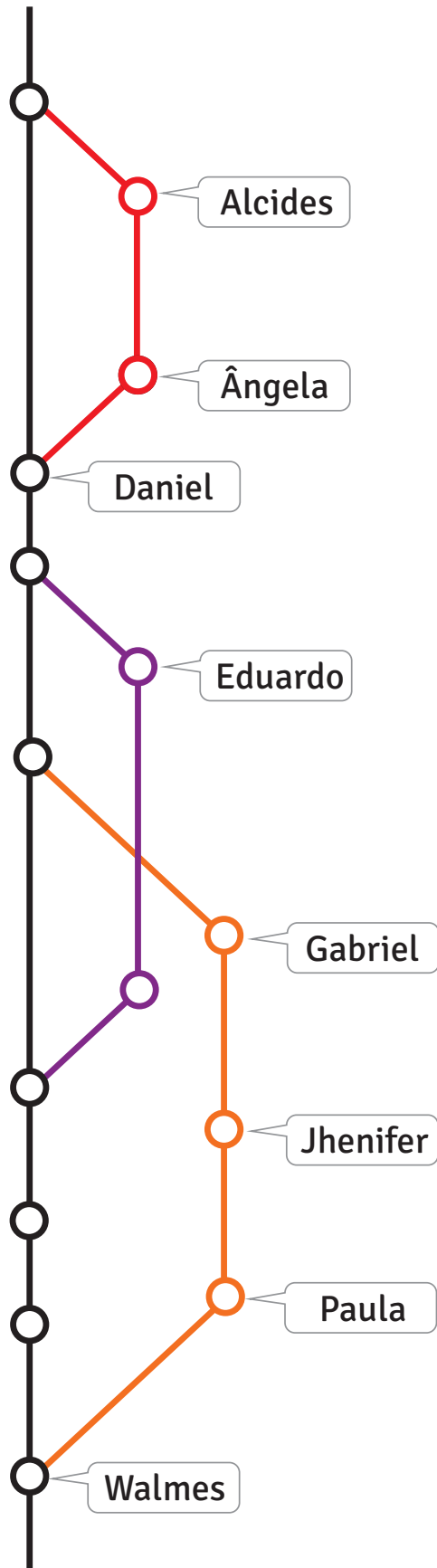


Universidade Federal do Paraná



Apostila Git



PET-Estatística
UFPR

*Não é a vontade de vencer que ganha o jogo, e
sim a vontade de se preparar para vencê-lo.*

—Paul “Bear” Bryant

Sumário

1	Sistemas de controle de versão	9
2	Instalação e Configuração	11
2.1	Instalação	11
2.1.1	Windows	11
2.1.2	Linux	16
2.1.3	MacOS	17
2.2	Configurando Perfil	17
2.2.1	Usuário	17
2.2.2	Atalhos	18
2.2.3	Ignorar Arquivos	20
3	Repositórios Locais	21
3.1	Instruções do Git	21
3.2	Meu Primeiro Repositório	24
3.3	Complementos	27
3.4	Versões de Arquivos Diferentes	28
3.5	Trabalhando com Ramos	37
3.6	Resolvendo conflitos	48
4	Projetos Remotos	55
4.1	Repositório remoto pessoal	55
4.2	Repositório remoto coletivo	58
4.3	Fluxo de trabalho com repositório remoto, do clone ao push	59
4.3.1	Git clone	59
4.3.2	Git Push	59
4.3.3	Git Pull	60
4.3.4	Git fetch	60
4.4	Listar branches locais/remotos	60
4.5	Adicionar, renomear, deletar remote	61
4.5.1	Adicionando repositórios remotos	61
4.5.2	Obtendo informações de um Remoto	61
4.5.3	Renomeado Remotos	61
4.5.4	Removendo Remotos	61
4.5.5	Deletar ramos no servidor	62
4.5.6	Clonar apenas um <i>branch</i> , <i>commit</i> ou <i>tag</i>	62
4.6	Criando um Repositório Git	62
4.7	Git no servidor	63
4.8	Configuração de Conexão SSH com Servidor	63

5	Serviços Web para Projetos Git	67
5.1	Serviços Web para Git	67
5.1.1	GitHub	68
5.1.2	GitLab	70
5.2	Criar um perfil	71
5.2.1	Habilitar comunicação	72
5.2.2	Gerenciar repositórios	74
5.3	Fluxo de trabalho	79
5.4	Macanismos de colaboração	83
5.4.1	Issues	83
5.4.2	Fork	84
5.4.3	Merge Request	84
5.5	Integração contínua	86
5.5.1	GitHub	88
5.5.2	GitLab	89
6	Ferramentas gráficas	95
6.1	Interfaces Git	96
6.1.1	git-gui	96
6.1.2	gitk	98
6.1.3	Outras Interfaces	100
6.2	Interfaces de comparação	106
7	Trabalhando em equipe	115
7.1	Boas práticas de colaboração	115
7.2	Modelos de fluxos de trabalho	118
7.2.1	Centralized workflow	118
7.2.2	Feature branch workflow	119
7.2.3	Gitflow workflow	120
7.2.4	Forking workflow	120
7.3	Fluxo de trabalho PET no GitLab	121
	Apêndice	122
A	Exemplos de rotinas	123
B	Dicionário de termos	131
B.0.1	Config	131
B.0.2	SSH Key	131
B.0.3	Help	132
B.0.4	Repositório	132
B.0.5	Stagin Area	132
B.0.6	Remote	132
B.0.7	Clone	132
B.0.8	Status	133
B.0.9	Add	133
B.0.10	Commit	133
B.0.11	Branch	133
B.0.12	Checkout	134
B.0.13	Merge	134

B.0.14	Rm	134
B.0.15	Mv	135
B.0.16	Push	135
B.0.17	Fetch	135
B.0.18	Pull	135
B.0.19	HEAD	136
B.0.20	Tag	136
B.0.21	Stash	136
B.0.22	Reset	137
B.0.23	Rebase	137
B.0.24	Blame	137
B.0.25	Bisect	137

Capítulo 1

Sistemas de controle de versão

Inicialmente, podemos dizer que Git é um Sistema de Controle de Versão, que permite ao programador armazenar diversas cópias de versão do seu trabalho, restaurar versões anteriores, sincronizar entre diversos computadores de trabalho e trabalhar colaborativamente com outros programadores. Só com essas possibilidades já faz do Git uma ferramenta muito útil a quem programa. Mas o Git é muito mais! É possível utilizar o Git através do Shell (linha de comando) ou através de diversas interfaces gráficas e até mesmo dentro do Rstudio. Você pode integrar seu projeto com o Github ou Gitlab e disponibilizar os arquivos na web. Assim, você pode acessá-los e até editar seus arquivos via navegador. Pode deixar os arquivos públicos e disponibilizar seu código à comunidade de programadores. Outras pessoas podem até vir a colaborar nos seus projetos. Neste conceito, você pode fazer uso ou colaborar com projetos de outros programadores! Acompanhar o desenvolvimento de projetos que sequer foram lançados, fazer sugestões, tirar dúvidas e entrar em contato direto com equipes e desenvolvedores. Isso transforma o Github e Gitlab numa rede social de programadores!

O Git não é o único sistema de controle de versão. Nem foi o primeiro. Os primeiros sistemas de controle de versão foram lançados na década de 70. Há sistemas abertos e proprietários. E também, sistemas que trabalham somente de forma local, cliente-servidor ou sistema distribuído. Dentre os sistemas abertos, os mais conhecidos são o Apache Subversion (SVN), Mercurial, Git, Veracity e Bazaar. Mas, porque tantos sistemas foram desenvolvidos? Cada sistema foi desenvolvido buscando resolver os mesmos problemas de diferentes formas. A comunidade desenvolvedora do kernel (núcleo) do Linux utilizava o BitKeeper, um software proprietário que decidiu revogar a licença gratuita. Linus Torvalds, desenvolvedor do primeiro kernel, estudou os diversos softwares de controle de versão para ser o substituto do BitKeeper. Mas nenhum software atendia as necessidades, principalmente na performance de um projeto tão grande. Então, ele iniciou o desenvolvimento do software que ele chamou de Git e em menos de dois meses todo o gerenciamento do kernel já estava transferido para o novo sistema.

Então, utilizar o Git é a garantia de um sistema robusto de controle de versionamento. Um sistema aberto e muito utilizado por programadores, estatísticos e cientistas de dados.

Seja você mais um desenvolvedor a resolver os seus problemas e participar desta comunidade.

Capítulo 2

Instalação e Configuração

Agora, devidamente apresentados ao sistema de versionamento Git vamos utilizá-lo. Porém, antes de começarmos a entender os comandos Git, é necessário sua instalação e configuração. Neste capítulo veremos primeiramente como instalar o programa Git em diferentes sistemas operacionais e posteriormente como configurar algumas opções para viabilizar e facilitar seu uso.

2.1 Instalação

2.1.1 Windows

Usuários Windows devem visitar Git for Windows¹, clicar em “Download” e baixar o arquivo “.exe”.

