproveitam de estruturas de dados para realizar a análise das árvores de arquivos, e pelo modo como esse sistema é modificado ao longo do tempo.



Vamos a um exemplo prático da utilização do Git e de como ele pode nos ajudar a ganhar mais desempenho no processo de desenvolvimento:

Imagine que você esteja desenvolvendo uma nova versão para o seu sistema, a versão 2.0, e que esteja adicionando novas funcionalidades. Você, então, recebe uma solicitação de reparo em uma versão mais antiga, a versão 1.5, desse mesmo sistema. Você irá realizar os *commits* das alterações já realizadas até o momento da versão 2.0 e irá para uma outra *branch* do seu repositório, com o intuito de corrigir o problema reportado anteriormente na versão 1.5. As alterações serão realizadas em uma outra versão do código e a alteração que você realizará afetarão apenas a versão mais antiga (versão 1.5). O objetivo é sanar o problema, salvar suas alterações, lançar a versão 1.5.1 e, então, retornar aos trabalhos na versão 2.0. Conforme mencionado anteriormente, esse processo ocorre de forma fácil, rápida e confiável, sem nenhum acesso à rede. Assim que você estiver confortável em subir suas alterações para um repositório remoto, você as enviará através de um *push* contendo as alterações realizadas.

Esse exemplo é um de diversos outros que podemos trazer quando falamos de Git e que nos mostra como é simples o processo de correção do código e como podemos, através de alguns comandos, realizar manutenções em nosso sistema, de maneira rápida e confiável. Mas, você deve ter percebido que existem diversas palavras que ainda não citamos até o momento. Pois bem, vamos começar a aprofundar nossos conhecimentos ainda mais no assunto.

# INSTALANDO O GIT

## Windows

1. Baixe o executável mais recente do Git para Windows, disponibilizado no site oficial do Git.
2. Quando iniciar o instalador, siga os avisos de *Next* e *Finish* para concluir a instalação.
3. Abra o *prompt* de comando (ou procure por "Git Bash") e escreva "git --version", então pressione *Enter*. O esperado é que seja retornada a versão do Git instalada na sua máquina.
4. Divirta-se.

Você pode acessar o executável mais recente do Git para Windows através do link ou do código QR a seguir:

<https://git-scm.com/download/win>



## Linux/Unix

### Debian/Ubuntu (apt-get)

1. Abra o *shell* para realizar a instalação utilizando o comando “sudo apt-get install git”.
2. Verifique se a instalação foi bem-sucedida utilizando o “git --version”; assim que a versão for retornada, você irá confirmar que o Git está instalado em seu computador.
3. Divirta-se.

### Fedora (yum)

1. Abra o *shell* para realizar a instalação utilizando o comando “sudo yum install git”.
2. Verifique se a instalação foi bem-sucedida utilizando o “git --version”; assim que a versão for retornada, você irá confirmar que o Git está instalado em seu computador.
3. Divirta-se.

Você pode acessar o passo a passo mais recente do Git para Linux através do link ou do código QR a seguir:

<https://git-scm.com/download/linux>





# MacOs

## Homebrew

1. Abra o terminal para realizar a instalação utilizando o comando “brew install git”.
2. Verifique se a instalação foi bem-sucedida utilizando o “git --version”; assim que a versão for retornada, você irá confirmar que o Git está instalado em seu computador.
3. Divirta-se.

## MacPorts

1. Abra o terminal para realizar a instalação utilizando o comando “sudo port install git”.
2. Verifique se a instalação foi bem-sucedida utilizando o “git --version”; assim que a versão for retornada, você irá confirmar que o Git está instalado em seu computador.
3. Divirta-se.

Você pode acessar o passo a passo mais recente do Git para macOS através do link ou do código QR a seguir:

<https://git-scm.com/download/mac>



Caso fique com alguma dúvida no processo de instalação, ou seu sistema operacional não tiver sido citado anteriormente, utilize a página oficial do Git como seu fiel escudeiro. Você pode acessar a página oficial do Git através do link ou do código QR a seguir:

<https://git-scm.com/>



# COMANDOS GIT

Agora que temos o Git instalado em nossa máquina, vamos nos aprofundar nos comandos, utilizando-os em nosso terminal/*prompt* de comando (ferramenta de linha de comando).

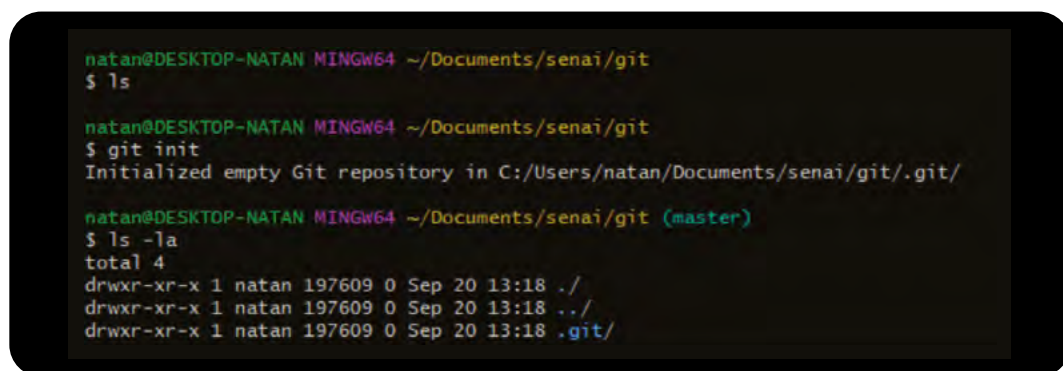
## Criar um repositório

Um repositório é uma estrutura de diretórios que dispõe de arquivos que irão resolver um problema em comum. Você pode ter um repositório contendo um código fonte para uma aplicação web, assim como você pode ter um repositório contendo arquivos de texto que serão utilizados para escrever um artigo. A ideia é que os repositórios sejam um armazenamento virtual que lhe permite salvar as versões do seu código/arquivos e acessá-las quando necessário.

Abaixo, seguem alguns comandos que são essenciais para o gerenciamento do repositório, mais especificamente para a criação de um.

### *git init*

O *git init* é uma forma de realizar a criação de um repositório em um diretório já criado. Por exemplo, imagine que você esteja começando seu projeto agora e que já deseje criar o seu repositório local, sem comunicação com um servidor remoto. Para isso, basta utilizar o comando *git init*, que realizará toda a configuração inicial do repositório, além de realizar a criação da *branch* principal.



```
natan@DESKTOP-NATAN MINGW64 ~/Documents/senai/git
$ ls

natan@DESKTOP-NATAN MINGW64 ~/Documents/senai/git
$ git init
Initialized empty Git repository in C:/Users/natan/Documents/senai/git/.git/

natan@DESKTOP-NATAN MINGW64 ~/Documents/senai/git (master)
$ ls -la
total 4
drwxr-xr-x 1 natan 197609 0 Sep 20 13:18 ./
drwxr-xr-x 1 natan 197609 0 Sep 20 13:18 ../
drwxr-xr-x 1 natan 197609 0 Sep 20 13:18 .git/
```

Figura 1 - Imagem do console de desenvolvimento ilustrando a inicialização de um repositório Git local

Fonte: Do autor (2022)

A imagem apresentada ilustra o processo de inicialização de um novo repositório. Inicialmente, tínhamos um diretório vazio, e a partir da execução do *git init*, o diretório “.git/” foi criado, com o intuito de armazenar todas as configurações do nosso repositório. Existe uma questão muito interessante que podemos considerar para avaliar se o repositório realmente foi criado: a primeira é utilizar o comando “*ls -la*” para listarmos arquivos/diretórios ocultos (que possuem o ponto no início do nome); a outra é verificar se existe o nome da *branch* corrente – na imagem apresentada anteriormente, o nome da *branch* é *master*, que foi configurada assim que criamos o repositório.

