

**INSTITUTO  
FEDERAL**  
São Paulo

PFDA1 - PRÁTICAS E FERRAMENTAS  
DE DESENVOLVIMENTO DE  
SOFTWARE

THIAGO J. INOCÊNCIO<sup>1</sup>

---

<sup>1</sup> Departamento de Informática e Turismo, Instituto Federal de Educação, Ciência e Tecnologia de São Paulo, São Paulo, Brasil

## SUMÁRIO

1	Terminal e Linha de Comando	3
1.1	Shell Script	3
1.2	Diretório	4
1.3	Comandos	5
2	Controle de versão	12
2.1	Conceitos básicos	12
2.2	Sistemas de controle de versão locais	12
2.3	Sistemas de controle de versão centralizados	12
2.4	Sistemas de controle de versão distribuídos	13
2.5	Introdução ao Git	13

## LISTA DE FIGURAS

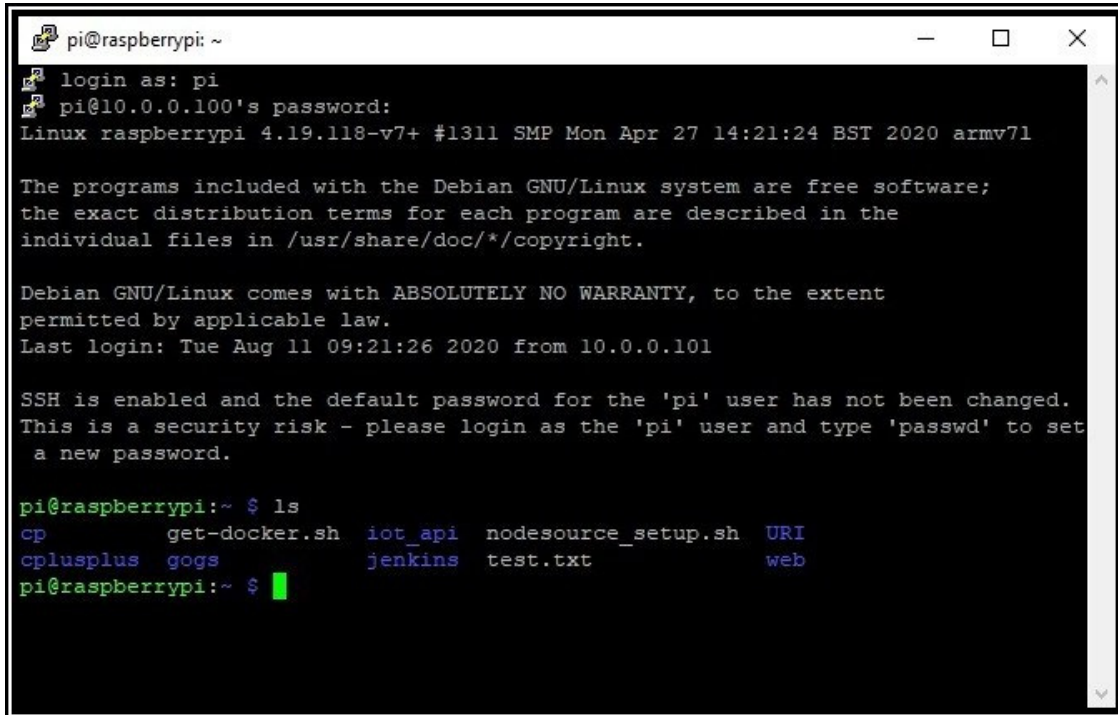
Figura 1	Ilustração de um terminal de linha de comando	3
Figura 2	Árvore de diretório de um sistema Linux/UNIX	4
Figura 3	Exemplo da execução do comando <code>pwd</code>	6
Figura 4	Exemplo da execução do comando <code>ls</code> para listagem de arquivos e pastas	6
Figura 5	Exemplo da execução do comando <code>ls</code> com parâmetro <code>-la</code>	8
Figura 6	Sistema de controle de versão local	12
Figura 7	Sistema de controle de versão centralizado	13
Figura 8	Sistema de controle de versão distribuídos	14
Figura 9	Árvore de trabalho, área de preparo e diretório GIT	14
Figura 10	Obtendo o link para clonagem de repositório no git	17
Figura 11	Modelo ilustrativo de fluxo de trabalho do git	18
Figura 12	Modelo ilustrativo do Git workflow	19
Figura 13	Modelo ilustrativo do Git workflow	20

## LISTA DE TABELAS

Tabela 1	Comandos para operações em diretórios	5
Tabela 2	Parâmetros úteis do comando <code>ls</code>	7
Tabela 3	Explicação do retorno do comando <code>ls</code> com parâmetro <code>-la</code>	7
Tabela 4	Explicação do retorno do comando <code>ls</code> com parâmetro <code>-la</code>	8

## 1 TERMINAL E LINHA DE COMANDO

Um terminal (linha de comando) é uma interface utilizada para se comunicar com o sistema operacional. A partir de um terminal, o usuário pode enviar comandos ao sistema operacional para que o computador realize alguma tarefa. A Figura 1 mostra um terminal.



```
pi@raspberrypi: ~  
login as: pi  
pi@10.0.0.100's password:  
Linux raspberrypi 4.19.118-v7+ #1311 SMP Mon Apr 27 14:21:24 BST 2020 armv7l  
  
The programs included with the Debian GNU/Linux system are free software;  
the exact distribution terms for each program are described in the  
individual files in /usr/share/doc/*/copyright.  
  
Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent  
permitted by applicable law.  
Last login: Tue Aug 11 09:21:26 2020 from 10.0.0.101  
  
SSH is enabled and the default password for the 'pi' user has not been changed.  
This is a security risk - please login as the 'pi' user and type 'passwd' to set  
a new password.  
  
pi@raspberrypi:~ $ ls  
cp          get-docker.sh  iot_api      nodesource_setup.sh  URI  
cplusplus   gogs          jenkins      test.txt             web  
pi@raspberrypi:~ $
```

Figura 1: Terminal de linha de comando.

Embora seja possível interagir com o sistema operacional por meio de uma interface gráfica<sup>1</sup>, muitas funcionalidades avançadas são mais facilmente realizadas utilizando um terminal. Por exemplo, configuração de rede, administração de contas de usuários etc. A Figura X mostra o mesmo diretório exibido na Figura 1 só que a partir de uma interface gráfica.

Podemos citar alguns exemplos de comandos que podem ser utilizados em terminais de sistemas UNIX/Linux, como: **date**, **whoami**, **pwd** etc. Nas próximas seções aprenderemos mais sobre comandos via terminal.

### 1.1 Shell Script

De acordo com Jargas [2] um script é uma lista de comandos para serem executados em sequência (um comando após o outro). Um roteiro predefinido de comandos e parâmetros.

Um Script Shell é um arquivo com uma lista de comandos que são executados em sequência e são interpretados por programas do tipo shell. No linux os interpretadores Shell mais conhecidos são o sh e o bash. No Windows podemos citar o PowerShell, entretanto nem todos os comandos que existem no Shell de sistemas UNIX são compatíveis com o PowerShell de sistemas Windows. O Algoritmo 1.1 mostra um script shell básico para exibir a mensagem 'Hello World':

<sup>1</sup> Ambiente visual com janelas

### Algorithm 1: Sheel Script básico para exibir a mensagem Hello World

```

1  #!/bin/bash
2  echo "Hello World"

```

Conforme visto no exemplo, a primeira linha de todo script sheel deve conter o local do sistema que está o interpretador sheel. No caso desse interpretador shell utilizado foi o *bash* que está localizado no diretório */bin/bash*.

## 1.2 Diretório

Diretório<sup>2</sup> é o local utilizado para armazenar conjuntos de arquivos para melhor organização e localização dentro de um sistema de arquivos. Em alguns sistemas como os Linux/UNIX os nomes de diretórios são *Case Sensitive*, ou seja, há diferenciação da nomenclatura de uma pasta chamada Documentos e DOCUMENTOS. A Figura 2 mostra um exemplo da árvore de diretórios de um sistema Linux/UNIX. O primeiro diretório / mostrado no topo da figura é conhecido como diretório raiz, a partir dele é iniciado uma hierárquica de subdiretórios (i.e bin, boot, dev etc) cada um com seus arquivos e subdiretórios. Como veremos na seção 1.3.2, existem diversos comandos que o usuário do sistema precisa saber para navegar e manipular os diretórios e seus arquivos.