Após o download, execute o arquivo e você terá a tela conforme figura 2.1:

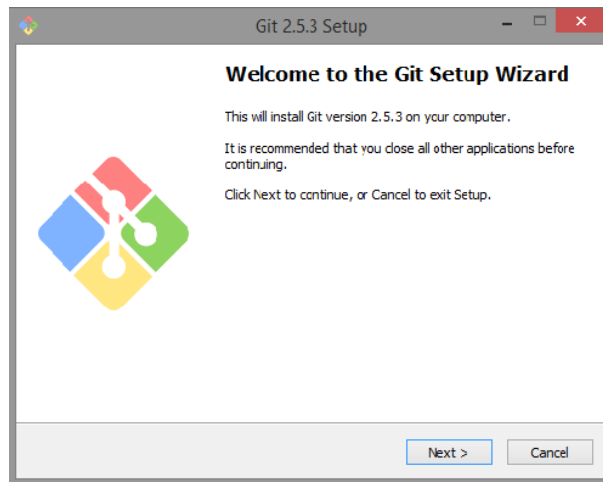
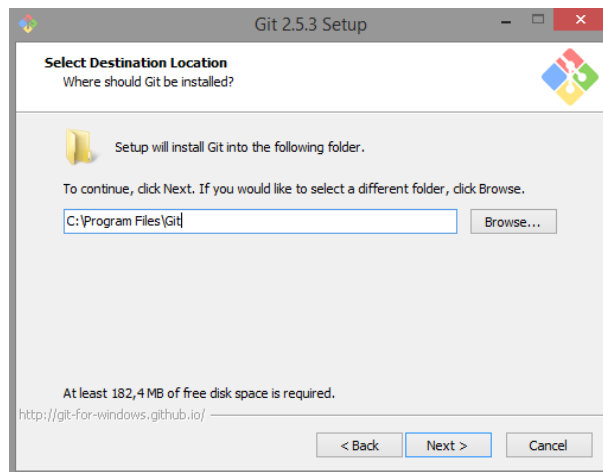
Como de costume, clique em “Next”. Para dar continuidade a instalação aceite a licença do Git.

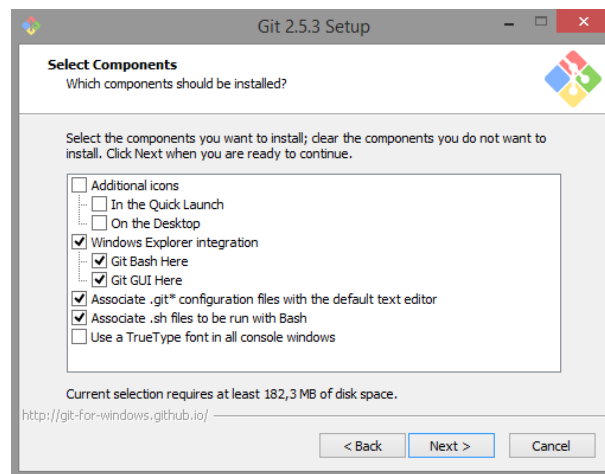
O diretório apresentado na figura 2.2 vem como default, porém é possível alterar a instalação para um diretório de sua preferência. Depois de selecionado o caminho da instalação, clique em “Next” para prosseguir.

Na tela de componentes (figura 2.3) podemos definir atalhos, integração ao menu de contexto do Windows Explorer, associação de arquivos e uso de font TrueType. O Git Bash é o prompt de comandos próprio, que além dos comandos Git também fornece alguns comandos Unix que podem ser bem úteis. Já o Git GUI é uma interface gráfica para trabalhar com Git. É recomendável a seleção de ambos os itens.

Depois de selecionado os componentes de sua preferência, clique em “Next” para dar continuidade.

¹<https://git-for-windows.github.io/>

Figura 2.1: *Printscreen* do passo 1Figura 2.2: *Printscreen* do passo 2

Figura 2.3: *Printscreen* do passo 3

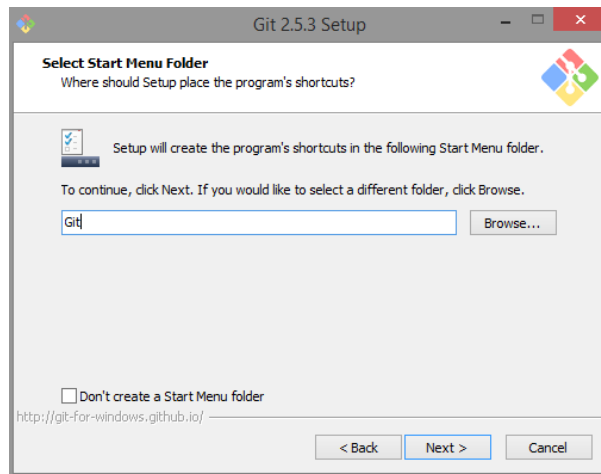
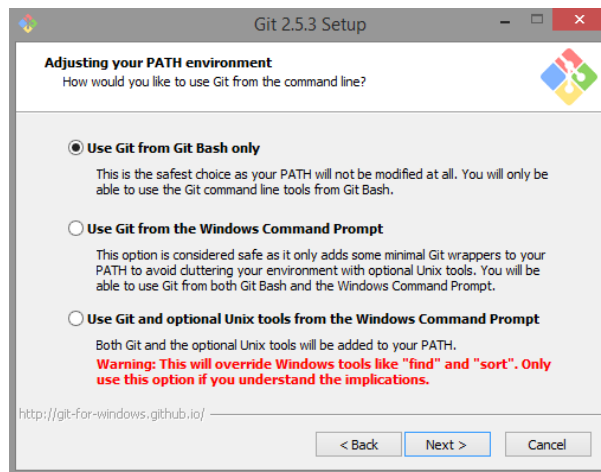
No passo 4, representado pela figura 2.4, o instalador nos oferece a oportunidade de mudar o nome da pasta no menu iniciar, recomenda-se deixar o padrão para fácil localização posteriormente.

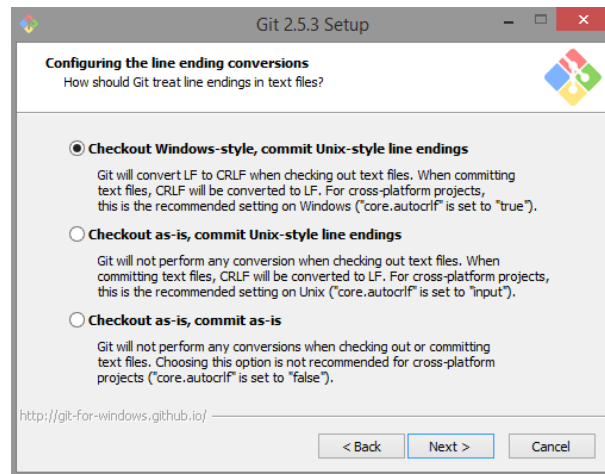
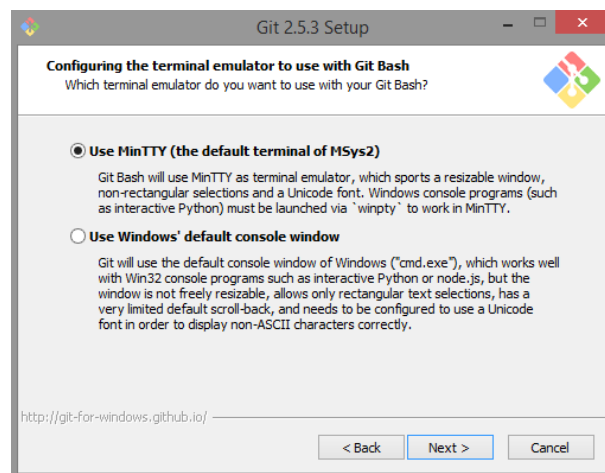
Na tela de configuração “PATH environment”, conforme a figura 2.5, podemos escolher as formas de integração do Git com o sistema. A primeira opção nos permite usar o Git apenas pelo “Git Bash” (é o prompt de comando do Git), a segunda opção nos possibilita executar os comandos no “Git Bash” e no prompt de comando do Windows (cmd.exe), e a terceira opção é a junção das duas de cima, porém alguns comandos do Windows serão substituídos por comandos Unix com mesmo nome. Essa última opção não é recomendada, a primeira opção é a desejável.

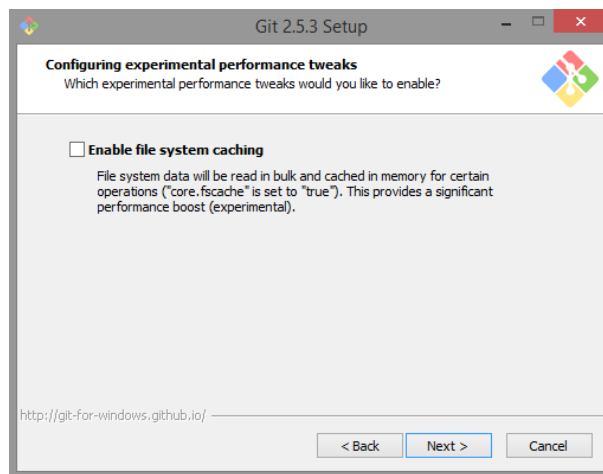
Na figura 2.6, temos a configuração de quebra de linha. Windows e sistemas Unix (Linux, Mac) possuem formatos diferentes de quebra de linha em arquivos de texto. Se você escreve um código com quebras de linha no formato Windows, outra pessoa pode ter problemas ao abrir o mesmo arquivo em um Linux, e vice-versa. Este passo, portanto, permite normalizar isso. A primeira opção converte automaticamente os arquivos para padrão Windows quando receber algum arquivo e converterá para padrão Unix quando “comitar” (enviar alterações) ao repositório. A segunda opção, não faz nenhuma conversão ao receber arquivos, mas convertem para padrão Unix ao “comitar”. Já a terceira opção, o Git não fará nenhuma conversão. Recomenda-se a seleção da opção “Checkout Windows-style, commit Unix-style line endings”.