## git clone

Uma outra forma de trabalharmos com repositórios é utilizando o *git clone*. Com ele, podemos clonar um repositório já existente de um repositório remoto para o nosso desenvolvimento local. A partir do momento em que você realiza uma cópia ativa do repositório, as operações realizadas nele ficam armazenadas localmente até você enviá-las para o repositório remoto.

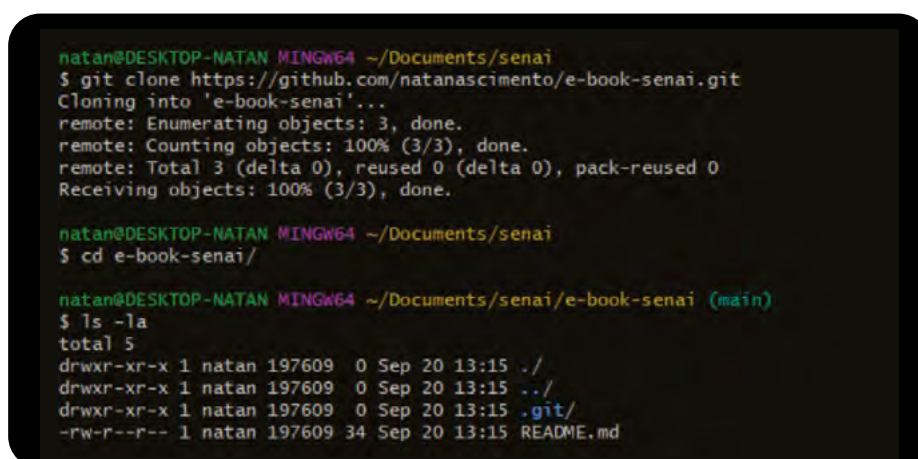
A terminal window with a dark background and light green text. The prompt is 'natan@DESKTOP-NATAN MINGW64 ~/Documents/senai'. The first command is '\$ git clone https://github.com/natanascimento/e-book-senai.git'. The output shows 'Cloning into 'e-book-senai'...', 'remote: Enumerating objects: 3, done.', 'remote: Counting objects: 100% (3/3), done.', 'remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0', and 'Receiving objects: 100% (3/3), done.'. The second command is '\$ cd e-book-senai/'. The third command is '\$ ls -la'. The output shows 'total 5', 'drwxr-xr-x 1 natan 197609 0 Sep 20 13:15 ./', 'drwxr-xr-x 1 natan 197609 0 Sep 20 13:15 ../', 'drwxr-xr-x 1 natan 197609 0 Sep 20 13:15 .git/', and '-rw-r--r-- 1 natan 197609 34 Sep 20 13:15 README.md'. The current directory is shown as '(main)'.

Figura 2 - Imagem do console de desenvolvimento ilustrando o clone do repositório remoto

Fonte: Do autor (2022)

Acima, foi realizada a criação de um repositório na nuvem (no GitHub – iremos falar sobre isso daqui a pouco), e, através do comando *git clone*, foi feita a cópia do repositório remoto para o dispositivo local utilizando comunicação *HTTPS*.

## git config

A função do *git config* é realizar a configuração do Git nos projetos em um nível global (que afetará todos os projetos presentes em seu dispositivo) ou local (que afetará apenas um único repositório).



Na sequência, apresentamos um exemplo de configuração do nome do usuário e do e-mail:

```
git config --global user.name "Fulano"
```

Configuração do nome do usuário

```
git config --global user.email "dominio@provedor.com"
```

Configuração do e-mail do usuário

## Analizando o repositório

Em um repositório Git, temos três estados nos quais o nosso código pode estar. Esses estados são diretórios ou ramificações que o Git controla para nós. Os estados são o *working directory*, o *staging area* e o *git directory* (repositório).

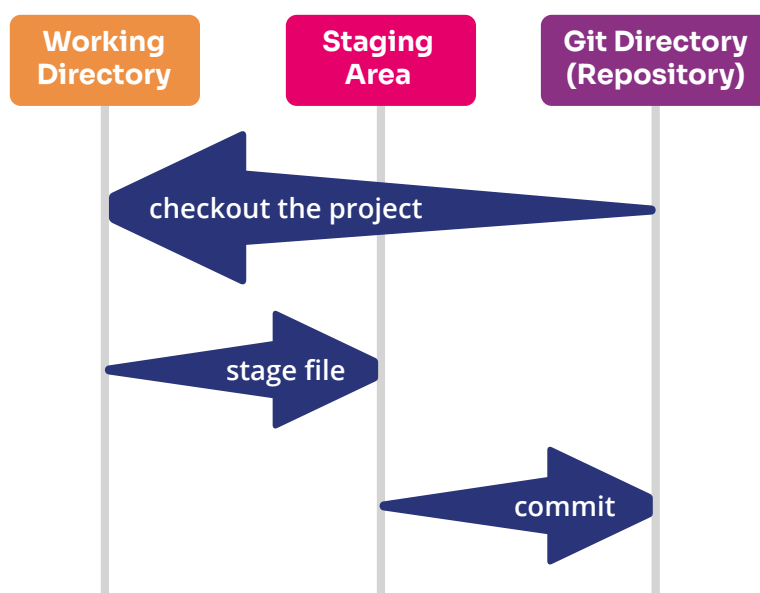


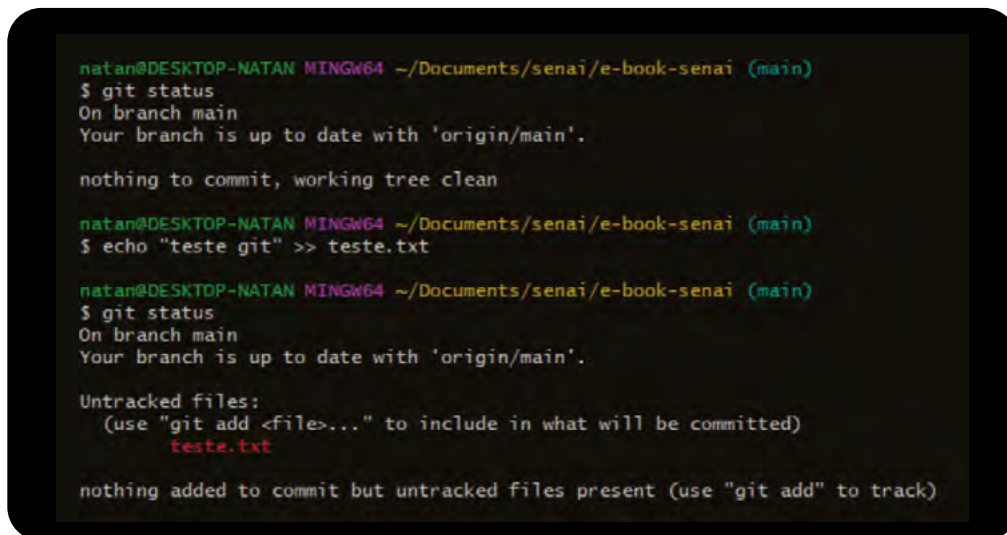
Figura 3 - Três estados em que o código pode estar em um repositório Git