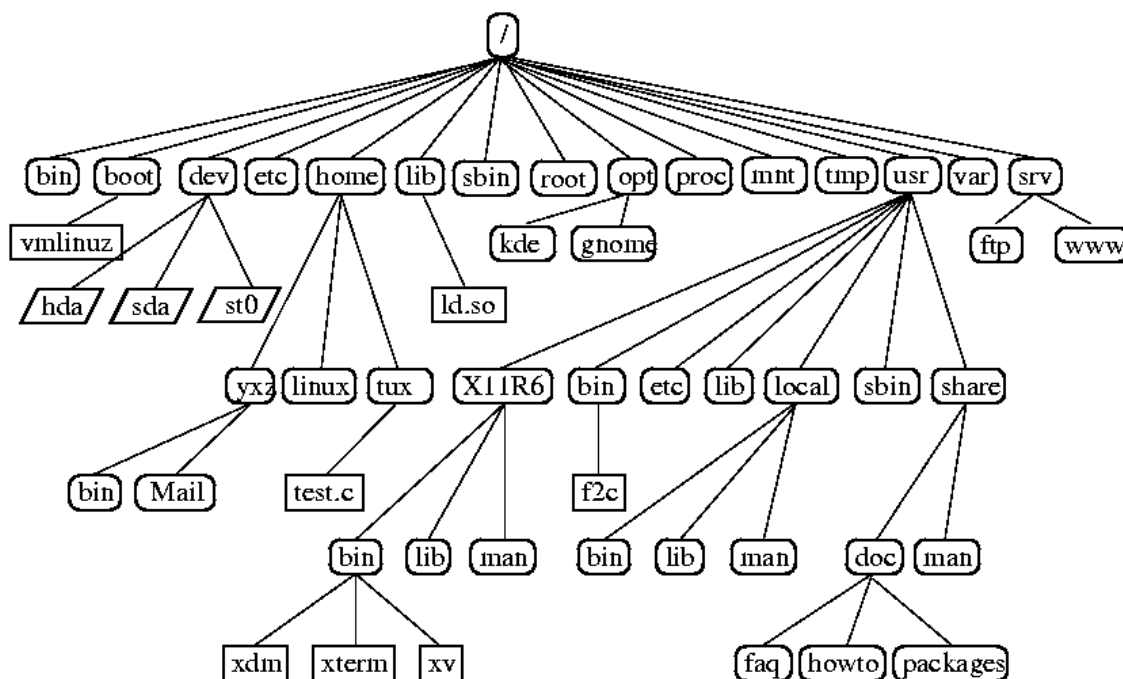


Figura 2: Árvore de diretório de um sistema Linux/UNIX [1]

Os caminhos na estrutura de diretórios são os caminhos que o usuário deve percorrer para chegar até um arquivo ou diretório desejado. Na figura 2 que mostra a hierarquia de diretórios de um sistema Linux/UNIX podemos chegar até o diretório *home/tux* fazendo o caminho que começa no diretório raiz / e vai até */home/linux*. Quando é feito um caminho partindo-se da origem (/) chamamos de um caminho (path) absoluto. Por exemplo, o caminho */home/tux* é um caminho absoluto que parte do diretório raiz até o diretório */home/tux*.

Caso o caminho não inicie da raiz (/) e sim de outro diretório chamamos de caminho relativo, pois o caminho é relativo a um determinado diretório. Por exemplo, se estivermos no diretório */usr/local* e

<sup>2</sup> De acordo com [3] o termo diretório é sinônimo de pasta

precisamos acessar o diretório `/usr/local/bin`, pelo caminho absoluto podemos acessar a pasta usando `cd /usr/local/bin`, no entanto, como estamos a um diretório `bin` na hierarquia de diretórios, tudo que precisamos fazer é utilizar o caminho relativo do diretório atual (local) e digitar `cd bin`.

### 1.3 Comandos

Comandos são muito utilizados em ambientes de execução de linha de comando. Como vimos na seção 1.1 é possível criar arquivos (scripts) contendo listagens de comandos para serem executados pelo sistema. Essas listagens de comandos são utilizadas pelos administradores de sistemas para executar tarefas repetitivas de maneira automatizada.

Na próxima seção, apresentaremos os principais comandos utilizados para navegação em diretórios.

#### 1.3.1 Comandos de Navegação

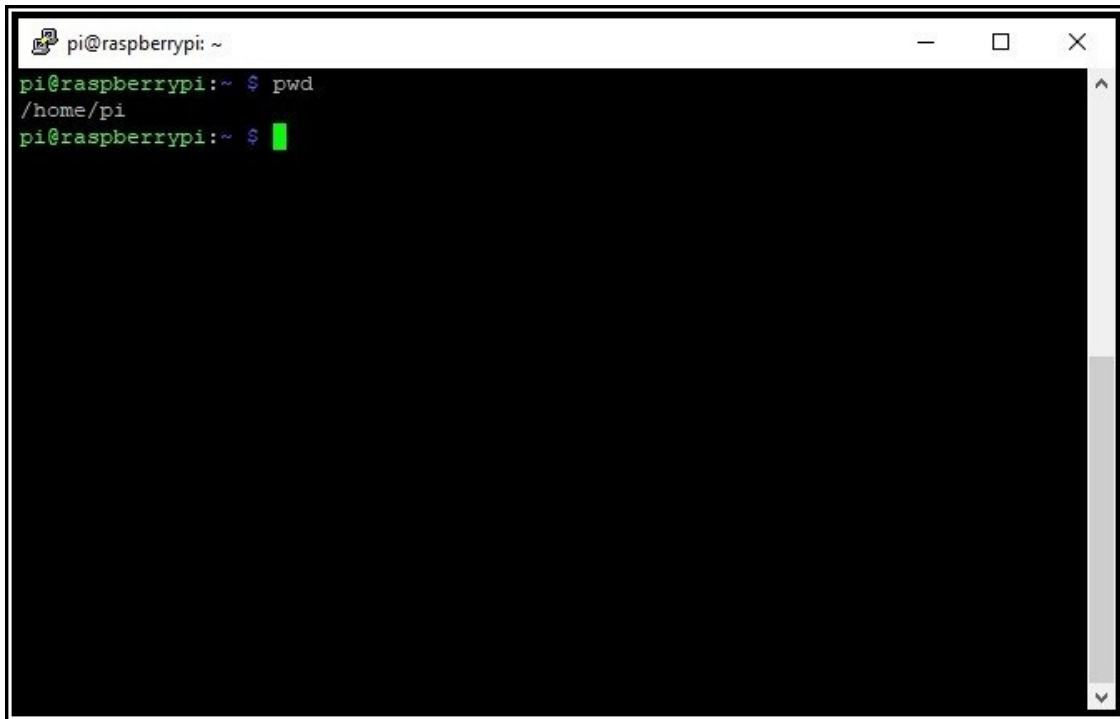
A Tabela 2 mostra os principais comandos para navegação em diretórios.