No passo da figura 2.7, temos a configuração do emulador de terminal para usar com o Git Bash. A primeira opção utiliza o terminal MSys2 (Shell), que permite utilizar comandos Unix no Windows. Já a segunda opção, utiliza o terminal padrão do Windows. Recomendamos a primeira opção. Feito isso, dê continuidade a instalação.

E por último, a figura 2.8, que configura ajustes de performance. Essa opção é para habilitar o sistema de cache de arquivo.

Figura 2.4: *Printscreen* do passo 4Figura 2.5: *Printscreen* do passo 5

Figura 2.6: *Printscreen* do passo 6Figura 2.7: *Printscreen* do passo 7

Figura 2.8: *Printscreen* do passo 8

Feito isso, “Next”, “Finish” e o Git está instalado.

2.1.2 Linux

Em qualquer sistema Linux, pode-se utilizar o gerenciador de pacotes da respectiva distribuição para instalar o Git. Basta executar o código de instalação de sua respectiva distribuição.

Debian

Em uma sessão de terminal Linux de distribuições Debian (Ubuntu, Mint), execute o código abaixo. Adicione o ppa para obter a versão mais recente do Git.

```
sudo add-apt-repository ppa:git-core/ppa
sudo apt-get update
```

Agora, execute o comando abaixo para instalação do Git. Siga as instruções do prompt de comando, primeiro confirmando a instalação dos pacotes e suas dependências, depois confirmando a instalação do pacote git-core.

```
sudo apt-get install git git-core git-man git-gui git-doc \
ssh openssh-server openssh-client
git --version
```

Para adicionar ferramentas complementares, execute:

```
sudo apt-get install gitk meld
```

Arch


```
pacman -S git openssh meld
git --version
```

Fedora

```
yum install git
git --version
```

Usuários de outra versão do Linux podem visitar Download for Linux².

2.1.3 MacOS

Existem duas maneiras de instalar o Git no Mac, uma pelo instalador e outra através do MacPorts.

Utilizando o Instalador

O usuário deverá acessar Download for Mac³, clicar em “Download” e baixar o arquivo “.dmg”.

Após o download, é necessário clicar duas vezes para ter acesso ao pacote de instalação. Dentro do arquivo “.dmg”, execute o arquivo “.pkg” para iniciar a instalação. Siga os passos até concluir a instalação. É recomendável utilizar a instalação padrão.

Para testar a instalação, abra o terminal e digite o comando “git”. A saída deverá ser similar a figura 2.9:

Utilizando o MacPorts

A maneira mais fácil de instalar Git no Mac é via MacPorts⁴, para isso basta executar o seguinte comando:

```
sudo port install git-core
```

2.2 Configurando Perfil

As configurações vão determinar algumas opções globais do Git, sendo necessário fazê-las apenas uma vez.

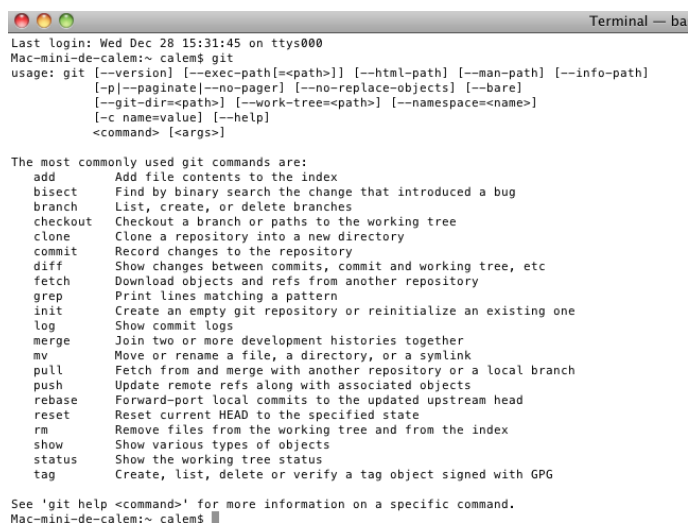
2.2.1 Usuário

Os comandos abaixo vão configurar o nome de usuário e endereço de e-mail. Esta informação é importante pois é anexada aos commits que você realiza,

²<https://git-scm.com/download/linux>

³<http://git-scm.com/downloads>

⁴<http://www.macports.org>



```

Last login: Wed Dec 28 15:31:45 on ttys000
Mac-mini-de-calem:~ calem$ git
usage: git [--version] [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
        [-p|--paginate|--no-pager] [--no-replace-objects] [--bare]
        [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
        [-c name=value] [--help]
        <command> [<args>]

The most commonly used git commands are:
add      Add file contents to the index
bisect   Find by binary search the change that introduced a bug
branch   List, create, or delete branches
checkout Checkout a branch or paths to the working tree
clone    Clone a repository into a new directory
commit   Record changes to the repository
diff     Show changes between commits, commit and working tree, etc
fetch    Download objects and refs from another repository
grep     Print lines matching a pattern
init     Create an empty git repository or reinitialize an existing one
log      Show commit logs
merge    Join two or more development histories together
mv       Move or rename a file, a directory, or a symlink
pull     Fetch from and merge with another repository or a local branch
push     Update remote refs along with associated objects
rebase   Forward-port local commits to the updated upstream head
reset    Reset current HEAD to the specified state
rm       Remove files from the working tree and from the index
show     Show various types of objects
status   Show the working tree status
tag      Create, list, delete or verify a tag object signed with GPG

See 'git help <command>' for more information on a specific command.
Mac-mini-de-calem:~ calem$

```

Figura 2.9: Printscreens do passo 9

ou seja, as configurações ficarão associadas ao trabalho em desenvolvimento, permitindo que os colaboradores/gestores do projeto identifiquem suas contribuições. Caso o projeto seja individual, a importância de configurar usuário e e-mail se mantém. Uma vez que se trabalha com duas ou mais máquinas, a maneira de identificar a pessoa que está desenvolvendo o trabalho é pelo nome de usuário.

Em um terminal Bash, execute o código abaixo:

```

git config --global user.name "Knight Rider"
git config --global user.email "batman@justiceleague.org"

```

A opção `--global` usará essa informação para todo projeto Git da máquina.

É possível fazer definições para cada projeto, ou seja, não globais. Para isso é necessário executar o comando a seguir sem a opção `--global`.

```

git config user.name "Knight Rider"
git config user.email "batman@justiceleague.org"

```

Uma vez configurado o perfil, o Git está pronto para uso.

2.2.2 Atalhos

Os atalhos no Git são chamados de *Aliases*. Com ele podemos mapear comandos que repetidamente usamos para algumas poucas teclas. Estes atalhos podem ser criados de dois modos: através do comando no terminal ou editando diretamente no arquivo `.gitconfig`.

Pelo terminal:

Execute o comando abaixo com o atalho de sua preferência e o nome completo do comando o qual deseja criar o alias.

```
git config --global alias.nome_do_alias "comando inteiro"
```

Um exemplo bem simples é o seguinte:

```
git config --global alias.st "status"
```

Assim, ao executar `git st` é o mesmo que executar `git status`.

Pelo método citado acima, o alias é adicionado automaticamente no seu arquivo `/.gitconfig`.

Pelo arquivo `/.gitconfig`:

Pode-se criar atalhos através de um bloco no seu arquivo de configuração. Para isso, é necessário localizar o diretório do Git e adicionar a lista de comandos desejada, como no exemplo:

```
[alias]
st = status
ci = commit
br = branch
co = checkout
df = diff
```

Assim que adicionar este bloco com os comandos de sua escolha, ele irá funcionar imediatamente.

Segue abaixo os caminhos para encontrar o arquivo `/.gitconfig` nos sistemas operacionais:

- Windows:
 1. `C:\\Pasta_do_seu_projeto\\.git\\config`
 2. `C:\\Documents and Settings\\Seu_usuario\\.gitconfig`
 3. `C:\\Arquivos de programas\\Git\\etc\\gitconfig`
- Mac:
 1. `/Pasta_do_seu_projeto/.git/config`
 2. `/Users/Seu_usuario/.gitconfig`
 3. `/usr/local/git/etc/gitconfig`
- Linux:
 1. Crie um arquivo como *sudo* dentro da pasta `etc/` com nome de `gitconfig` e coloque os atalhos de sua escolha.

Obs: Os arquivos de configuração do Git não tem extensão.

Não importa o método você utilize, suas configurações sempre ficarão salvas no arquivo `/.gitconfig`.

2.2.3 Ignorar Arquivos

Usando o arquivo `.gitignore` podemos ignorar arquivos que não desejamos versionar no repositório, pode ser feito por projeto e por usuário. Configurar um arquivo `.gitignore` antes de começar a trabalhar, é importante, pois evita commits acidentais de arquivos que não deveriam ir para o seu repositório Git.