Fonte: Adaptado de Chacon e Straub (2022)

Seguindo a imagem apresentada, o *working directory* é o estado atual do código no nosso repositório. Já o *staging area* é basicamente uma área de preparação antes de fechar o *commit*. Mais adiante, discutiremos um pouco mais sobre como levar os arquivos para o estado de *staging*. Por fim, o *git directory* é onde o Git armazena os metadados do projeto, é a parte mais importante do Git, pois se trata da parte que é copiada quando se clona um projeto de um outro local, ou seja, funciona como um banco de objetos do Git.

## git status

Para analisarmos o estado no qual o nosso código está dentro do repositório, podemos utilizar o comando *git status*. Dessa forma, o valor que será retornado é onde o seu arquivo está entre todos os estados anteriormente citados. Veja um exemplo na imagem a seguir:



```
natan@DESKTOP-NATAN MINGW64 ~/Documents/senai/e-book-senai (main)
$ git status
On branch main
Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean

natan@DESKTOP-NATAN MINGW64 ~/Documents/senai/e-book-senai (main)
$ echo "teste git" >> teste.txt

natan@DESKTOP-NATAN MINGW64 ~/Documents/senai/e-book-senai (main)
$ git status
On branch main
Your branch is up to date with 'origin/main'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)
       teste.txt

nothing added to commit but untracked files present (use "git add" to track)
```

Figura 4 - Imagem do console de desenvolvimento ilustrando o estado do repositório antes e após a criação de um novo arquivo

Fonte: Do autor (2022)

Realizado o primeiro *git status*, foi possível perceber que estava sem nenhuma alteração local, isto é, o repositório local está igual ao repositório remoto. Foi, então, criado um novo arquivo, e o *git status* foi executado novamente. Dessa forma, podemos ver que existe um arquivo *untracked*, ou seja, um arquivo que está no *working directory*. Mais adiante, abordaremos todos os estados citados anteriormente.

## Salvar alterações

Quando estamos trabalhando em um projeto, realizamos diversas alterações. Quanto a essas alterações, é necessário salvá-las em nosso repositório. Vamos falar um pouco mais sobre como salvar as alterações de forma local e, até mesmo, visualizar um comparativo entre as alterações realizadas.

## git add

Utilizamos o comando *git add* para levar um arquivo do *working directory* para o *staging area*. Assim, na nossa linha de comando, teremos a informação que N arquivos estarão prontos para ser enviados ao *git directory*, conforme imagem a seguir.

```
natan@DESKTOP-NATAN MINGW64 ~/Documents/senai/e-book-senai (main)
$ git status
On branch main
Your branch is up to date with 'origin/main'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)
       teste.txt

nothing added to commit but untracked files present (use "git add" to track)
9
natan@DESKTOP-NATAN MINGW64 ~/Documents/senai/e-book-senai (main)
$ git add teste.txt
warning: LF will be replaced by CRLF in teste.txt.
The file will have its original line endings in your working directory

natan@DESKTOP-NATAN MINGW64 ~/Documents/senai/e-book-senai (main)
$ git status
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
       new file:   teste.txt
```

Figura 5 - Imagem do console de desenvolvimento ilustrando o processo de *staging* de um arquivo

Fonte: Do autor (2022)

## git commit

O *commit* é o responsável por levar o arquivo do estado de *staging* para o *git directory*. Podemos utilizar alguns parâmetros, como "-m", para informar que queremos escrever uma mensagem ao nosso *commit*. É importante adicionar uma mensagem descritiva no *commit*, pois, dessa forma, caso um dia exista a necessidade de reverter a alteração realizada, podemos analisar, através da mensagem do *commit*, a qual versão queremos retornar.



```
natan@DESKTOP-NATAN MINGW64 ~/Documents/senai/e-book-senai (main)
$ git status
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   teste.txt

natan@DESKTOP-NATAN MINGW64 ~/Documents/senai/e-book-senai (main)
$ git commit -m 'adicionando arquivo de teste'
[main ef41775] adicionando arquivo de teste
1 file changed, 1 insertion(+)
create mode 100644 teste.txt

natan@DESKTOP-NATAN MINGW64 ~/Documents/senai/e-book-senai (main)
$ git status
On branch main
Your branch is ahead of 'origin/main' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
```

Figura 6 - Imagem do console de desenvolvimento ilustrando o processo de *commit* de um novo arquivo

Fonte: Do autor (2022)

Conforme a imagem apresentada, ao executarmos o status após o *commit*, a mensagem retornada é a de que a sua *branch* local está um *commit* à frente da mesma *branch* no repositório remoto. Mais adiante, analisaremos o que isso significa.

## git diff

O *diff* é um comando que, quando executado, realiza uma comparação entre fontes de dados. Vamos para um exemplo da utilização desse comando em nosso repositório: imagine que você tenha criado um arquivo novo em seu projeto e levado esse arquivo para a *staging area*, e que, momentos depois, tenha feito uma alteração nesse arquivo. A partir do momento em que a alteração for realizada, ao executar o *diff*, você terá o que foi realizado no arquivo, como ilustra a imagem a seguir.

```

natan@DESKTOP-NATAN MINGW64 ~/Documents/senai/e-book-senai (main)
$ echo "teste git 2" >> teste.txt

natan@DESKTOP-NATAN MINGW64 ~/Documents/senai/e-book-senai (main)
$ git status
On branch main
Your branch is ahead of 'origin/main' by 1 commit.
(use "git push" to publish your local commits)

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   teste.txt

no changes added to commit (use "git add" and/or "git commit -a")

natan@DESKTOP-NATAN MINGW64 ~/Documents/senai/e-book-senai (main)
$ git diff
warning: LF will be replaced by CRLF in teste.txt.
The file will have its original line endings in your working directory
diff --git a/teste.txt b/teste.txt
index bb26e4a..e064740 100644
--- a/teste.txt
+++ b/teste.txt
@@ -1,3 @@
 teste git
+teste git 2
+teste git 2

```

Figura 7 - Imagem do console de desenvolvimento ilustrando como verificar o que foi alterado nos arquivos através do comando *git diff*

Fonte: Do autor (2022)

O status “*modified*” mostra que uma modificação foi realizada naquele arquivo. Ao executarmos o *git diff*, poderemos ver que foram adicionadas duas novas linhas ao arquivo “teste.txt”.

## Sincronizar repositório

Enquanto você estiver desenvolvendo seu projeto, sempre tenha em mente que você precisará sincronizar as suas alterações, principalmente em projetos colaborativos, em que, muitas das vezes, há cinco ou até dez desenvolvedores trabalhando em um mesmo repositório. Nessa situação, utilizaremos alguns comandos, que serão nossos aliados em projetos colaborativos.