Tabela 1: Comandos para operações em diretórios

<b>pwd</b>	Mostra o diretório atual
<b>ls</b>	Lista os arquivos e diretórios do diretório atual <i>dir</i>
<b>cd <i>dir</i></b>	Acessa o diretório <i>dir</i>

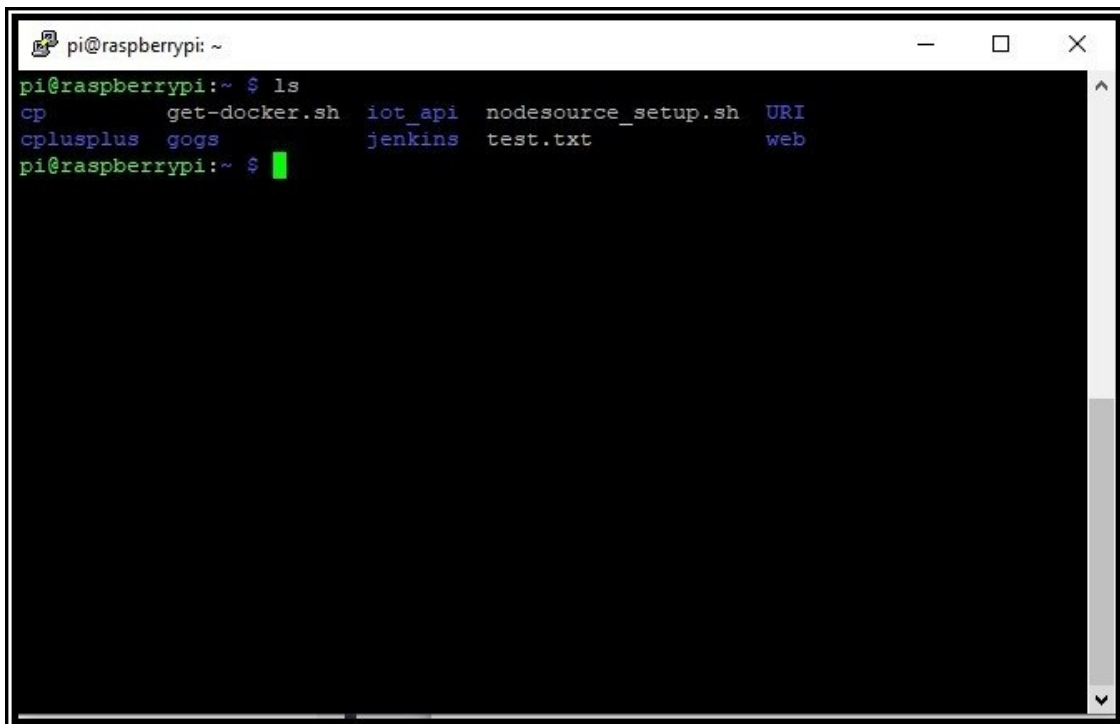
Ao digitar o comando **pwd** no terminal será exibido em qual diretório o usuário se encontra. Esse comando é útil quando o usuário precisa saber o diretório atual antes de executar algum comando. Como pode ser observado na Figura 3, após a execução do comando o retorno no terminal será o diretório atual, neste exemplo `/home/pi`.

O comando **ls** é muito utilizado para visualizar os arquivos e pastas presentes dentro de um diretório. A Figura 4 mostra a execução do documento `ls` em um terminal. O comando **ls** pode ser usado também com uma série de parâmetros. A Tabela 2 mostra alguns parâmetros úteis para serem utilizando com comando **ls**. Após a execução do comando **ls -la** (cf. Tabela 2), A Figura 5 mostra um exemplo da execução do comando `ls` com esses parâmetros. Na Tabela 4 é apresentado uma descrição do retorno da execução do comando `ls -la` conforme observado na Figura 5.

A terminal window titled 'pi@raspberrypi: ~' with standard window controls. The prompt is 'pi@raspberrypi:~ \$'. The command 'pwd' has been entered and executed, resulting in the output '/home/pi'. The prompt is now 'pi@raspberrypi:~ \$' with a green cursor.

```
pi@raspberrypi:~ $ pwd
/home/pi
pi@raspberrypi:~ $
```

Figura 3: Exemplo da execução do comando pwd

A terminal window titled 'pi@raspberrypi: ~' with standard window controls. The prompt is 'pi@raspberrypi:~ \$'. The command 'ls' has been entered and executed, resulting in a two-line listing of files and directories: 'cp get-docker.sh iot\_api nodesource\_setup.sh URI' and 'cplusplus gogs jenkins test.txt web'. The prompt is now 'pi@raspberrypi:~ \$' with a green cursor.

```
pi@raspberrypi:~ $ ls
cp      get-docker.sh  iot_api  nodesource_setup.sh  URI
cplusplus  gogs          jenkins  test.txt              web
pi@raspberrypi:~ $
```

Figura 4: Exemplo da execução do comando ls para listagem de arquivos e pastas

O comando **cd** é utilizado para navegação entre diretórios e para isso o usuário precisa ter permissão de leitura do diretório que ele pretende acessar. A sintaxe do comando **cd** é a seguinte: **cd** *[diretório]*, sendo *[diretório]* o nome do diretório que se deseja acessar.

Tabela 2: Parâmetros úteis do comando `ls`

<code>ls -a [diretório]</code>	Lista todos os itens incluindo os ocultos de um <i>diretório</i> . Caso o parâmetro <i>[diretório]</i> não seja fornecido, o diretório <code>.</code> (atual) será utilizado.
<code>ls -l [diretório]</code>	Parecido com o comando anterior ( <code>ls -a [diretório]</code> ), porém exibe informações detalhadas linha a linha sobre os arquivos e pastas.
<code>ls -la [diretório]</code>	Parecido com o comando anterior ( <code>ls -l [diretório]</code> ), porém exibe informações detalhadas linha a linha sobre os arquivos e pastas visíveis e ocultos.
<code>ls -F [diretório]</code>	Mostra os arquivos e pastas em <i>[diretório]</i> e adiciona uma barra <code>\</code> quando o item for do tipo diretório
<code>ls -lh [diretório]</code>	Mostra o tamanho dos arquivos em Kbytes, Mbytes e Gbytes presentes em <i>[diretório]</i>
<code>ls -n [diretório]</code>	Mostra a identificação numérica de usuários e grupos ao invés dos nomes.
<code>ls -r [diretório]</code>	reverte a ordem da listagem.
<code>ls -S [diretório]</code>	ordena pelo tamanho do arquivo.
<code>ls -t [diretório]</code>	ordena pela data do arquivo.
<code>ls -X [diretório]</code>	ordena pela extensão do arquivo.

Tabela 3: Explicação do retorno do comando `ls` com parâmetro `-la`

<code>-rw-r--r--</code>	Permissões de acesso ao arquivo/diretório
<code>1</code>	Caso seja um diretório, mostra a quantidade de subdiretórios existentes dentro dele. Caso for um arquivo o valor será <code>1</code> .
<code>pi</code>	Nome do usuário dono do arquivo/diretório
<code>pi</code>	Nome do grupo ao qual o arquivo/diretório pertence
<code>13857</code>	Tamanho do arquivo em bytes
<code>Jun</code>	Mês que o arquivo/diretório foi criado ou modificado.
<code>5</code>	Dia que o arquivo/diretório foi criado ou modificado.
<code>23:31</code>	horário que o arquivo/diretório foi criado ou modificado.
<code>get-docker.sh</code>	nome do arquivo.

A Tabela X apresenta alguns exemplos de utilização do comando `cd`.

```

pi@raspberrypi:~ $ ls -la
total 112
drwxr-xr-x 13 pi pi 4096 Aug 13 05:13 .
drwxr-xr-x 7 root root 4096 Jun 17 17:57 ..
-rw-r--r-- 1 pi pi 6325 Aug 11 14:38 .bash_history
-rw-r--r-- 1 pi pi 220 May 27 08:10 .bash_logout
-rw-r--r-- 1 pi pi 3523 May 27 08:10 .bashrc
drwxr-xr-x 2 root root 4096 Jul 18 04:09 cp
drwxr-xr-x 2 pi pi 4096 Jun 7 21:07 cplusplus
-rw-r--r-- 1 pi pi 13857 Jun 5 23:31 get-docker.sh
-rw-r--r-- 1 pi pi 72 Jun 5 23:50 .gitconfig
drwxr-xr-x 3 pi pi 4096 Jun 5 20:22 .gnupg
drwxr-xr-x 2 pi pi 4096 Jun 5 23:53 gogs
drwxr-xr-x 3 pi pi 4096 Jun 10 17:28 .local
drwxr-xr-x 2 pi pi 4096 Jun 17 20:50 .local
drwxr-xr-x 3 pi pi 4096 Jun 5 23:39 .local
-rw-r--r-- 1 pi pi 12991 Jun 10 17:34 .local
drwxr-xr-x 4 pi pi 4096 Jun 14 12:28 .npm
-rw-r--r-- 1 pi pi 807 May 27 08:10 .profile
drwxr-xr-x 3 pi pi 4096 Jun 7 21:45 .ssh
-rw-r--r-- 1 pi pi 0 Aug 11 10:50 test.txt
-rw-r--r-- 1 pi pi 0 Aug 13 05:13 text.backup
drwxr-xr-x 3 pi pi 4096 Jun 7 23:20 URI
drwxr-xr-x 3 root root 4096 Jul 18 04:10 web
-rw-r--r-- 1 pi pi 165 Jun 5 22:24 .wget-hsts
pi@raspberrypi:~ $