Ignorar Arquivos por Projeto:

Em todos os projetos que necessitam de um controle de versão há sempre casos em que arquivos não precisam ser versionados. Para isso é preciso criar um arquivo `.gitignore` no diretório raiz do projeto, o qual contém padrões (pattern) que serão ignorados, cada padrão fica em uma linha como no exemplo:

```
$ cat .gitignore
*.oa
*~
```

A primeira linha fala para o Git ignorar qualquer arquivo finalizado em `.o` ou `.a` e a segunda linha ignora todos os arquivos que terminam com um til (`~`). Esses padrões podem serem feitos de acordo com a necessidade de cada projeto.

Ignorar Arquivos por Usuário (Globalmente):

Para não precisar criar uma lista de comandos para serem ignorados em cada projeto, é possível ignorar arquivos em todos os repositórios. Para isso, basta criar um arquivo `.gitignore` em seu diretório *home* contendo os padrões os quais deseja ignorar e executar o comando abaixo no terminal a partir da pasta onde está localizado o arquivo `.gitignore`:

```
git config --global core.excludesfile ~/.gitignore
```

A partir disso, todos os arquivos que estão na lista serão ignorados pelo usuário.

Finalmente com a instalação, configuração essencial (usuário e e-mail) e configurações adicionais concluídas, podemos começar a utilizar o Git para versionar nossos projetos.

Capítulo 3

Repositórios Locais

3.1 Instruções do Git

Neste capítulo, as instruções serão todas feitas no terminal mesmo que existam alternativas gráficas para as mesmas. Isso enfatiza no que está sendo feito além do fato de que no terminal todos devem ter os mesmos recursos e os comandos irão produzir os mesmos resultados, o que faz esse tutorial algo reproduzível. Casos você queira usufruir das ferramentas gráficas vá para o capítulo 6.

Sabendo que você executou os comandos de perfil que no capítulo anterior, temos o Git devidamente configurado, com credenciais (nome e e-mail) e configurações aplicadas. Vamos então ver como o sistema de controle de versão acontece.

```
## Apenas verificando o cadastro.  
git config user.name "Knight Rider"  
git config user.email "batman@justiceleague.org"
```

Todas as instruções do Git são sinalizadas por começar com git seguido da instrução/comando e seus argumentos complementares, se existirem/necessários.

```
## Padrão de instruções Git.  
# git <instrução> <complementos...>
```

```
cd meulrepo ## Diretório de teste de comandos
```

```
## Padrão de instruções Git.  
git <instrução> <complementos ...>
```

Os comandos abaixo revelam tudo o Git possui, embora dizer o que ele tem não signifique nada diante do que ele pode fazer com o que tem.

```
## Ajuda resumida do Git, principais comandos com descrição.
```

```
git help -a
```

```
usage: git [--version] [--help] [-C <path>] [-c name=value]
        [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
        [-p | --paginate | --no-pager] [--no-replace-objects] [--bare]
        [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
        <command> [<args>]
```

```
available git commands in '/usr/lib/git-core'
```

add	merge-octopus
add--interactive	merge-one-file
am	merge-ours
annotate	merge-recursive
apply	merge-resolve
archive	merge-subtree
bisect	merge-tree
bisect--helper	mergetool
blame	mktag
branch	mtree
bundle	mv
cat-file	name-rev
check-attr	notes
check-ignore	pack-objects
check-mailmap	pack-redundant
check-ref-format	pack-refs
checkout	patch-id
checkout-index	prune
cherry	prune-packed
cherry-pick	pull
citool	push
clean	quiltimport
clone	read-tree
column	rebase
commit	receive-pack
commit-tree	reflog
config	relink
count-objects	remote
credential	remote-ext
credential-cache	remote-fd
credential-cache--daemon	remote-ftp
credential-store	remote-ftps
daemon	remote-http
describe	remote-https
diff	remote-testsvn
diff-files	repack
diff-index	replace
diff-tree	request-pull

difftool	rerere
difftool--helper	reset
fast-export	rev-list
fast-import	rev-parse
fetch	revert
fetch-pack	rm
filter-branch	send-pack
fmt-merge-msg	sh-i18n--envsubst
for-each-ref	shell
format-patch	shortlog
fsck	show
fsck-objects	show-branch
gc	show-index
get-tar-commit-id	show-ref
grep	stage
gui	stash
gui--askpass	status
hash-object	strip-space
help	submodule
http-backend	subtree
http-fetch	symbolic-ref
http-push	tag
imap-send	unpack-file
index-pack	unpack-objects
init	update-index
init-db	update-ref
instaweb	update-server-info
interpret-trailers	upload-archive
log	upload-pack
ls-files	var
ls-remote	verify-commit
ls-tree	verify-pack
mailinfo	verify-tag
mailsplit	web--browse
merge	whatchanged
merge-base	worktree
merge-file	write-tree
merge-index	

git commands available from elsewhere on your \$PATH

cola dag

'git help -a' and 'git help -g' list available subcommands and some concept guides. See 'git help <command>' or 'git help <concept>' to read about a specific subcommand or concept.

3.2 Meu Primeiro Repositório

```
git init
```

Initialized empty Git repository in /home/walmes/GitLab/apostila-git/meulrepo/.git/

O Git retorna a mensagem de inicialização do repositório. Nesse momento ele cria um diretório oculto `.git/` com subdiretórios que são o coração do sistema de versionamento. Você não deve modificar nada nesse diretório. É por essa razão que ele é oculto. Alterar o conteúdo pode prejudicar ou interromper o funcionamento do Git. Se você quiser encerrar o processo de versionamento fazendo com que esse diretório seja como qualquer outro diretório, é só excluir a diretório `.git/`. Cada subdiretório do `.git/` tem um propósito mas deixaremos os esclarecimentos para o futuro. Por agora vamos apenas conferir a sua estrutura.

```
## Mostra todo conteúdo do diretório.  
tree --charset=ascii -a
```

```
.  
|-- .git  
|   |-- branches  
|   |-- config  
|   |-- description  
|   |-- HEAD  
|   |-- hooks  
|       |-- applypatch-msg.sample  
|       |-- commit-msg.sample  
|       |-- post-update.sample  
|       |-- pre-applypatch.sample  
|       |-- pre-commit.sample  
|       |-- prepare-commit-msg.sample  
|       |-- pre-push.sample  
|       |-- pre-rebase.sample  
|       |-- update.sample  
|   |-- info  
|       |-- exclude  
|   |-- objects  
|       |-- info  
|       |-- pack  
|-- refs  
    |-- heads  
    |-- tags
```

10 directories, 13 files

NOTA: o `tree` é um programa instalado a parte (*third party software*) que retorna arte ASCII representado a estrutura de diretórios. Se você usa distribuição Debian, instale com `sudo apt-get install tree`. Windows: [tree](#).

Vamos começar da maneira mais simples: criando um arquivo com uma linha de texto apenas. Bem, vale avisar que ao longo desse capítulo, os arquivos serão sempre bem pequenos e dificilmente realistas, mas como o enfoque está no funcionamento, não haverá prejuízo.

Vamos criar o arquivo com conteúdo também pelo terminal. Se você preferir, abra eu editor de texto favorito (Emacs, Gedit, Geany, RStudio, Bloco de Notas, Notepad++, etc) e faça algo mais criativo.

```
## Cria um arquivo com uma linha de conteúdo.  
echo "Meu primeiro repositório Git" > README.txt
```

```
## Lista os arquivos do diretório.  
## tree --charset=ascii
```

```
## Reconhecimento do Git sobre arquivo criado.  
git status
```

On branch master

Initial commit

Untracked files:

(use "git add <file>..." to include in what will be committed)

README.txt

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

E o que o Git “acha” de tudo isso? O comando *status* é o mais usado do Git, principalmente nas etapas de aprendizado. Uma característica diferente do Git, se comparado a outras aplicações de uso por terminal, é que ele é realmente camarada. Nas mensagens de *output*, o Git informa o que aconteceu e também sugere o que fazer.

Este diretório agora é um diretório sob versionamento Git, portanto todas as alterações realizadas serão observadas pelo sistema. Ao criarmos o arquivo e pedirmos a situação (*status*), o Git indica que existe um arquivo não rastreado (*untracked*) no diretório. Inclusive sugere uma ação que seria adicionar o arquivo (*add*). Se o seu sistema operacional está em português, parte dos outputs do Git podem estar traduzidos.

De forma geral temos 3 estágios 3.1 de arquivos considerados no sistema de controle de versionamento Git. São eles *working directory*, *Staged Area* e *Committed*, os discutiremos ao longo do texto. Todo arquivo criado em um diretório versionado deve necessariamente passar pelos três estágios. Voltando para a nossa situação temos o arquivo README.txt criado e atualmente ele está no estágio *working directory*, faremos todo o procedimento para que chegue ao estágio *committed*.