### *git remote*

Através do *git remote*, passamos a criar, ver e excluir conexões com repositórios remotos. As conexões são realizadas do repositório local para o repositório remoto. Dessa forma, conseguimos conceder acesso em tempo real a outros repositórios.

```
natan@DESKTOP-NATAN MINGW64 ~/Documents/senai/git (master)
$ git remote -v

natan@DESKTOP-NATAN MINGW64 ~/Documents/senai/git (master)
$ git remote add origin https://github.com/natanascimento/e-book-senai.git

natan@DESKTOP-NATAN MINGW64 ~/Documents/senai/git (master)
$ git remote -v
origin  https://github.com/natanascimento/e-book-senai.git (fetch)
origin  https://github.com/natanascimento/e-book-senai.git (push)
```

Figura 8 - Imagem do console de desenvolvimento ilustrando uma conexão do repositório local com o repositório remoto

Fonte: Do autor (2022)

O comando "*git remote -v*" realiza a listagem de todas as conexões criadas em seu repositório. Além da listagem, conseguimos renomear as conexões com o "*git remote rename <antigo\_nome> <nome\_atual>*" e removê-las com o "*git remote rm <nome\_da\_conexão>*". O nome da conexão é passado no momento da adição do "marcador". No nosso exemplo apresentado, utilizamos o "*git remote add origin <url>*", sendo *origin* o nome da nossa conexão.

## git push

O *git push* é utilizado para publicar as modificações do *git directory* local a um repositório remoto. A partir do momento em que o fluxo de trabalho é feito, ou seja, quando os arquivos são carregados para a *staging area* e depois para o estado do *git directory*, conforme informado anteriormente, você terá a possibilidade de enviar os arquivos para um repositório central. Esse processo é utilizado para compartilhar as modificações realizadas por você com os demais membros da equipe.

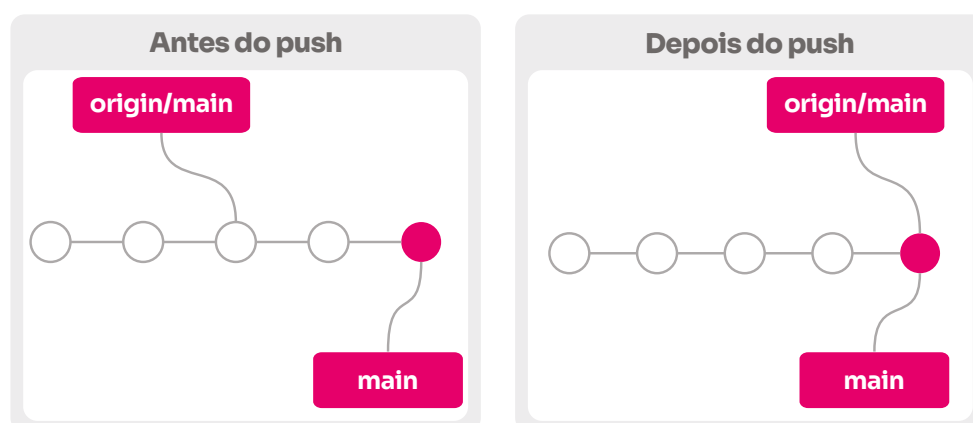


Figura 9 - Git push

Fonte: do Autor (2022)

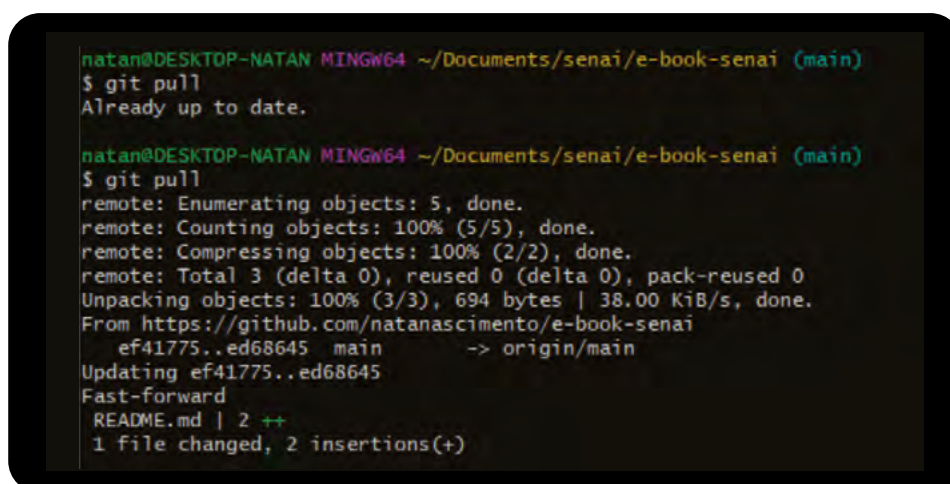




Na imagem apresentada, observe que, através do comando *push*, conseguimos sincronizar os arquivos que estavam no repositório local, com origem na ramificação “*main*”.

## git pull

O comando “*git pull*” é utilizado para ir a um repositório remoto, buscar e baixar o conteúdo. Ele é responsável por realizar a atualização imediata do repositório local para que fique 100% sincronizado com o repositório remoto.



```
natan@DESKTOP-NATAN MINGW64 ~/Documents/senai/e-book-senai (main)
$ git pull
Already up to date.

natan@DESKTOP-NATAN MINGW64 ~/Documents/senai/e-book-senai (main)
$ git pull
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 694 bytes | 38.00 KiB/s, done.
From https://github.com/natanascimento/e-book-senai
   ef41775..ed68645  main      -> origin/main
Updating ef41775..ed68645
Fast-forward
 README.md | 2 ++
 1 file changed, 2 insertions(+)
```

Figura 10 - Imagem do console de desenvolvimento ilustrando o processo de comunicação e download das alterações presentes no repositório remoto

Fonte: Do autor (2022)

Neste exemplo da imagem, é realizado um primeiro *git pull* e, como o repositório local está igual ao repositório remoto, o *git* informa que o repositório na ramificação atual (*main*) já está atualizado. Após essa primeira execução, foi realizada uma alteração no repositório remoto, mais especificamente no arquivo “*README.md*”. Para retornarmos essas modificações, precisamos executar o *git pull* novamente, o qual, então, é invocado, executado e realiza o download das alterações. A mensagem de retorno informa que um arquivo foi alterado, contendo nele duas novas inserções.