```

Figura 5: Exemplo da execução do comando ls com parâmetro -la

Tabela 4: Explicação do retorno do comando ls com parâmetro -la

cd .	Mantém no diretório atual, uma vez que . é um aliás que representa o diretório atual.
cd	Ao digitar apenas o comando cd sem o parâmetro [diretório] o terminal retornará ao diretório home do usuário (nesse caso /home/pi)
cd ~	Mesmo retorno do comando anterior (cd).
cd /	Terminal retorna ao diretório raiz sistema.
cd -	Terminal retorna ao último diretório acessado.
cd ..	Retorna um diretório na hierarquia. Por exemplo: /home/pi => /home
cd ../[diretório]	Retorna um diretório na hierarquia e depois acessa o diretório [diretório]

### 1.3.2 Comandos para manipulação de arquivos e diretórios

**mkdir** é um comando utilizado para criar um diretório no sistema. Como foi visto na seção 1.2, um diretório é usado para armazenar arquivos e também outros diretórios. A sintaxe do comando **mkdir** é a seguinte: **mkdir** [opções] [caminho/diretório]. Nesse caso:



- *[caminho]*: é o caminho na árvore de diretórios que o novo diretório será criado.
- *diretorio*: nome do diretório que será criado.
- *opções*:
  - **-p** criar toda estrutura de diretórios conforme o caminho informado. Por exemplo, para o comando **mkdir -p /home/thiago/exemplo/novapasta**, caso o diretório */exemplo* não exista, ele será criado se o usuário fornecer a opção **-p**.
  - **-verbose** para cada diretório criado será mostrado uma mensagem sobre a criação. **-p**.

É possível também criar muitos diretórios com apenas um comando **mkdir**, para isso o usuário precisa utilizar a seguinte sintaxe:

- **mkdir** *[opcoes]* *[caminho/diretório]* *[camibho1/diretorio1]* *[camibho2/diretorio2]*

**rm** comando utilizado para apagar arquivos e diretórios e seus subdiretorios. A sintaxe do comando **rm** é a seguinte: **rm** *[opcoes]* *[caminho][arquivo/diretório]* *[caminho][arquivo1/diretorio1]*. Nesse caso:

- **[caminho]**: localização do arquivo ou diretório que será apagado. Caso o caminho não seja fornecido, será considerado que o arquivo/diretório esteja no diretório atual do usuário.
- **[arquivo/diretório]** nome do arquivo ou diretório que será apagado
- *Opcoes*:
  - **-i** pergunta antes de remover um arquivo/diretorio.
  - **-v** mostra os arquivos na medida que são removidos.
  - **-r** remove arquivos e diretórios em subdiretorios. Nesse caso ele apaga recursivamente todos os subdiretorios do diretório fornecido no comando **rm**.
  - **-f** remove os arquivos sem perguntar.

Alguns exemplos de uso do comando **rm**:

- **rm aula01.txt** - apaga o arquivo *aula01.txt* no diretório atual.
- **rm -r meusdocumentos** - apaga recursivamente todos os arquivos e subdiretorios do diretório *meusdocumentos*.
- **rm -r meusdocumentos/\*** - apaga todos os arquivos e subdiretorios do diretório *meusdocumentos*, mas não apaga o diretório *meusdocumentos*.
- **rm \*.txt** - apaga todos os arquivos do diretório atual que tenham a extensao *.txt*.
- **rm -r ../outrosarquivos/\*** - sobe um nível na estrutura de diretórios e apaga todos os arquivos e subdiretorios do diretório *outrosarquivos*.
- **rm -r /home/thiago/backup** - apaga todos os arquivos e subdiretorios do diretório */home/thiago/backup* inclusive o próprio diretório.

O comando **cp** é utilizado para copiar arquivos e diretorios. tal comando copia arquivos da *[origem]* para o *[destino]*. Ambos *[origem]* e *[destino]* terão o mesmo arquivo/diretório de *[origem]* após a execução do comando. A sintaxe do comando **cp** é a seguinte: **cp** *[opções]* *[origem]* *[destino]*. Nesse caso:

- **[origem]** é o arquivo ou diretório que será copiado.
- **[destino]** O caminho do arquivo ou diretório que o arquivo/diretório presente em [origem] será copiado.
- **opcoes:**
  - **-i** pergunta antes de substituir um arquivo existente.
  - **-f** substitui tudo sem perguntar.
  - **-R** copia recursivamente os arquivos e sub-diretorios de [origem] para [destino].

Alguns exemplos de utilização do comando **cp** são vistos a seguir:

- **cp** aula01.txt teste02.txt - copia o arquivo aula01.txt para o arquivo aula02.txt. Se o arquivo aula02.txt não existir ele é criado após a execução do comando.
- **cp** aula01.txt /aulas - copia o arquivo aula01.txt para o diretório /aulas.
- **cp \*** /aulas - copia todos os arquivos do diretório atual para o diretório /aulas.
- **cp** aulas/\* aulasbackup - copia todos os arquivos presentes no diretório aulas para o diretório aulas-backup
- **cp** backupaulas/\* . - copia todos os arquivos em backupaulas para o diretório atual.

O comando **mv** é utilizado para mover ou renomear um arquivo ou diretório. O processo é semelhante ao do comando **cp** mas no final da execução do comando o arquivo de [origem] é movido para [destino] e depois é apagado de [origem]. A sintaxe do comando **mv** é a seguinte: **mv** [opcoes] [origem] [destino]. Nesse Caso:

- **[origem]** é o arquivo ou diretório que será movido.
- **[destino]** O caminho do arquivo ou diretório que o arquivo/diretório presente em [origem] será movido.
- **opcoes:**
  - **-f** - substitui o arquivo de destino sem perguntar
  - **-v** mostra os arquivos que estão sendo movidos

Alguns exemplos de utilização do comando **mv** são vistos a seguir:

- **mv** aula01.txt aula02.txt - renomeia o arquivo aula01.txt para aula02.txt
- **mv** aulas/\* aulasbackup - move todos os arquivos presentes no diretório aulas para o diretório aulasbackup.
- **mv** aulasbackup backupaulas - renomeia o diretório aulasbackup para backupaulas.

O comando **touch** é utilizado principalmente para alterar a data de um arquivo. Também é utilizado com frequência para criar um novo arquivo. A seguir é mostrado a sintaxe do comando: **touch** [opções] [arquivos]. Em nosso caso não estamos preocupados com a alteração da data de um arquivo, mas sim a criação dele, portanto vejamos alguns exemplos:

- **touch** aula01.txt - cria o arquivo aula01.txt
- **touch** aula01.txt aula02.txt aula03.txt - cria os arquivos aula01.txt, aula02.txt e aula03.txt no diretório atual.
- **touch** aulas/aula01.txt - cria o arquivo aula01.txt no diretório aulas.

O comando **cat** mostra o conteúdo de um arquivo binário ou de texto. A sintaxe do comando é a seguinte: **cat** [opcoes] [diretório/arquivo]. Nesse caso:

- **[diretório/arquivo]** - localização do arquivo que se deseja visualizar o conteúdo.
- Opções:
  - **-n** - mostra o número de linhas enquanto o conteúdo do arquivo é listado.

Alguns exemplos de utilização do comando **cat** são vistos a seguir:

- **cat** aula01.txt - mostra o conteúdo do arquivo aula 01.txt
- **cat** aulas/aula01.txt - mostra o conteúdo do arquivo aula01.txt que está dentro do diretório aulas.

O comando **head** mostra as primeiras linhas de um arquivo de texto. A Sintaxe do comando é a seguinte: **head** [opcoes] [diretório/arquivo]. Nesse caso:

- **[diretório/arquivo]** - localização do arquivo que se deseja visualizar o conteúdo.
- Opções
  - **-n** - mostra o número de linhas enquanto o conteúdo do arquivo é listado.