Alterações em arquivos no *working directory* não são armazenadas, por isso o sugestivo nome “diretório de trabalho”. Portanto, para que o arquivo seja

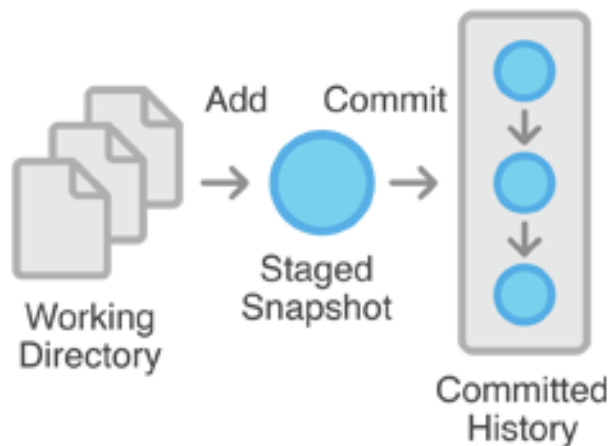


Figura 3.1: Passos para versionamento

incluído no monitoramento é necessário que ele receba o primeiro comando *add*. Isso marca a entrada dele no projeto como um arquivo que a partir de então será versionado.

Para que o arquivo seja incluído no monitoramento é necessário que ele receba o primeiro comando *add*. Isso marca a entrada dele no projeto como um arquivo que a partir de então será versionado. O *status* agora não indica mais que ele está *untracked* mas sim que existem mudanças para serem registradas (*changes to be committed*). A melhor tradução de *commit*, pensando no seu uso em Git, é **fechar sob segurança**. Quando um *commit* é feito, cria-se um instante na linha do tempo que salva o estado do projeto. Para esse instante o projeto pode ser retrocedido, voltando o condição/conteúdo de todos os arquivos para o momento no qual o mencionado *commit* foi feito. Você pode voltar para um *commit* de semanas e até anos atrás.

O controle de versão não é apenas voltar os arquivos para o conteúdo que eles tinham no passado. Arquivos rastreados que foram deletados ou renomeados são recuperados. Até mesmo as permissões de leitura/escrita/execução dos arquivos são contempladas no versionamento.

```
## O primeiro `add` submete o arquivo ao versionamento.
git add README.txt
git status
```

On branch master

Initial commit

Changes to be committed:

(use "git rm --cached <file>..." to unstage)

new file: README.txt

O arquivo `README.txt` já é visto pelo Git como um arquivo com o qual ele deve se “preocupar”, pois está sob versionamento. Vamos agora fazer um registro definitivo sobre o estado desse arquivo (*commit*). É de fundamental importância que a mensagem de notificação, ou mensagem de *commit*, reflita as modificações feitas. São as mensagens que serão consultadas quando você precisar desfazer/voltar. Ela deve ser curta (≤ 72 caracteres) e ao mesmo tempo informativa. A minha primeira mensagem não será, todavia.

Boas Práticas de *commit*:

- Verbo no indicativo
- Frases curtas
- Dizer o que fez e não como fez

Evite mensagens de *commit* como:

- "Modificações realizadas"
- "Trabalhei muito hoje"
- "Terminando este trabalho na madrugada"

```
## Registro de versão.  
git commit -m 'Cria arquivo com título'
```

```
[master (root-commit) 612c338] Cria arquivo com título  
1 file changed, 1 insertion(+)  
create mode 100644 README.txt
```

O retorno da instrução de *commit* indica o número de arquivos incluídos no *commit* e o número de inserções e deleções de linhas. O mais importante está na primeira linha que informa o ramo de trabalho atual (*branch*) e o *sha1* do *commit*. O *sha1* é uma sequência hexadecimal de 40 dígitos que representa unicamente o *commit*, então são 16^{40} possibilidades. É por meio do *sha1* que podemos retroceder o projeto. São mostrados apenas os 7 primeiros dígitos porque são suficientes para diferenciar *commits*, seja de projetos pequenos ou até mesmo de projetos moderados ou grandes.

3.3 Complementos

Para o registro de desenvolvimento, existe marcação por tags, que seriam commits especiais, geralmente usado para marcar pontos de destaque do projeto, por exemplo versão alfa, beta, teste servidor.

```
# Criando a tag
git tag versao2

# Podemos marcar uma tag com um commit
git tag -a versao2 -m "Versão 2 está pronta"

# Para ver todas as tags
git tag

# Excluir tags
git tag -d versao1
```

Caso haja mudança no nome do arquivo que você esteja versionado, deve ser alterado pelo próprio Git, para que fique no atual estágio de versionamento.

```
## git mv antigo novo
git mv -f README.txt LEIAME.txt
```

Caso você queira excluir o arquivo

```
git rm README.txt
```

3.4 Versões de Arquivos Diferentes

Vamos criar mais arquivos e acrescentar conteúdo ao já rastreado pelo Git para percebermos o funcionamento. Escrever uma lista de razões para usar o Linux. Deixei a lista curta poder ampliar no futuro e com erros de português para corrigir depois.

```
## Adiciona mais linhas ao README.txt
echo "
A filosofia do Linux é 'Ria na face do perigo'.
Ôpa. Errado. 'Faça você mesmo'. É, é essa.
    -- Lunus Torvalds" >> README.txt

## Cria uma lista de pontos sobre o porquê de usar o Linux.
echo "Por que usar o Linux?

* É livre
* É seguro
* É customizavel" > porqueLinux.txt

## Mostra o conteúdo do arquivo.
less README.txt
```

Meu primeiro repositório Git

A filosofia do Linux é 'Ria na face do perigo'.
Ôpa. Errado. 'Faça você mesmo'. É, é essa.
-- Lunus Torvalds

```
## Mostra o conteúdo do arquivo.  
less porqueLinux.txt
```

Por que usar o Linux?

- * É livre
- * É seguro
- * É customizável

```
git status
```

```
On branch master  
Changes not staged for commit:  
  (use "git add <file>..." to update what will be committed)  
  (use "git checkout -- <file>..." to discard changes in working directory)
```

```
    modified:   README.txt
```

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

```
    porqueLinux.txt
```

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

O Git retornou dois campos. No primeiro ele diz que existem mudanças no README.txt não encaminhadas para receber registro (*not staged for commit*) e no segundo ele aponta que o porqueLinux.txt é um arquivo não rastreado (fora de monitoramento).

Vamos explorar um pouco mais os recursos do Git antes de adicionar as recentes mudanças. Até o momento, temos apenas um *commit* feito. Para acessar o histórico de registros usamos `git log`. O histórico mostra o *sha1*, o autor, data e mensagem de *commit*. Uma saída mais enxuta é obtida com `git log --oneline`, útil quando o histórico é longo. É possível fazer restrição no `git log`, como mostrar os *commits* a partir de certa data, certo intervalo de datas, para um único autor ou único arquivo.

```
## Mostra o histórico de commits.  
git log
```

```
commit 612c338a6480db837dcef86df149e06c90ed6ded  
Author: Walmes Zeviani <walmes@ufpr.br>
```

Date: Thu Dec 17 09:29:02 2015 -0200

Cria arquivo com título

O comando *diff* mostra as diferenças no conteúdo dos arquivos/diretório. No caso, apenas o README.txt está sendo rastreado, por isso o *output* indica a adição (+) de novas linhas. Na saída tem-se os *sha1* das versões comparadas e o intervalo de linhas envolvido na porção modificada (@@). Visite: [git-diffs](#).

```
## Diferença nos arquivos versionados.
git diff
```

```
diff --git a/README.txt b/README.txt
index 07d3585..d0af1d3 100644
--- a/README.txt
+++ b/README.txt
@@ -1,5 @@
  Meu primeiro repositório Git
+
+ A filosofia do Linux é 'Ria na face do perigo'.
+ Ôpa. Errado. 'Faça você mesmo'. É, é essa.
+
+ -- Lunus Torvalds
```

Vamos aplicar o primeiro *add* ao porqueLinux.txt para que ele comece a ser versionado. Perceba que ao adicioná-lo, as mudanças, no caso a criação do arquivo com conteúdo, já são separadas para receber registro (*changes to be committed*).

```
## Adiciona o arquivo ao processo de reastreamento.
git add porqueLinux.txt
git status
```

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: porqueLinux.txt

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: README.txt

```
## Mensagem que registra as modificações adicionadas.
git commit -m "Lista de inicial de o porquê usar o Linux."
```

```
[master 8acac99] Lista de inicial de o porquê usar o Linux.
1 file changed, 5 insertions(+)
create mode 100644 porqueLinux.txt
```

```
git status
```

```
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   README.txt

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

Ainda resta o README.txt para receber registro. Você não precisa fazer agora. Pode continuar editando caso não tenha atingido uma quantidade de modificação merecedora de *commit*. Lembre-se que registros geram conteúdo no diretório .git. Quanto mais *commits*, mais conteúdo gerado. Mas também, toda vez que você faz um *commit*, você define um ponto de retorno, um estado salvo, caso precise no futuro recuperar/visitar. O que é uma unidade de modificação “comitável” você irá definir aos poucos com a prática.

```
## Encaminha o arquivo para receber registro.
```

```
git add README.txt
git status
```

```
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   README.txt
```

```
## Atribui mensagem de notificação.
```

```
git commit -m "Adiciona frase do Linux Torvalds."
```

```
[master 1aa33c3] Adiciona frase do Linux Torvalds.
1 file changed, 4 insertions(+)
```

Agora temos 3 *commits* e o *log* apresenta os *sha1* e as mensagens correspondentes aos mesmos. Com `--oneline` resumimos o *output* mostrando apenas o *sha1* e a mensagem de *commit*.