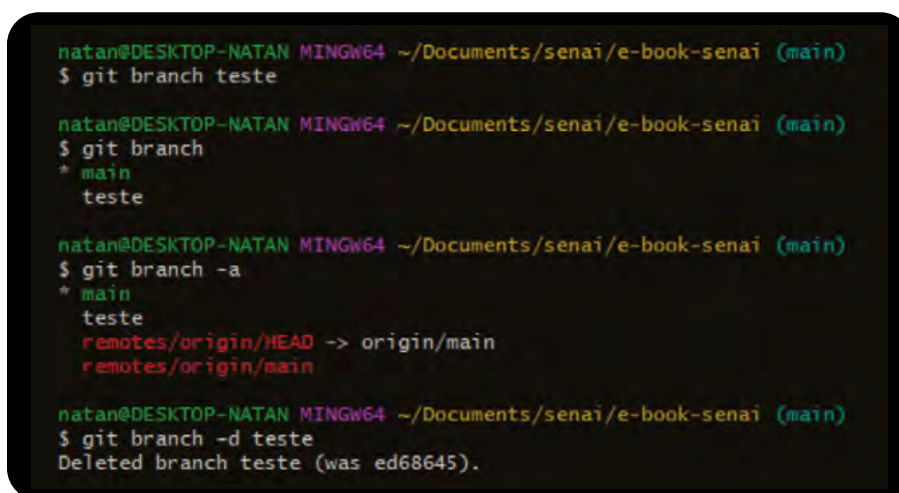
# Trabalhando com *branches* (ramificações)

Uma *branch* ou ramificação é um recurso que está presente na maioria dos sistemas de controle de versão. Diariamente, utilizamos *branches* como aliadas no processo de desenvolvimento, quando queremos solucionar um problema ou até adicionar novas funcionalidades ao sistema. Basta criarmos uma nova *branch* e acoplar o código. Ao final desse processo, vamos mesclar o código com a base do código principal.

## *git branch*

*Git branch* é o “apelido” para o comando “git branch --list”. Quando utilizamos o “git branch” sem nenhum parâmetro, temos a possibilidade de listar todas as *branches* disponíveis em nosso repositório. Podemos utilizar o comando para a criação de *branches*, apenas com o “git branch <nome\_da\_branch>”. Dessa forma, o Git irá criar uma ramificação da *branch* onde você está. Para remover uma *branch* de forma segura, utilizamos o “git branch -d <nome\_da\_branch>”, porém, caso queira forçar a exclusão, utilize o comando “git branch -D <nome\_da\_branch>”.

A seguir, temos um exemplo do uso do comando para a construção, listagem e remoção de *branches*:



```
natan@DESKTOP-NATAN MINGW64 ~/Documents/senai/e-book-senai (main)
$ git branch teste

natan@DESKTOP-NATAN MINGW64 ~/Documents/senai/e-book-senai (main)
$ git branch
* main
  teste

natan@DESKTOP-NATAN MINGW64 ~/Documents/senai/e-book-senai (main)
$ git branch -a
* main
  teste
remotes/origin/HEAD -> origin/main
remotes/origin/main

natan@DESKTOP-NATAN MINGW64 ~/Documents/senai/e-book-senai (main)
$ git branch -d teste
Deleted branch teste (was ed68645).
```

Figura 11 - Imagem do console de desenvolvimento ilustrando o processo de visualização das ramificações do repositório e remoção de uma ramificação

Fonte: Do autor (2022)

## git checkout

O comando *git checkout* é utilizado para alternar entre as versões no seu repositório. Além de realizar a verificação das *branches*, é uma forma de sair de uma *branch* e ir para outra. Utilizando o *checkout*, conseguimos navegar em diferentes ramificações criadas anteriormente, além de ser uma maneira através da qual conseguimos escolher em qual linha de desenvolvimento queremos ficar. Imagine, por exemplo, que esteja trabalhando em uma versão do código e que precise realizar uma alteração em outra versão. Basta utilizar o *git checkout* para sair da sua *branch* atual e ir para a *branch* destino. Porém, vale lembrar que, para você sair da sua *branch* atual, você não pode ter modificações nos arquivos da sua ramificação atual em *working directory*, pois as alterações não serão salvas no momento da navegação entre as *branches*.

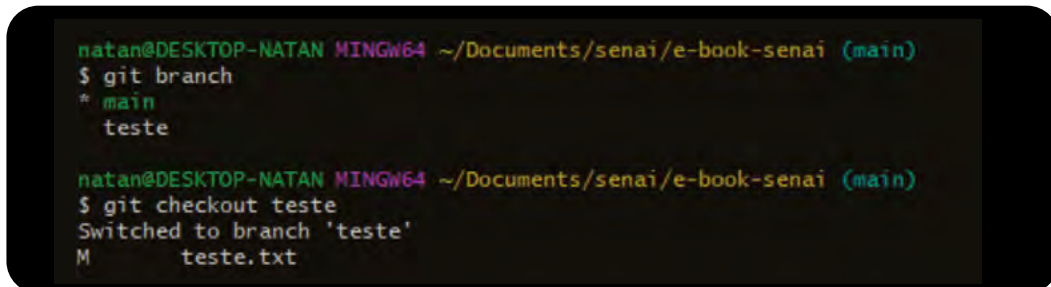
A terminal window with a dark background and light green text. The prompt is 'natan@DESKTOP-NATAN MINGW64 ~/Documents/senai/e-book-senai (main)'. The first command is '\$ git branch', which outputs '\* main' and 'teste'. The second command is '\$ git checkout teste', which outputs 'Switched to branch 'teste'' and 'M teste.txt'.

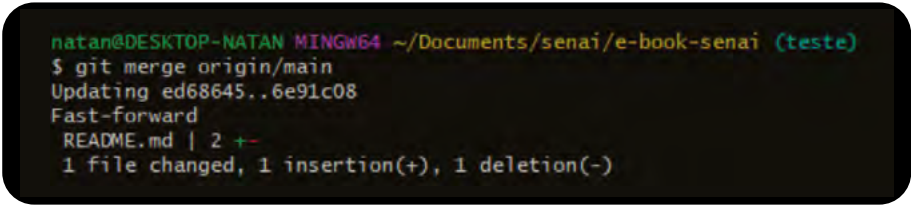
Figura 12 - Imagem do console de desenvolvimento ilustrando o processo de visualização das ramificações do repositório e acessando uma dessas ramificações

Fonte: Do autor (2022)

O exemplo dado mostra como realizar o processo de navegação entre as *branches* utilizando o *git checkout*. Veja que, primeiro, foi validado qual *branch* estava disponível localmente e, depois, a *branch* foi acessada. Com o *git checkout*, podemos realizar a construção de *branches* utilizando o “*git checkout -b <nome\_da\_branch>*”. Essa é uma forma mais interessante, pois faz uma junção do “*git branch <nome\_da\_branch>*” com o “*git checkout <nome\_da\_branch>*”. Com isso, você terá uma maior agilidade no momento de criação das suas ramificações.