O comando **tail** mostra as últimas linhas de um arquivo de texto. A Sintaxe do comando é a seguinte: **tail** [opcoes] [diretório/arquivo]. Nesse caso:

- **[diretório/arquivo]** - localização do arquivo que se deseja visualizar o conteúdo.
- Opções
  - **-n** - mostra o número de linhas do final do arquivo
  - **-f** mostra continuamente linhas adicionadas no final do arquivo.

## 2 CONTROLE DE VERSÃO

### 2.1 Conceitos básicos

Controle de versão é um sistema que armazena alterações em arquivo e conjunto de arquivos no decorrer do tempo para facilitar a recuperação de versões específicas desses arquivos. Esse sistema permite reverter arquivos modificados para versões anteriores, comparar mudanças no decorrer do tempo, ver quem realizou atualizações no código que pode estar causando problemas, quem pode ter adicionado algum código específico que pode estar travando o sistema etc.

### 2.2 Sistemas de controle de versão locais

O método de controle de versão utilizado por muitas pessoas é o de copiar arquivos em outro diretório. Essa abordagem é muito comum porque é muito simples, mas também é incrivelmente sujeita a erros. É fácil esquecer em qual diretório você está e atualizar acidentalmente um arquivo errado ou copiar arquivos indesejados.

Para lidar com este tipo de problema, os programadores desenvolveram há muito tempo sistemas de controle de versões locais que tinham um banco de dados simples que mantinha todas as mudanças nos arquivos.

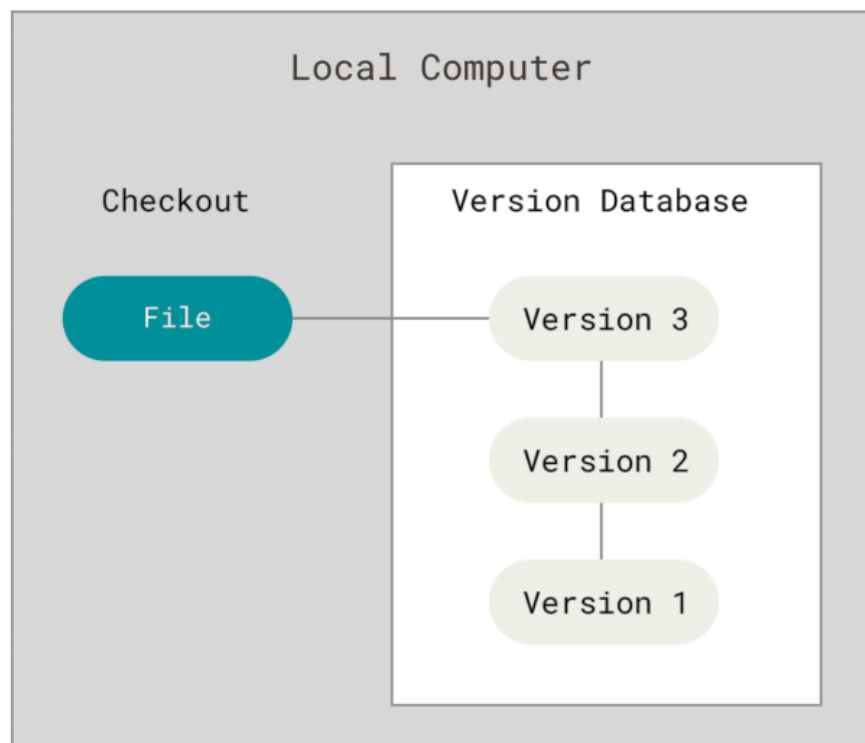


Figura 6: Sistema de controle de versão local

### 2.3 Sistemas de controle de versão centralizados

Com o passar do tempo outro grande problema que as pessoas encontraram foi a necessidade de colaborar com os desenvolvedores em outros sistemas. Para lidar com este problema, os Sistemas Centralizados de Controle de Versão (CVCSs) foram desenvolvidos. Esses sistemas (como CVS, Subversion e Perforce) têm

um único servidor que contém todos os arquivos com controle de versão e vários clientes que fazem o check-out dos arquivos daquele local central.

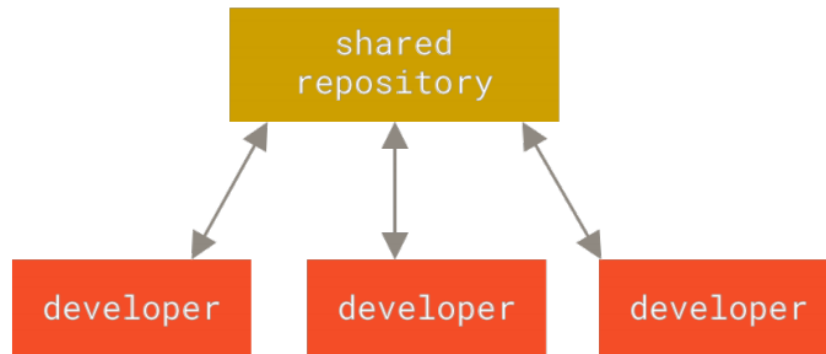


Figura 7: Sistema de controle de versão centralizado

Essa configuração oferece muitas vantagens, especialmente em relação aos Sistemas de controle de versão locais. Por exemplo, todo mundo sabe o que todos os outros no projeto estão fazendo. Os administradores têm controle refinado sobre quem pode fazer o quê, e é muito mais fácil administrar um Sistema de controle de versão centralizado do que lidar com bancos de dados locais em cada cliente.

No entanto, essa configuração também tem grandes desvantagens. A mais óbvia é que o servidor centralizado representa o único ponto de falha do sistema. Se o servidor ficar inativo por uma hora, então durante essa hora ninguém pode colaborar ou adicionar novas versões em nada que esteja trabalhando. Se o disco rígido que o banco de dados central está armazenando os dados for corrompido e backups adequados não forem realizados, absolutamente tudo pode ser perdido - todas as versões projeto, exceto as versões que estejam nas máquinas locais dos usuários. Os sistemas de controle de versão locais sofrem deste mesmo problema - sempre que você tem todo o histórico do projeto em um único lugar, você corre o risco de perder tudo caso algum problema ocorra na máquina que o banco de dados esteja.

## 2.4 Sistemas de controle de versão distribuídos

Em um Sistema de controle de versão distribuído (como Git, Mercurial, Bazaar ou Darcs), os clientes não apenas possuem as versões mais recentes dos arquivos; em vez disso, eles realizam uma cópia completa do repositório, incluindo seu histórico completo. Assim, se algum servidor morrer, qualquer um dos repositórios do cliente pode ser copiado de volta para o servidor com objetivo de recuperá-lo. Cada clone é, na verdade, um backup completo de todos os dados.

Além disso, muitos desses sistemas lidam bem com vários repositórios, possibilitando que diferentes grupos de pessoas de diferentes maneiras possam trabalhar simultaneamente dentro do mesmo projeto. Isso permite que você configure vários tipos de fluxos de trabalho que não são possíveis em sistemas centralizados ou locais.

## 2.5 Introdução ao Git

O Git é um sistema de controle de versão distribuído que é bastante usado para o desenvolvimento de software. Além disso o Git é gratuito e *open source*, sendo capaz de gerenciar projetos pequenos e grandes com muita rapidez e eficiência. O git é fácil de aprender e ocupa pouco espaço no sistema, garantindo também alta performance.

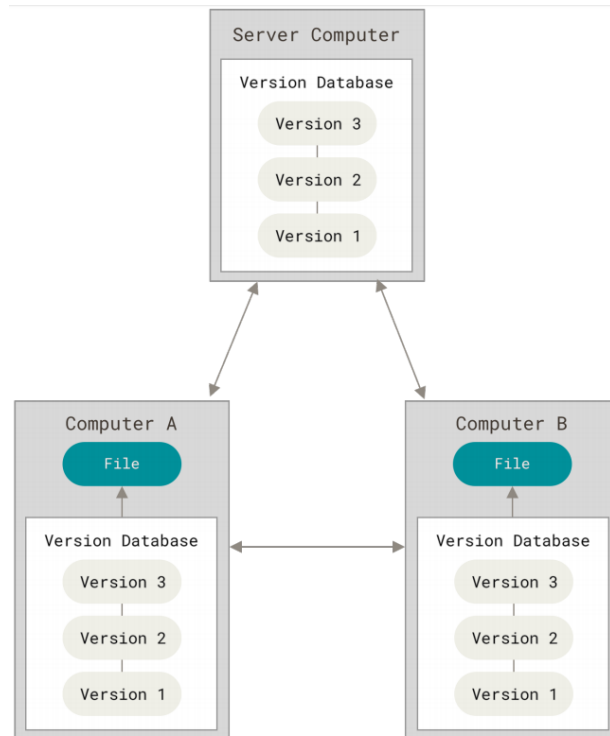


Figura 8: Sistema de controle de versão distribuídos

O Git tem três estados principais em que os arquivos de um repositório podem residir: *modified* (modificado), *staged* (preparado) e *committed* (confirmado).