```
git log --oneline
```

```
1aa33c3 Adiciona frase do Linux Torvalds.
8acac99 Lista de inicial de o porquê usar o Linux.
612c338 Cria arquivo com título
```

Por meio dos *sha1*, podemos aplicar o *diff* para visitarmos as modificações entre dois *commits*, não necessariamente consecutivos, por exemplo. Também podemos retroceder (*checkout*, *reset*, *revert*) o projeto para alguns desses pontos.

```
git diff HEAD@{1}
```

```
diff --git a/README.txt b/README.txt
index 07d3585..d0af1d3 100644
--- a/README.txt
+++ b/README.txt
@@ -1,5 @@
  Meu primeiro repositório Git
+
+A filosofia do Linux é 'Ria na face do perigo'.
+Ôpa. Errado. 'Faça você mesmo'. É, é essa.
+
-- Lunus Torvalds
```

Instruímos o Git mostrar as diferenças para um *commit* atrás. A cabeça (*HEAD*) é o que se entende como estado mais recente. Usa-se o termo *HEAD* (cabeça) pois considera-se um crescimento do histórico debaixo para cima no qual um novo *commit* é colocado em cima dos demais (empilhado). O *HEAD@{0}* ou apenas *HEAD* representa o último registro feito. Não é necessário escrever o último *HEAD* na instrução abaixo.

Agora inspecionado uma distância de 2 *commits* a partir do último. Aqui tem-se os dois arquivos envolvidos nesse intervalo de 2 *commits*. Com *--name-only* retorna-se só o nome dos arquivos.

```
git diff HEAD@{2} HEAD@{0}
```

```
diff --git a/README.txt b/README.txt
index 07d3585..d0af1d3 100644
--- a/README.txt
+++ b/README.txt
@@ -1,5 @@
  Meu primeiro repositório Git
+
+A filosofia do Linux é 'Ria na face do perigo'.
+Ôpa. Errado. 'Faça você mesmo'. É, é essa.
+
-- Lunus Torvalds
diff --git a/porqueLinux.txt b/porqueLinux.txt
new file mode 100644
index 0000000..8ecd3da
--- /dev/null
+++ b/porqueLinux.txt
@@ -0,0 +1,5 @@
+Por que usar o Linux?
+
+* É livre
```



```
++ É seguro
++ É customizavel
```

```
git diff --name-only HEAD@{2} HEAD@{0}
```

```
README.txt
porqueLinux.txt
```

Vamos resolver logo o caso da palavra sem acento em `porqueLinux.txt`. Você abre o arquivo no seu editor de texto e modifica conforme necessário. A modificação compreende um linha apenas mas aí lembrei de mais coisas e acrescentei. O `git diff` mostra as diferenças. Epa! As diferenças não eram entre *commits*? O conteúdo adicionado ainda não recebeu notificação!

```
## Depois de corrigir palavras e adicionar conteúdo.
git status
```

```
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   porqueLinux.txt

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

O Git sugere você aplicar *add* para preparar para *commit*. Caso as modificações sejam um engano e não mais desejadas, você pode desfazê-las, abandonar a correção/inclusão das palavras usando *checkout*. Vamos aplicá-lo para ver como funciona.

```
## Palavras corrigidas e mais itens adicionados.
less porqueLinux.txt
```

Por que usar o Linux?

```
* É livre
* É seguro
* É customizável
* Tem repositórios de software
* Atualização constante
* Desempenho
```

```
## Abandona modificações feitas presentes no arquivo.
git checkout -- porqueLinux.txt
```

```
less porqueLinux.txt
```

Por que usar o Linux?

- * É livre
- * É seguro
- * É customizavel

Bateu o arrependimento? Tem formas de poder retroceder com mudanças ainda não registradas mas mantendo a possibilidade de recuperá-las. Mostraremos em breve.

NOTA: sempre que quiser testar um comando novo e não está seguro do que ele faz ou da extensão dos seus efeitos, faça uma cópia do projeto em outro diretório e experimente ele lá. Isso previne sabores amargos, pois algumas ações podem ser irreversíveis.

```
## Depois de desfazer as modificações no porqueLinux.txt
git status
```

```
On branch master
nothing to commit, working directory clean
```

Vamos seguir com as modificações em porqueLinux.txt que corrigem o texto e acrescentam itens novos.

```
git status
```

```
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
```

```
    modified:   porqueLinux.txt
```

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

O diff vazio compara o diretório de trabalho com o último registro (último *commit*). Quando você usa explicitamente na instrução `HEAD@{ }` seguido de número, então estão sendo comparadas versões “commitadas”. Existem variantes de sufixo para usar no `HEAD` que são:

- `HEAD@{n}`
- `HEAD^n`
- `HEAD~n`

em que n é um número inteiro não negativo. Cada sufixo tem uma finalidade que não detalharemos agora. Visite: [git-caret-and-tilde](#).

```
## Modificações no diretório vs último commit.
git diff
```

```
diff --git a/porqueLinux.txt b/porqueLinux.txt
index 8ecdfda..8747d1e 100644
--- a/porqueLinux.txt
+++ b/porqueLinux.txt
@@ -2,4 +2,7 @@ Por que usar o Linux?

* É livre
* É seguro
-* É customizavel
+* É customizável
+* Tem repositórios de software
+* Atualização constante
+* Desempenho
```

```
## Último commit vs dois ancestrais, usando ~.
git diff HEAD~1 HEAD~0
```

```
diff --git a/README.txt b/README.txt
index 07d3585..d0af1d3 100644
--- a/README.txt
+++ b/README.txt
@@ -1 +1,5 @@
Meu primeiro repositório Git
+
+A filosofia do Linux é 'Ria na face do perigo'.
+Ôpa. Errado. 'Faça você mesmo'. É, é essa.
+
-- Lunus Torvalds
```

```
## Último commit vs seu ancestral, usando @{}.
git diff HEAD@{1} HEAD@{0}
```

```
diff --git a/README.txt b/README.txt
index 07d3585..d0af1d3 100644
--- a/README.txt
+++ b/README.txt
@@ -1 +1,5 @@
Meu primeiro repositório Git
+
+A filosofia do Linux é 'Ria na face do perigo'.
+Ôpa. Errado. 'Faça você mesmo'. É, é essa.
+
-- Lunus Torvalds
```

```
## Último commit vs dois ancestrais.
## git diff HEAD~2 HEAD~0
git diff HEAD@{2} HEAD@{0}
```

```
diff --git a/README.txt b/README.txt
index 07d3585..d0af1d3 100644
--- a/README.txt
+++ b/README.txt
@@ -1,5 @@
   Meu primeiro repositório Git
+
+A filosofia do Linux é 'Ria na face do perigo'.
+Ôpa. Errado. 'Faça você mesmo'. É, é essa.
+                                -- Lunus Torvalds
diff --git a/porqueLinux.txt b/porqueLinux.txt
new file mode 100644
index 0000000..8ecdafa
--- /dev/null
+++ b/porqueLinux.txt
@@ -0,0 +1,5 @@
+Por que usar o Linux?
+
+* É livre
+* É seguro
+* É customizavel
```

Para ficar claro daqui em diante, você pode ao invés de pedir log, pedir o reflog que inclui as referências em termos da sequência do mais recente para os seus ancestrais.

```
## Mostra referências para commits os ancestrais.
git reflog
```

```
1aa33c3 HEAD@{0}: commit: Adiciona frase do Linux Torvalds.
8acac99 HEAD@{1}: commit: Lista de inicial de o porquê usar o Linux.
612c338 HEAD@{2}: commit (initial): Cria arquivo com título
```

```
## Adiciona e commita.
git add porqueLinux.txt
git commit -m "Novos argumentos."
```

```
[master 095046c] Novos argumentos.
1 file changed, 4 insertions(+), 1 deletion(-)
```

O Git permite um nível de rastreabilidade bem fino. Veja por exemplo que é possível saber quem modificou e quando cada linha do arquivo e qual o correspondente *sha1* do commit.


```
## Novo ramo criado aparece.
git branch
```

```
feature01
* master
```

```
## Muda o HEAD de ramo.
git checkout feature01
```

```
Switched to branch 'feature01'
```

```
## Situação no novo ramo.
git status
```

```
On branch feature01
nothing to commit, working directory clean
```

```
## Histórico de commits.
git log --oneline
```

```
095046c Novos argumentos.
1aa33c3 Adiciona frase do Linux Torvalds.
8acac99 Lista de inicial de o porquê usar o Linux.
612c338 Cria arquivo com título
```

Veja que o novo ramo não começa no zero ou vazio (sem arquivos) e sim a partir do ramo que é seu ancestral, ou seja, ele começa a partir do último *commit* existente, por padrão. Vamos adicionar um arquivo e commitar. O comando `wget` permite baixar arquivos pelo terminal. Será baixado um arquivo com um função para calcular o fator de inflação de variância (*vif*, *variance inflation factor*) usado em modelos de regressão, disponível na página da Professora Suely Giolo¹.

```
## Baixando arquivo da internet. Uma função do R.
wget 'http://people.ufpr.br/~giolo/CE071/Exemplos/vif.R'
```

```
--2015-12-17 09:29:03-- http://people.ufpr.br/~giolo/CE071/Exemplos/vif.R
Resolving people.ufpr.br (people.ufpr.br)... ????.???.???.??, 2801:82:8020:0:8377:0:100:2
Connecting to people.ufpr.br (people.ufpr.br)|???.???.???.??|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 560
Saving to: 'vif.R'
```

```
OK
```

```
100% 44,0M=0s
```

```
2015-12-17 09:29:03 (44,0 MB/s) - 'vif.R' saved [560/560]
```