## git merge

*Merge* ou mesclagem é uma forma de unificar um histórico entre duas ramificações. O comando *git merge* combina uma sequência de *commits* realizados construindo um histórico unificado.



```
natan@DESKTOP-NATAN MINGW64 ~/Documents/senai/e-book-senai (teste)
$ git merge origin/main
Updating ed68645..6e91c08
Fast-forward
 README.md | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Figura 13 - Imagem do console de desenvolvimento ilustrando o processo de *merge* da ramificação corrente (teste) com a ramificação principal (*main*)

Fonte: Do autor (2022)

Este exemplo ilustra a utilização do *merge* na prática. Basicamente, criou-se uma nova versão de código na ramificação principal (*main*), e precisamos coletar essa versão e mesclá-la com o que temos na ramificação atual (teste) no nosso ambiente local. Para tanto, utilizamos o comando *merge*, que irá até a ramificação selecionada e mesclará com o que temos na nossa ramificação. Note que, no momento da invocação do comando *merge*, foi utilizada o "*origin/<nome\_da\_branch>*". Isso se dá porque queremos que retornem as alterações que estão lá no nosso repositório remoto, ou seja, ao executarmos o comando visto neste exemplo, será aberta uma conexão com o repositório remoto e serão mescladas as alterações da última versão da *branch* selecionada com o que temos localmente. Dessa forma, garantimos que estamos atualizados com a ramificação de desejamos.

Esse processo é muito comum quando estamos trabalhando com um repositório com vários desenvolvedores plugados, ou seja, sempre haverá novas versões e você precisa trazer isso para o seu ambiente local antes de levar ao repositório remoto – isso evitará conflitos de versões quando você realizar "*merges*" no futuro.

## GITHUB

Você já deve ter ouvido falar sobre o tão famoso GitHub e sobre como ele é essencial para engenheiros de software atualmente. Bem, o GitHub é um serviço baseado em nuvem que realiza a hospedagem do Git (sistema de controle de versão). Ele é também uma excelente ferramenta que permite que os desenvolvedores colaborem e realizem mudanças nos projetos enquanto os registros são armazenados durante todo o progresso do projeto. Por ser uma das ferramentas mais populares atualmente, diversos repositórios de projetos *open source* são hospedados no GitHub, atualmente com cerca de 100 milhões de repositórios publicados na plataforma.

Para saber mais sobre o GitHub e sua atual marca de 100 milhões de repositórios, você pode acessar a notícia através do link ou do código QR a seguir:

<https://venturebeat.com/ai/github-passes-100-million-repositories/>



Na plataforma, você poderá criar repositórios, *branches*, gerenciar projetos, configurar pipelines de integração e entrega contínuas (GitHub Actions) e muito mais.



## METODOLOGIAS ÁGEIS

Visando ao aumento de produtividade, empresas adotam metodologias que auxiliam no processo de desenvolvimento como um todo. As metodologias ágeis são formas de potencializar e acelerar a entrega em projetos, sejam eles de software ou não. A ideia é ter um ambiente onde as pessoas possam errar rápido e corrigir mais rápido ainda, sempre visando entregar valor para o cliente.

É um meio em que a colaboração fala mais alto e, com isso, acaba tornando as entregas mais rápidas, com um alto valor agregado. Esse processo dá ao cliente a oportunidade de vislumbrar os resultados antecipadamente, garantindo uma ótima conexão entre cliente e empresa.



# Tradicional x Ágil

Os métodos ágeis surgiram como oposição aos métodos tradicionais de gerenciamento de projetos, como o modelo cascata. Na época da concepção dos métodos ágeis, a indústria de desenvolvimento de software estava passando por uma série de dificuldades no processo de gerenciamento dos projetos. Isso se dava pelo fato de o modelo cascata ser um modelo sequencial no qual os processos devem seguir uma sequência lógica definida, não sendo, portanto, uma abordagem que permite flexibilidade aos times. Sendo assim, tal modelo apresenta diversas limitações, visto que, para reparar algo que foi feito durante o processo, o esforço acaba sendo muito maior, pois deve-se aguardar a sequência de atividades finalizar para que se inicie tudo novamente.

Com isso, algumas pessoas influentes na comunidade de software, na época, se reuniram e discutiram sobre a construção de um “Manifesto para Desenvolvimento Ágil de Software”. Em função disso, através de diversas discussões, criou-se, no ano de 2001, o Manifesto Ágil. Os métodos ágeis têm em sua base o Manifesto Ágil, que foi construído contendo quatro valores e doze princípios, sendo eles:

• • • • •

Valorizar:

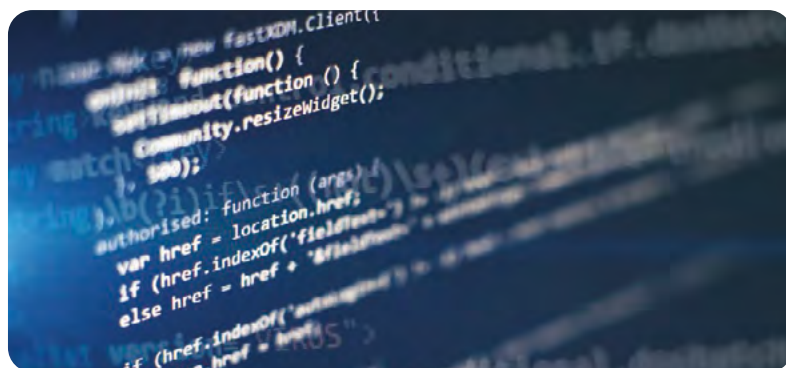
- Indivíduos e interações, mais do que processos e ferramentas.
  - Software em funcionamento, mais do que documentação abrangente.
  - Colaboração com o cliente, mais do que negociação de contratos
  - Responder a mudanças, mais do que seguir um plano.
- • • • •

Esses são os quatro valores levantados no Manifesto Ágil. Já os doze princípios são:

- • • • •
- Nossa maior prioridade é satisfazer o cliente, através da entrega contínua e adiantada de software com valor agregado.
  - Mudanças nos requisitos são bem-vindas, mesmo tardiamente, no desenvolvimento. Processos ágeis tiram vantagem das mudanças visando a vantagem competitiva para o cliente.
  - Entregar frequentemente software funcionando, de poucas semanas a poucos meses, com preferência à menor escala de tempo.
  - Pessoas de negócio e desenvolvedores devem trabalhar diariamente em conjunto por todo o projeto.
- • • • •

- Construir projetos em torno de indivíduos motivados. Providenciar o ambiente e o suporte necessários e confiar neles para fazer o trabalho.
- O método mais eficiente e eficaz de transmitir informações para e entre uma equipe de desenvolvimento é através de conversa face a face.
- Software funcionando é a medida primária de progresso.
- Os processos ágeis promovem desenvolvimento sustentável. Os stakeholders, desenvolvedores e usuários devem ser capazes de manter um ritmo constante indefinidamente.
- Contínua atenção à excelência técnica e bom design aumenta a agilidade.
- Simplicidade – a arte de maximizar a quantidade de trabalho não realizado é essencial.
- As melhores arquiteturas, requisitos e designs emergem de equipes auto-organizáveis.
- Em intervalos regulares, a equipe reflete sobre como se tornar mais eficaz e então refina e ajusta seu comportamento de acordo.

Os autores do Manifesto Ágil entendem que todos os itens listados acima possuem valor inerente e que, nos valores, é importante priorizar à esquerda (da vírgula) em detrimento dos itens da direita. O Manifesto Ágil promoveu uma revolução no processo de gerenciamento de projetos, pois permite a execução de desenvolvimentos simultâneos e contínuos, sempre com um foco em gerar valor agregado para o cliente final.



## Scrum

O Scrum é um framework ágil que é a base de muitos processos de desenvolvimento de software atualmente e vem ganhando muito reconhecimento no mercado. Já está sendo utilizado em diversas áreas, como construção civil, arquitetura, educação, entre outras. Atualmente, é um dos “frameworks ágeis” mais utilizados, focado em melhorar o processo