- **Modified** (modificado) significa que você alterou o arquivo, mas ainda não o enviou ao banco de dados.
- **Staged** (Preparado) significa que você marcou um arquivo modificado em sua versão atual para ir para o próximo **commit**.
- **Committed** (Confirmado) significa que os dados estão armazenados com segurança em seu banco de dados local.

Isso nos leva às três seções principais de um projeto Git: a árvore de trabalho (working tree), a área de preparo (staging area) e o diretório Git (Git directory).

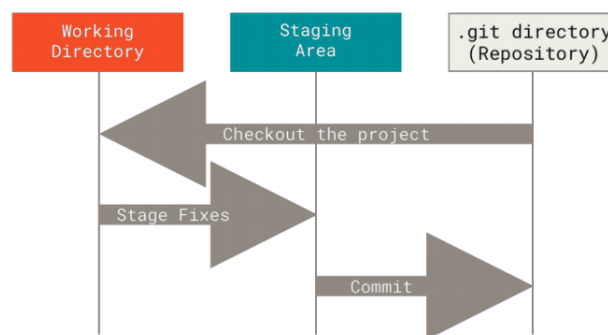


Figura 9: Árvore de trabalho, área de preparo e diretório GIT

A árvore de trabalho é um único check-out de uma versão específica do projeto. Esses arquivos são retirados do banco de dados compactado no diretório Git (repositório) e colocado em disco para você usar ou modificar.

A área de preparo é um arquivo, geralmente contido em seu diretório Git, que armazena informações sobre o que irá para o seu próximo **commit**.

O diretório Git é onde o Git armazena os metadados e o banco de dados de objetos do projeto. Esta é a parte mais importante do git é onde será copiado os arquivos quando você clona um repositório de outro computador.

O fluxo de trabalho básico do Git funciona da seguinte forma:

- Você modifica arquivos na sua árvore de trabalho;
- Você seleciona as mudanças que deseja que façam parte de seu próximo commit e encaminha elas para a área de preparo.
- Você faz um commit (confirmação), que encaminhará os arquivos conforme eles estão na área de preparo e armazena eles permanentemente em seu diretório Git (repositório).

### 2.5.1 Comandos do Git

Para saber se o git já está instalado no sistema e ao mesmo tempo a versão do git executamos o seguinte comando no terminal:

- **git -version**

Caso o comando anterior não seja reconhecido, será necessário instalar o Git no sistema. o download do Git pode ser feito no seguinte website: <https://git-scm.com/downloads>. Após acessar a página, baixe o Git de acordo com o sistema operacional a ser utilizado.

Depois da instalação do Git, podemos navegar até o diretório que queremos fazer controle de versão (geralmente o diretório já possui código fonte ainda não versionado) e criar um novo repositório do Git. Para isso, utilizando o terminal e dentro do diretório do novo repositório digitamos o seguinte comando:

- **git init**

Após a execução do comando anterior será criado um diretório oculto chamado `.git` que armazenará uma estrutura de arquivos e diretórios necessários para o Git conseguir manusear o repositório.

Para relacionar o repositório atual com um repositório remoto hospedado em um servidor git (i.e Github) é necessário primeiro criar o repositório remoto e depois copiar o link para esse repositório. Por exemplo, para o repositório que criamos durante as aulas o link para o repositório remoto seria:

- <https://github.com/tadsifsp/TADS-PFDA1.git>.

De posse do link do repositório, digitamos o seguinte comando no terminal:

- **git remote add origin «link para o repositório»**

Onde «*link para o repositório*» é o link que foi copiado do repositório remoto.

A partir desse momento nosso repositório local criado pelo comando **git init** está relacionado com o repositório remoto hospedado no **GitHub**. Como o nosso repositório remoto já tem alguns arquivos criados, o próximo passo é atualizar o repositório local com o conteúdo disponível no repositório remoto. Para isso executamos o seguinte comando no terminal:

- **git pull origin master**

Após a execução desse comando, todo o conteúdo do repositório será baixado e estará disponível no diretório atual junto com os demais arquivos que já estavam.

Após a execução desses procedimentos, precisamos saber quais arquivos presentes no diretório atual ainda não estão no repositório remoto. Para saber quais arquivos foram criados, modificados ou excluídos, executamos o seguinte comando:

- **git status**

Após a execução desse comando no terminal, uma lista de arquivos ainda não versionados (que não estão no repositório remoto) vai surgir com cada item na cor vermelha. Devemos agora decidir quais arquivos vamos querer enviar para nosso repositório remoto. A partir da lista de arquivos ainda não versionados, podemos adicionar um a um com o seguinte comando:

- **git add «nome do arquivo / diretório»**

Onde «*nome do arquivo / diretório*» é o nome do arquivo ou diretório que será posteriormente versionado. O uso do comando **git add «nome do arquivo / diretório»** deve ser repetido até todos os arquivos que se deseja versionar apareçam em verde ao executar o comando **git status**.

Caso queira adicionar todos os arquivos e diretórios que ainda não estão versionados, é possível usar o comando **git add** no diretório do repositório da seguinte forma:

- **git add .**

Depois que os arquivos e diretórios foram adicionados com o comando **git add**, é necessário criar um **commit** com as alterações realizadas. Um **commit** é como uma fotografia atual do todo o repositório com as alterações que foram introduzidas e adicionadas pelo comando **git add**. Para realizar um **commit** executamos o seguinte comando:

- **git commit -m «mensagem sobre o commit»**

Onde «*mensagem sobre o commit*» é uma mensagem descritiva a respeito das atualizações que foram adicionadas pelo **commit**. Depois que o comando **git commit** é executado, podemos fazer o upload das atualizações para o repositório. Como é o primeiro push que estamos fazendo a partir do repositório local, precisamos executar o seguinte comando:

- **git push -set-upstream origin master**

Depois de executado o comando anterior, será solicitado as credenciais do usuário do GitHub, preencha com as suas credenciais. Em seguida, todas as alterações realizadas e registrados no **commit**, serão enviadas para o repositório remoto, nesse momento uma nova versão do código será criada. Para futuras atualizações do código para esse repositório local, basta usar o comando **git push** pois já foi realizado o direcionamento para a **branch** master na primeira execução do comando.



### 2.5.2 Clonando repositórios

Uma alternativa a todo procedimento realizado anteriormente é fazer uma clonagem do repositório. Quando realizamos uma clonagem estamos copiando toda a estrutura do repositório e sua referência para o diretório de trabalho (*working directory*). Podemos realizar a clonagem de qualquer repositório nosso ou então qualquer outro repositório disponível no GitHub (e outros sites) que sejam públicos e, quando privados, aqueles no qual temos permissão de leitura. Para realizar a clonagem de um repositório executamos no terminal o seguinte comando:

- **git clone «link para o repositório»**

Por exemplo, para o repositório que criamos durante as aulas o comando para clonar esse repositório remoto seria:

- **git clone https://github.com/tadsifsp/TADS-PFDA1.git.**

A figura 10 mostra onde obter a URL do repositório para usar no comando git clone. Para obter o link, basta acessar o repositório que você deseja clonar e clicar no botão Code (1) que abrirá uma caixa de diálogo com os links do repositório (2).