¹<http://people.ufpr.br/~giolo/CE071/Exemplos/vif.R>

```
## Situação do repositório após o download.
```

```
git status
```

```
On branch feature01
```

```
Untracked files:
```

```
(use "git add <file>..." to include in what will be committed)
```

```
vif.R
```

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

```
git add vif.R
```

```
git commit -m "Adiciona função R para VIF."
```

```
[feature01 826f940] Adiciona função R para VIF.
```

```
1 file changed, 20 insertions(+)
```

```
create mode 100644 vif.R
```

```
## Estrutura do diretório.
```

```
tree --charset=ascii
```

```
.
|-- porqueLinux.txt
|-- README.txt
`-- vif.R
```

```
0 directories, 3 files
```

```
## Histórico de commits.
```

```
git reflog
```

```
826f940 HEAD@{0}: commit: Adiciona função R para VIF.
```

```
095046c HEAD@{1}: checkout: moving from master to feature01
```

```
095046c HEAD@{2}: commit: Novos argumentos.
```

```
1aa33c3 HEAD@{3}: commit: Adiciona frase do Linux Torvalds.
```

```
8acac99 HEAD@{4}: commit: Lista de inicial de o porquê usar o Linux.
```

```
612c338 HEAD@{5}: commit (initial): Cria arquivo com título
```

```
git status
```

```
On branch feature01
```

```
nothing to commit, working directory clean
```

Então acabamos de acrescentar um novo arquivo ao projeto. Agora que as modificações foram salvas (*commit* feito) podemos trocar de ramo. Você vai ver que ao voltarmos para o ramo *master* o arquivo baixado da internet vai “desaparecer”.

```
## Volta para o ramo master.
git checkout master
```

Switched to branch 'master'

```
## Estrutura do diretório.
tree --charset=ascii
```

```
.
|-- porqueLinux.txt
`-- README.txt
```

0 directories, 2 files

O arquivo vif.R não sumiu. Ele está no ramo feature01 e só passará para o ramo master quando mesclarmos o que existe no feature01 ao master. Então é assim: mudou de ramo, muda o conteúdo do diretório. Fazer isso é bem simples, basta dar um `git merge`. Antes vamos aprender como saber as diferenças que existem entre ramos.

```
## Mostra todas as modificações, cada linha modificada de cada arquivo.
git diff master feature01
```

```
diff --git a/vif.R b/vif.R
new file mode 100644
index 0000000..a114969
--- /dev/null
+++ b/vif.R
@@ -0,0 +1,20 @@
+vif<-function (obj, digits = 5) {
+  Qr <- obj$qr
+  if (is.null(obj$terms) || is.null(Qr))
+    stop("invalid 'lm' object: no terms or qr component")
+  tt <- terms(obj)
+  hasintercept <- attr(tt, "intercept") > 0
+  p <- Qr$rank
+  if (hasintercept)
+    p1 <- 2:p
+  else p1 <- 1:p
+  R <- Qr$qr[p1, p1, drop = FALSE]
+  if (length(p1) > 1)
+    R[row(R) > col(R)] <- 0
+  Rinv <- qr.solve(R)
+  vv <- apply(Rinv, 1, function(x) sum(x^2))
+  ss <- apply(R, 2, function(x) sum(x^2))
+  vif <- ss * vv
+  signif(vif, digits)
+}
```



```
## Mostra só os arquivos modificados.
git diff --name-only master feature01
```

```
vif.R
```

Como você já havia antecipado, a única diferença entre os ramos master e feature01 é o arquivo vif.R. Agora é só pedir o git merge.

```
## Mesclando as modificações em feature01 no master.
git merge feature01 master
```

```
Updating 095046c..826f940
Fast-forward
 vif.R | 20 ++++++++++++++++++++++
 1 file changed, 20 insertions(+)
 create mode 100644 vif.R
```

```
git log --oneline
```

```
826f940 Adiciona função R para VIF.
095046c Novos argumentos.
1aa33c3 Adiciona frase do Linux Torvalds.
8acac99 Lista de inicial de o porquê usar o Linux.
612c338 Cria arquivo com título
```

É possível criar um ramo a partir de um *commit* ancestral ao atual. Isso é extremamente útil para resolver os bugs. Vamos fazer um segundo ramo a partir do *commit* anterior a inclusão do arquivo R.

```
## Referencias aos commits.
git reflog
```

```
826f940 HEAD@{0}: merge feature01: Fast-forward
095046c HEAD@{1}: checkout: moving from feature01 to master
826f940 HEAD@{2}: commit: Adiciona função R para VIF.
095046c HEAD@{3}: checkout: moving from master to feature01
095046c HEAD@{4}: commit: Novos argumentos.
1aa33c3 HEAD@{5}: commit: Adiciona frase do Linux Torvalds.
8acac99 HEAD@{6}: commit: Lista de inicial de o porquê usar o Linux.
612c338 HEAD@{7}: commit (initial): Cria arquivo com título
```

```
## Volta para antes do arquivo de baixar o vif.R.
git checkout HEAD@{4}
```

Note: checking out 'HEAD@{4}'.

You are in 'detached HEAD' state. You can look around, make experimental changes and commit them, and you can discard any commits you make in this state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may do so (now or later) by using `-b` with the checkout command again. Example:

```
git checkout -b <new-branch-name>
```

HEAD is now at 095046c... Novos argumentos.

```
## Qual a situação.  
git status
```

```
HEAD detached at 095046c  
nothing to commit, working directory clean
```

```
## 0 histórico existente nesse ponto.  
git log --name-only --oneline
```

```
095046c Novos argumentos.  
porqueLinux.txt  
1aa33c3 Adiciona frase do Linux Torvalds.  
README.txt  
8acac99 Lista de inicial de o porquê usar o Linux.  
porqueLinux.txt  
612c338 Cria arquivo com título  
README.txt
```

Já que o *commit* mais recente dessa história aponta para o arquivo `compras`, vamos checar o seu conteúdo apenas por medida de segurança.

```
## Mostra o conteúdo do arquivo.  
less porqueLinux.txt
```

Por que usar o Linux?

- * É livre
- * É seguro
- * É customizável
- * Tem repositórios de software
- * Atualização constante
- * Desempenho

Dito e feito! Voltamos no tempo para o instante logo após o *commit* que incluí novos itens na lista. Podemos voltar para o presente se estivermos

satisfeitos com o passeio mas também podemos criar um ramo aqui, caso isso seja necessário. Primeiro vamos voltar para o presente depois mostramos como criar o ramo.

```
## Mostra onde estamos.  
git branch
```

```
* (HEAD detached at 095046c)  
  feature01  
  master
```

Se não fizemos nenhuma modificação, voltar é bem simples. Se modificações foram feitas é necessário saber se precisam ser mantidas e onde ou se devem ser descartadas.

```
## Volta para o presente.  
git checkout master
```

```
Previous HEAD position was 095046c... Novos argumentos.  
Switched to branch 'master'
```

```
git status
```

```
On branch master  
nothing to commit, working directory clean
```

```
git log --oneline
```

```
826f940 Adiciona função R para VIF.  
095046c Novos argumentos.  
1aa33c3 Adiciona frase do Linux Torvalds.  
8acac99 Lista de inicial de o porquê usar o Linux.  
612c338 Cria arquivo com título
```

```
## Fenda no tempo fechada. Sem sinal do detached HEAD.  
git branch
```

```
  feature01  
* master
```

```
## Sinal do passeio no tempo.  
git reflog
```

```
826f940 HEAD@{0}: checkout: moving from 095046c3927fe5fa9932d9d3d858e3128126b4fc to master  
095046c HEAD@{1}: checkout: moving from master to HEAD@{4}
```

```
826f940 HEAD@{2}: merge feature01: Fast-forward
095046c HEAD@{3}: checkout: moving from feature01 to master
826f940 HEAD@{4}: commit: Adiciona função R para VIF.
095046c HEAD@{5}: checkout: moving from master to feature01
095046c HEAD@{6}: commit: Novos argumentos.
1aa33c3 HEAD@{7}: commit: Adiciona frase do Linux Torvalds.
8acac99 HEAD@{8}: commit: Lista de inicial de o porquê usar o Linux.
612c338 HEAD@{9}: commit (initial): Cria arquivo com título
```

Vamos começar a ser ousados. Vamos voltar no passado, adicionar um arquivo, commitar e ver o que acontece, como o histórico é representado.

```
## Volta no tempo, após commit que aumenta porqueLinux.txt.
git checkout HEAD@{6}
```

Note: checking out 'HEAD@{6}'.