de desenvolvimento através de uma abordagem ágil e direcionado para o gerenciamento colaborativo em equipes de pequeno porte e multifuncionais.

O Scrum ajuda pessoas, times e organizações a gerar valor agregado ao cliente final por meio de soluções adaptativas para problemas complexos. Vários processos, técnicas e métodos podem ser empregados com o framework, pois ele é adaptável, iterativo, rápido e flexível.

O Scrum possui pilares empíricos, conforme descritos a seguir:

- **Transparência:**

Transparência dos processos, requisitos de entrega e status. Dessa forma, os trabalhos devem ser visíveis tanto para quem os executa quanto para quem os recebe.

- **Inspeção:**

Inspeção constante de tudo o que está sendo desenvolvido. O progresso em direção às metas acordadas deve ser inspecionado com frequência e diligência para detectar variações ou problemas potencialmente indesejáveis.

- **Adaptação:**

Adaptação, tanto do processo quanto da capacidade do produto de suportar mudanças. Caso exista algum aspecto que se desvie dos limites aceitáveis ou se o produto resultante for inaceitável, o ajuste deve ser feito o mais rápido possível para minimizar novos desvios.

## Papéis

No Scrum, existem três papéis fundamentais: O *Product Owner*, o *Scrum Master* e o time Scrum, com responsabilidades únicas sob as quais os participantes da equipe Scrum atuam. Esses papéis não são necessariamente um cargo, mas sim um conjunto mínimo de responsabilidades e prestação de contas, para ativar as entregas das equipes Scrum com eficiência.



## Time Scrum

A unidade fundamental do Scrum é um pequeno time de pessoas, chamado time Scrum. Em sua estrutura, temos o *Scrum Master*, o *Product Owner* e Desenvolvedores. Todos estão no time com um objetivo em comum, entregar valor ao cliente.

O time Scrum é um time multidisciplinar, em que, muitas das vezes, há pessoas com diferentes habilidades que ajudam a potencializar a entrega, além de serem multifuncionais, o que nos leva a entender que o time possui capacidade e habilidades necessárias para gerar valor a cada nova *Sprint*.

O time Scrum é um time pequeno, focado em permanecer ágil o suficiente para concluir suas atividades durante uma *Sprint*; normalmente, não passa de dez pessoas por time. Com o passar do tempo, os criadores do Scrum identificaram que times menores se comunicam melhor e são mais produtivos.

O trabalho em união e colaboração são a base desse time, sempre estão com o foco em melhoria contínua e visam trabalhar juntos para completar com sucesso uma *Sprint*.



## Product Owner

O *Product Owner* (ou PO) é o responsável por maximizar o valor do produto em que o time Scrum está trabalhando. O PO está sempre em contato com o cliente, a fim de entender suas necessidades e dores. O *Product Owner* também é responsável por todo o gerenciamento do *Product Backlog*, que inclui desenvolver e comunicar explicitamente a meta do produto, definir e priorizar os itens de *Backlog* e garantir que o *Backlog* seja transparente, visível e compreensível.

O *Product Owner* é uma pessoa que está ali para compreender plenamente as preocupações dos clientes ou stakeholders com os assuntos que envolvem o negócio, qualidade, mudanças no produto e análise de aspectos relacionados a riscos no projeto. É uma pessoa que está sempre procurando as melhores maneiras de realizar o incremento do produto de forma sustentável.





## Scrum Master

O *Scrum Master* é o responsável por estabelecer o Scrum no time. Ele é considerado o líder do time Scrum e é a pessoa que defende o projeto e o time, além de ser o responsável pela eficácia do time Scrum. O *Scrum Master* é quem dissemina o conhecimento relacionado ao Scrum e assegura o cumprimento dos princípios ágeis, além de fornecer suporte à equipe durante o planejamento e a execução da *Sprint*.

O *Scrum Master* tem um papel fundamental, pois ele ajuda o time Scrum a se concentrar na criação do produto de forma incremental, visando gerar o mais alto valor para o cliente final, treinando os membros do time em autogerenciamento e cross-funcionalidade. Está sempre de olho no painel Scrum, e, quando identifica algo que impede o time Scrum de prosseguir, ele prontamente vai atrás de como sanar esse impedimento, bem como de garantir que todas as cerimônias presentes no Scrum aconteçam de forma positiva, produtiva, e que sejam finalizadas conforme o horário definido.

## Cerimônias

No Scrum, existem alguns eventos que são uma oportunidade formal de inspecionar e adaptar os artefatos do Scrum. Esses eventos ou cerimônias são elaborados especificamente para projetar a transparência necessária para todos os integrantes do time Scrum e, até mesmo, para quem não é do time. São eventos que criam uma recorrência de encontros e minimizam a necessidade de reuniões desnecessárias durante uma *Sprint*.



## Sprint Planning

A *Sprint Planning* é um marco inicial de uma nova *Sprint*. É um evento onde o time se reúne com o intuito de planejar o que será executado durante a próxima *Sprint*, e o plano é definido a partir da colaboração de todos os envolvidos no time Scrum. Nessa cerimônia, o *Product Owner* garante que os participantes estejam preparados para discutir os itens do *Product Backlog* e define uma ordem de priorização para a discussão dos itens.

Na *Sprint Planning*, são abordados alguns tópicos, como: “Por que esta *Sprint* é valiosa?”, “O que pode ser feito nesta *Sprint*?”, “Como o trabalho será realizado?”. A partir desses tópicos, o time trabalha em conjunto para definir quais itens vão para a *Sprint* atual, qual o risco de levar determinado card e qual a capacidade de cards que podem ser levadas para essa *Sprint*. Essa é uma métrica interessante, pois a partir de *Sprints* passadas, o time consegue definir a capacidade futura de entrega.

Todo o time colabora na definição da meta da *Sprint* (itens que serão desenvolvidos). Durante esse evento, é desenvolvido um artefato muito importante, que é o *Sprint Backlog* – vamos falar um pouco mais sobre ele daqui a pouco.

## Sprint

No Scrum, as *Sprints* são onde as ideias são transformadas em valor, onde é realmente desenvolvido o que foi planejado; funcionam como o coração do Scrum. Esse evento possui uma duração fixa, que pode variar de uma semana a quatro semanas. Essa duração é definida pelo time e, através do feedback, pode ser revista a qualquer momento.

Todo o trabalho necessário para atingir a meta do produto, incluindo as cerimônias do Scrum, acontece dentro da *Sprint*. Durante as *Sprints*, mudanças podem acontecer, porém, deve-se tomar muito cuidado para não afetar a meta da *Sprint* como um todo.

Nessa etapa, é garantida a inspeção e adaptação do progresso em direção à meta do produto. É necessário que a *Sprint* tenha um tempo pré-definido, pois se uma *Sprint* tiver uma jornada muito longa, a capacidade de aprendizagem pode ficar um pouco menor, e a complexidade e o risco podem aumentar. Por isso, quanto mais curtas as *Sprints*, melhor, pois, dessa forma, podemos gerar mais ciclos de aprendizado e limitar o risco de custo e esforço do time em um período menor.

Uma *Sprint* pode ser cancelada se sua meta se tornar obsoleta, ou seja, caso o time identifique que aquelas tarefas planejadas não fazem mais sentido de serem desenvolvidas, a *Sprint* pode ser cancelada.