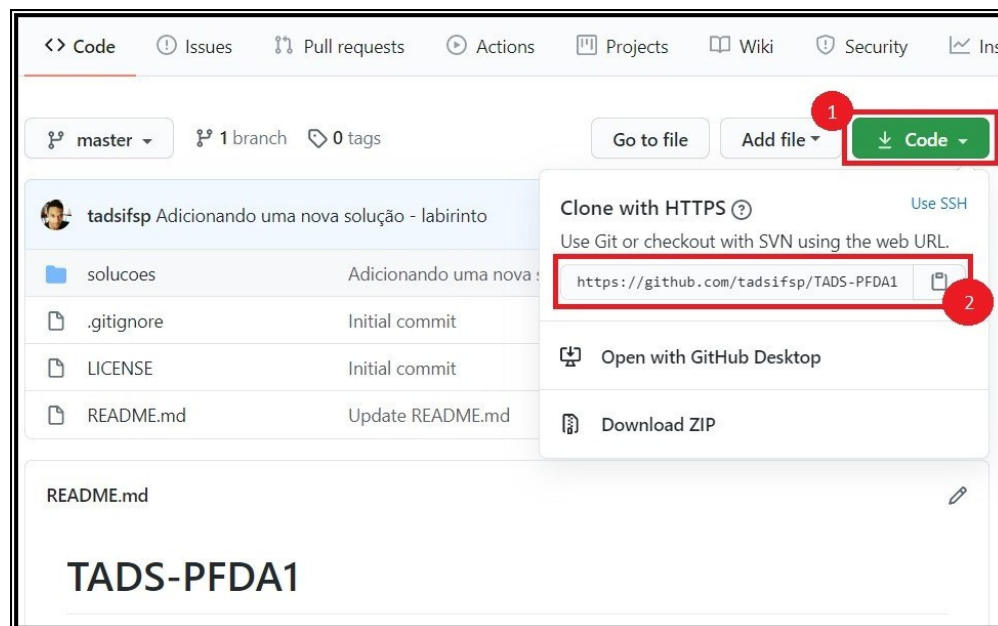


Figura 10: Obtendo o link para clonagem de repositório no git

Após realizar atualizações no repositório clonado, poderemos enviar essas alterações para o repositório remoto. Devemos seguir todos os procedimentos vistos anteriormente: (a) adicionar os arquivos alterações na área de staging (usando git add); (b) confirmar as alterações (usando git commit); e (c) subir as alterações para o repositório remoto (usando o git push).

### 2.5.3 Trabalhando com branches

Quando pensamos no termo *branch* do git estamos nos referindo a uma espécie de ramificação da árvore do projeto, ou seja, uma duplicação de uma versão do repositório que contém código-fonte referente a uma versão específica do projeto. Criamos branches para realizar modificações específicas de forma

paralela ao desenvolvimento de outras modificações feitas por outros desenvolvedores do projeto. É comum organizarmos nosso repositório utilizando workflows (fluxos de trabalhos) predefinidos com base em abordagens adotadas em larga escala pela comunidade de desenvolvimento de software.

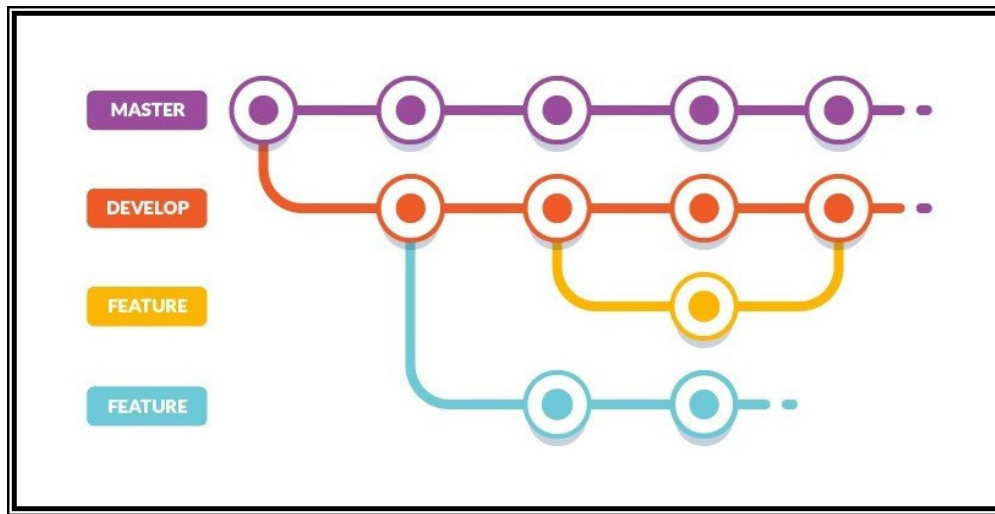


Figura 11: Modelo ilustrativo de fluxo de trabalho do git

A Figura 11 mostra a organização de um *workflow* ilustrativo, onde podemos observar 4 *branches* (master, develop, feature e feature), cada uma dessas ramificações é responsável por armazenar código pertinente a uma determinada função. A cada nova versão do código (quando é realizado o *commit* e *push* no repositório) é representado por um círculo na Figura 11, quando uma nova branch é criada, copiamos uma versão específica de outra branch para servir como base para o desenvolvimento de outra funcionalidade. Observamos que a branch develop é originada de uma versão da branch master e, por sua vez, as branches de features (funcionalidades) a partir da branch develop. No geral a *branch master* mantém o código principal e mais atualizado do repositório, representa a versão do projeto que está pronta para ser utilizada em ambiente de produção (ambiente que roda o projeto oficial utilizado pelos clientes), a *branch develop* por sua vez contém o código que ainda está em desenvolvimento e muitas vezes é utilizada como uma branch intermediária entre a branch master e as de features para reunir código de vários desenvolvedores para serem realizados testes. Cada desenvolvedor será incumbido de desenvolver uma funcionalidade (feature) específica para o projeto, para isso eles devem criar uma branch de feature a partir de uma versão da branch master ou develop mais recente (em alguns casos de uma versão específica, não sendo necessariamente a mais recente) realizar as modificações, atualizar a branch e realizar um *merge request* (solicitação de junção) para adicionar o código novo da branch de feature na branch develop.

A seguir será apresentado alguns modelos de *workflows* mais conhecidos.

#### 2.5.4 Git Workflow

Originalmente proposto por Vincent Driessen, Git Workflow é um fluxo de trabalho de desenvolvimento usando git e várias *branches*. A ideia desse fluxo de trabalho é ter *branches* separadas e reservadas para partes específicas do desenvolvimento:

- **branch master:** Essa branch é responsável por versionar o código de produção mais recente;
- **branch develop:** Essa branch é responsável por versionar o código em desenvolvimento mais recente. As alterações realizadas pelos desenvolvedores são todas aqui reunidas para serem testadas e posteriormente incluídas nas novas versões do projeto;

- **branch hotfix:** As branches hotfix são utilizadas para corrigir pequenos bugs que surgem em produção (durante a execução do sistema) e que não podem esperar pela próxima versão. Geralmente, essas branches são geradas a partir da última versão da branch master e após realizada as correções o código corrigido é enviado tanto para a branch master tanto para as branches de desenvolvimento.
- **branch release:** As branches de release são responsáveis pelo envio de grandes alterações da branch develop para a branch master. São branches de lançamento onde se antecede o processo de deploy que é realizado depois que a branch master recebe essas novas alterações.
- **branch feature:** As branches de features (funcionalidades) são reservadas para desenvolvimento de funcionalidades específicas do projeto. Na divisão de tarefas, cada desenvolvedor criará uma *branch feature* (com um nome específico para a funcionalidade que será desenvolvida), após o desenvolvimento dessa funcionalidade o desenvolvedor fará um requisição para as alterações serem incluídas na **branch develop** que posteriormente será adicionada na **branch master**.

A Figura 12 apresenta uma representação visual desse modelo:

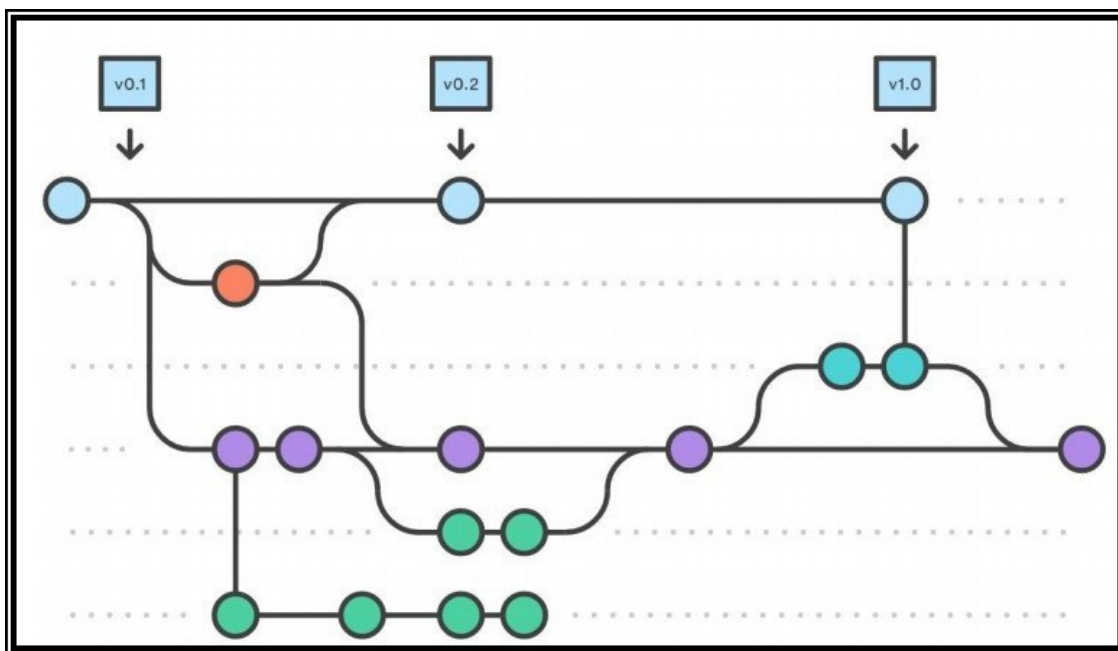


Figura 12: Modelo ilustrativo do Git workflow

A Figura 13 mostra uma linha do tempo da execução do Git workflow:

### 2.5.5 Comandos para criação de branches

Agora que sabemos pra que servem as branches está na hora de aprendermos como fazer pra criar as nossas. Utilizando o terminal, e dentro do diretório do repositório executamos o seguinte comando:

- **git branch** «nome da branch»

Onde «nome da branch» é o nome da nova branch que estamos criando, que não pode ter caracteres especiais em nem espaços. Para identificarmos qual a branch atual digitamos apenas o comando **git branch** sem especificar um nome de branch.

Agora que criamos a nossa branch, precisamos fazer um "checkout" para torná-la a branch ativa, para isso executamos:

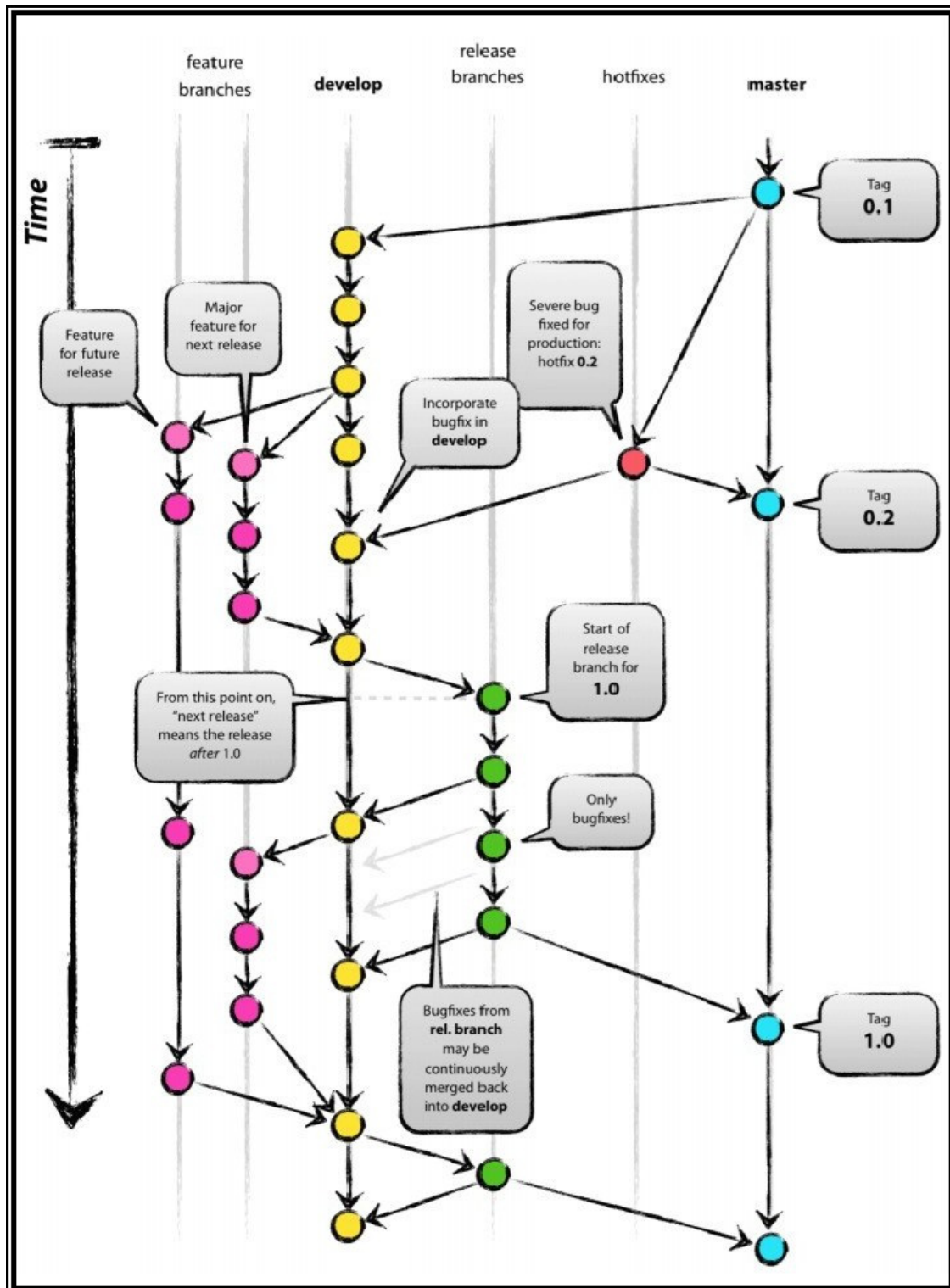


Figura 13: Modelo ilustrativo do Git workflow

- **git checkout** «nome da branch»

Após executarmos o comando anterior já estaremos trabalhando em cima da nova branch criada. Para termos certeza que de fato estamos na nova branch, basta digitarmos **git branch** novamente sem especificar um nome da branch.

Após a execução dos procedimentos anteriores, podemos realizar novas atualizações conforme necessárias. Após realizar as atualizações, precisamos realizar o fluxo de operações do git para versionar um novo código, conforme vimos anteriormente em nosso curso. Como estamos trabalhando em um branch que só existe localmente no computador, precisamos enviar essa branch para o repositório remoto. Para isso precisamos executar o seguinte comando:

- **git push -u origin** «nome da branch»

Após a execução desse comando, nossa branch será enviada para o repositório remoto junto com as suas alterações. Posteriormente será necessário mesclar essas alterações com a branch master, esse procedimento é conhecido como *merge* e vamos estudar sobre isso na próxima seção.

---

## REFERENCES

- [1] Bartsch et al. Suse Linux. User Guide. <https://www-uxsup.csx.cam.ac.uk/pub/doc/suse/suse9.0/userguide-9.0/ch24s02.html>. [Online; acessado em 20 agosto de 2020].
- [2] A.M. Jargas. *Shell Script Profissional*. Novatec Editora, 2017. ISBN 9788575225769. URL <https://books.google.com.br/books?id=B2YkDwAAQBAJ>.
- [3] Wikipedia. Diretório (computação) — Wikipedia, the free encyclopedia. [http://pt.wikipedia.org/w/index.php?title=Diret%C3%B3rio%20\(computa%C3%A7%C3%A3o\)&oldid=56988516](http://pt.wikipedia.org/w/index.php?title=Diret%C3%B3rio%20(computa%C3%A7%C3%A3o)&oldid=56988516), 2020. [Online; acessado em 05 Março de 2020].