## Daily Scrum

A *Daily Scrum* ou *Daily* é uma reunião cujo objetivo é ter uma recorrência diária, visando inspecionar o progresso em direção a meta da *Sprint* e adaptar o *Sprint Backlog* caso exista a necessidade. A *Daily* é um momento em que o time Scrum se reúne durante um espaço de tempo de 15 minutos. A ideia é reportar o progresso das atividades que estão sendo desenvolvidas e disseminar conhecimento sobre o que foi realizado no dia anterior, além de identificar impedimentos e priorizar as tarefas do dia.

O *Daily Scrum* é uma sessão em que muitos times costumam responder três perguntas principais:

Essas perguntas servem como guia para que os participantes da reunião possam identificar a evolução das atividades que estão sendo desenvolvidas.

• • • • •

- O que fiz ontem?
- O que irei fazer hoje?
- Existe algum impedimento que comprometa a meta da *Sprint*?

• • • • •

A *Daily* é um momento que contribui para a melhoria da comunicação no time, identifica os impedimentos que o time tem, promove a rápida tomada de decisão e, conseqüentemente, elimina a necessidade de outras reuniões.

A *Daily* não é um único momento em que o time pode ajustar o plano. Muitas vezes, os integrantes do time se reúnem ao longo do dia para discussões mais detalhadas sobre replanejamento de trabalho etc. A *Daily* é um momento mais focado em passar o estado e progresso das atividades.



## Review Meeting

A *Review Meeting* ou *Review* é uma cerimônia que tem como objetivo inspecionar o resultado da *Sprint* e determinar adaptações futuras. Nesse evento, o time se reúne e analisa os resultados do trabalho desenvolvido para os principais stakeholders, além do progresso da meta do produto.

Durante essa cerimônia, o time costuma analisar métricas relacionadas às entregas e discutir determinadas situações que aconteceram durante a *Sprint*, por exemplo reportar o motivo do atraso de uma determinada atividade e descrever qual foi o aprendizado obtido a partir dessa situação.

A ideia principal é que seja apenas uma apresentação com um tempo limitado para que todos possam tecer análises em conjunto – o ideal é que dure no máximo uma hora.

## Retrospective Sprint

A *Retrospective Sprint* ou *Retro*, como muitos a chamam, é uma reunião com o foco em planejar maneiras de aumentar a qualidade e a eficácia do time. Com isso, o time Scrum inspeciona como foi a última *Sprint* em relação aos indivíduos, interações, processos, ferramentas e eventos que aconteceram durante a reunião.

É um momento de aprendizado em que, a partir do feedback dos integrantes do time, podemos melhorar a experiência do time como um todo. Por exemplo, imaginemos que estamos em um time com cinco integrantes e que temos quatro feedbacks de “muitas atividades não entregues nesta *Sprint*”; o time como um todo analisa a situação e identifica que, para reduzir a quantidade de atividades não entregues, seria interessante aumentar o tamanho da *Sprint* ou, até mesmo, diminuir o tamanho das atividades.





É no momento da *Retro* que o time identifica as mudanças mais úteis que podem melhorar significativamente a eficácia e o desempenho. As melhorias mais impactantes são endereçadas o mais rápido possível e têm maior prioridade frente às outras menos impactantes. O time discute, também, dificuldades enfrentadas durante a Sprint e como esses problemas foram (ou não) solucionados.

## Artefatos

No Scrum, os artefatos são representados como trabalho ou valor. Eles são projetados para maximizar a transparência, tanto para o time quanto para os clientes. São meios de trazer uma maior facilidade de adaptação e capacidade de relatar o que está sendo feito. Cada artefato gerado pelo Scrum possui um objeto, um compromisso em comum, e visa garantir que ele forneça informações e aumente a transparência e o foco no progresso.

São gerados três grandes artefatos: o *Product Backlog*, o *Sprint Backlog* e o Incremento. Juntos, eles reforçam os valores do Scrum, tanto para o time Scrum quanto para os stakeholders.

## Product Backlog

O *Product Backlog*, ou *Backlog do Produto*, é uma lista ordenada contendo o que é necessário para o produto. É a única fonte de trabalho realizado pelo time Scrum. Os itens podem ser realizados pelo time Scrum a partir do momento em que estiverem preparados para a seleção na *Planning*.



No processo de construção do *Backlog*, existe uma etapa muito importante, que é o refinamento das atividades. Essa etapa consiste, basicamente, em realizar a quebra das atividades, incluir algumas definições adicionais aos itens de *Backlog* e deixar a atividade cada vez mais descritiva e de fácil entendimento.

O *Product Backlog* é uma organização de longo prazo que o *Product Owner* constrói para o time. Nele, temos tudo o que é necessário para o desenvolvimento do produto. O *Product Owner* coleta diversas informações com o cliente e identifica quais atividades são necessárias e quando elas podem ser desenvolvidas. Então, ele organiza a lista por prioridade – sempre quem está no topo tem maior prioridade frente às outras atividades.



## **Sprint Backlog**

O *Sprint Backlog* é composto por atividades que são desenvolvidas durante aquela *Sprint*, ou seja, para cada nova *Sprint* existe um novo *Sprint Backlog*. É a meta da *Sprint*, contendo itens retirados do *Product Backlog* selecionados para serem desenvolvidos naquela *Sprint*. O *Sprint Backlog* é desenvolvido pelo time durante a reunião de Planejamento, a *Planning*.

## **Entrega**

A Entrega, ou incremento do produto, é o último estágio da *Sprint*, pois, ao final de cada *Sprint*, novos incrementos são criados no produto, e valores são gerados para o cliente final. Dessa forma, cada incremento é adicionado a todos os incrementos anteriores, que, em conjunto, geram maior valor agregado ao cliente.

Como podemos ver, o Scrum é um framework muito completo para gerenciamento de projeto. O seu ciclo se baseia em:

- Construção do *Product Backlog*.
- *Sprint Planning Meeting* para definição do *Sprint Backlog*.
- *Sprint* com um tempo determinado para a entrega das atividades.
- *Daily Meeting*.
- *Sprint Review* + *Sprint Retrospective* ao final da *Sprint*.

## REFERÊNCIAS

AGHINA, W.; HANDSCOMB, C.; SALO, O.; THAKER, S. The impact of agility: How to shape your organization to compete. **McKinsey & Company**, Atlanta [GA], 25 maio 2021. Disponível em: <https://www.mckinsey.com/capabilities/people-and-organizational-performance/our-insights/the-impact-of-agility-how-to-shape-your-organization-to-compete>. Acesso em: 16 set. 2022.

CHACON, S.; STRAUB, B. **Pro Git**. 2. ed. Version 2.1.359-2-g27002dd, 2022-10-03. New York [NY]: Apress, 2014 ver. atual. 2022.

LOELIGER, J.; MCCULLOUGH, M. **Version Control with Git**: Powerful tools and techniques for collaborative software development. Sebastopol [CA]: O'Reilly Media, 2012.

SCHWABER, K.; SUTHERLAND, J. The 2020 Scrum Guide™. **ScrumGuides.org**, November 2020. Disponível em: <https://scrumguides.org/scrum-guide.html>. Acesso em: 16 set, 2022.



**Natan Nascimento Oliveira Matos**

Engenheiro de Dados Sênior na John Deere e Mentor  
Educatonal no LAB365 do SENAI/SC.

**SENAI** <LAB365>