You are in 'detached HEAD' state. You can look around, make experimental changes and commit them, and you can discard any commits you make in this state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may do so (now or later) by using `-b` with the checkout command again. Example:

```
git checkout -b <new-branch-name>
```

HEAD is now at 095046c... Novos argumentos.

```
## Baixa arquivo de dados da internet.
wget 'http://www.leg.ufpr.br/~walmes/data/pimentel_racoes.txt'
```

```
--(date +"%Y-%m-%d %H:%M:%S")-- http://www.leg.ufpr.br/~walmes/data/pimentel_racoes.tx
Resolving www.leg.ufpr.br (www.leg.ufpr.br)... ????.???.????.??
Connecting to www.leg.ufpr.br (www.leg.ufpr.br)|????.???.????.??|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 217 [text/plain]
Saving to: 'pimentel_racoes.txt'
```

OK

100% 68,9M=0s

```
(date +"%Y-%m-%d %H:%M:%S") (68,9 MB/s) - 'pimentel_racoes.txt' saved [217/217]
```

```
'pimentel_racoes.txt' -> '../meulrepo/pimentel_racoes.txt'
```

```
git status
```

HEAD detached at 095046c

Untracked files:

(use "git add <file>..." to include in what will be committed)

pimentel_racoes.txt

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

Adiciona para registrar a inclusão do arquivo.

```
git add pimentel_racoes.txt
```

```
git commit -m "Adiciona arquivo de dados de experimento com rações."
```

```
[detached HEAD f26d9b4] Adiciona arquivo de dados de experimento com rações.  
1 file changed, 24 insertions(+)  
create mode 100644 pimentel_racoes.txt
```

```
git status
```

```
HEAD detached from 095046c  
nothing to commit, working directory clean
```

Log num formato gráfico ASCII para mostrar o novo ramo.

```
git log --graph --oneline --decorate --date=relative --all
```

```
* 826f940 (master, feature01) Adiciona função R para VIF.  
| * f26d9b4 (HEAD) Adiciona arquivo de dados de experimento com rações.  
|/  
* 095046c Novos argumentos.  
* 1aa33c3 Adiciona frase do Linux Torvalds.  
* 8acac99 Lista de inicial de o porquê usar o Linux.  
* 612c338 Cria arquivo com título
```

No nosso projeto temos o *master* e o *feature01* em igual condição, sem nenhuma diferença. O *commit* recém feito indica um novo seguimento a partir daquele onde adicionamos novos itens na lista. Vamos criar um novo ramo porque atualmente estamos em um ramos suspenso (*detached HEAD*) que não é persistente.

```
git branch
```

```
* (HEAD detached from 095046c)  
feature01  
master
```

Um novo ramo a partir do atual HEAD.

```
git checkout -b feature02
```

Switched to a new branch 'feature02'

```
git branch
```

```
feature01
* feature02
master
```

```
git log --graph --oneline --decorate --date=relative --all
```

```
* 826f940 (master, feature01) Adiciona função R para VIF.
| * f26d9b4 (HEAD -> feature02) Adiciona arquivo de dados de experimento com rações.
|/
* 095046c Novos argumentos.
* 1aa33c3 Adiciona frase do Linux Torvalds.
* 8acac99 Lista de inicial de o porquê usar o Linux.
* 612c338 Cria arquivo com título
```

Vamos explorar bem a funcionalidade. Vamos voltar para o feature01 e criar mais coisas lá. Você já deve estar pensando que tudo isso é absurdo e jamais alguém ficaria indo e voltando assim. Você está certo, porém, quando o projeto envolve mais pessoas, certamente as coisas irão bifurcar em algum ponto.

```
git checkout feature01
```

```
Switched to branch 'feature01'
```

```
Diretório existe.
Arquivo brasilCopa2014.txt já existe.
'brasilCopa2014.txt' -> '../meulrepo/brasilCopa2014.txt'
```

```
wget 'http://www.leg.ufpr.br/~walmes/cursoR/geneticaEsalq/brasilCopa2014.txt'
```

```
--2015-12-17 09:29:04-- http://www.leg.ufpr.br/~walmes/cursoR/geneticaEsalq/brasilCopa
Resolving www.leg.ufpr.br (www.leg.ufpr.br)... ????.???.????.??
Connecting to www.leg.ufpr.br (www.leg.ufpr.br)|????.???.????.??|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 1254 (1,2K) [text/plain]
Saving to: 'brasilCopa2014.txt'
```

```
0K .
```

```
100% 69,6M=0s
```

```
2015-12-17 09:29:04 (69,6 MB/s) - 'brasilCopa2014.txt' saved [1254/1254]
```

```
git add brasilCopa2014.txt
```

```
git commit -m "Arquivo sobre copa 2014 seleção brasileira."
```

```
[feature01 3044d9b] Arquivo sobre copa 2014 seleção brasileira.
1 file changed, 22 insertions(+)
create mode 100644 brasilCopa2014.txt
```

```
git log --graph --oneline --decorate --date=relative --all
```

```
* 3044d9b (HEAD -> feature01) Arquivo sobre copa 2014 seleção brasileira.
* 826f940 (master) Adiciona função R para VIF.
| * f26d9b4 (feature02) Adiciona aquivo de dados de experimento com rações.
|/
* 095046c Novos argumentos.
* 1aa33c3 Adiciona frase do Linux Torvalds.
* 8acac99 Lista de inicial de o porquê usar o Linux.
* 612c338 Cria arquivo com título
```

Agora nos temos o *feature01* na frente do *master* e o *feature02* ao lado. O conteúdo dos dois ramos terá que ser incorporado ao *master* em algum momento porque é assim que funciona. Mas não há razões para preocupação pois o propósito do Git é justamente facilitar esse processo. Nesse caso, por exemplo, como as diferenças nos ramos consiste na adição de arquivos diferentes, incorporar as modificações no *master* será uma tarefa simples para o Git. O agravante é quando em dois ramos (ou duas pessoas) o mesmo arquivo é modificado no mesmo intervalo de linhas. Nessa situação o *merge* nos arquivos deve ser supervisionado e não automático. Vamos incorporar as modificações dos ramos ao *master* então.

```
## Volta para o master.
```

```
git checkout master
```

```
Switched to branch 'master'
```

```
## Mescla o feature01.
```

```
git merge feature01 master
```

```
Updating 826f940..3044d9b
Fast-forward
 brasilCopa2014.txt | 22 +++++
1 file changed, 22 insertions(+)
create mode 100644 brasilCopa2014.txt
```

```
## Mescla o feature02.
```

```
git merge feature02 master
```

```
Merge made by the 'recursive' strategy.
 pimentel_racoes.txt | 24 +++++
1 file changed, 24 insertions(+)
create mode 100644 pimentel_racoes.txt
```

```
git log --graph --oneline --decorate --date=relative --all
```

```
* 039b7b6 (HEAD -> master) Merge branch 'feature02'
|\
| * f26d9b4 (feature02) Adiciona aquivo de dados de experimento com rações.
* | 3044d9b (feature01) Arquivo sobre copa 2014 celeção brasileira.
* | 826f940 Adiciona função R para VIF.
|/
* 095046c Novos argumentos.
* 1aa33c3 Adiciona frase do Linux Torvalds.
* 8acac99 Lista de inicial de o porquê usar o Linux.
* 612c338 Cria arquivo com título
```

```
tree --charset=ascii
```

```
.
|-- brasilCopa2014.txt
|-- pimentel_racoes.txt
|-- porqueLinux.txt
|-- README.txt
`-- vif.R
```

```
0 directories, 5 files
```

3.6 Resolvendo conflitos

Agora vamos de propósito mostrar uma situação em que não é possível fazer o merge automaticamente. Vamos criar um conflito. Para isso vou criar um ramo novo, modificar o arquivo na última linha e commitar. Vou voltar par ao *master* e fazer o mesmo mas vou usar um texto diferente para incluir no arquivo. Já que os ramos *feature01* e *feature02* não são mais necessários, podemos removê-los. No entanto, eles permanecem na história do projeto e poder ressurgir se você voltar no tempo.

```
## Remove ramos.
git branch -d feature01
git branch -d feature02
git branch
```

```
Deleted branch feature01 (was 3044d9b).
Deleted branch feature02 (was f26d9b4).
* master
```

```
git log --graph --oneline --decorate --date=relative --all
```



```
* 039b7b6 (HEAD -> master) Merge branch 'feature02'
|\
| * f26d9b4 Adiciona arquivo de dados de experimento com rações.
* | 3044d9b Arquivo sobre copa 2014 seleção brasileira.
* | 826f940 Adiciona função R para VIF.
|/
* 095046c Novos argumentos.
* 1aa33c3 Adiciona frase do Linux Torvalds.
* 8acac99 Lista de inicial de o porquê usar o Linux.
* 612c338 Cria arquivo com título
```

Agora vou criar um novo ramo, adicionar um arquivo e encurtar o nome das colunas no cabeçalho.

```
## Muda para um ramo criado na hora.
git checkout -b feature03
```

Switched to a new branch 'feature03'

```
## Baixa o arquivo.
wget 'http://www.leg.ufpr.br/~walmes/data/bib1.txt'
```

```
--2015-12-17 09:29:04-- http://www.leg.ufpr.br/~walmes/data/bib1.txt
Resolving www.leg.ufpr.br (www.leg.ufpr.br)... ????.???.????.??
Connecting to www.leg.ufpr.br (www.leg.ufpr.br)|????.???.????.??|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 535 [text/plain]
Saving to: 'bib1.txt'
```

0K

100% 35,0M=0s

2015-12-17 09:29:04 (35,0 MB/s) - 'bib1.txt' saved [535/535]

```
## Mostra as 4 primeiras linhas.
head -4 bib1.txt
```

REPT	VARI	BLOC	Y
1	1	1	20
1	2	1	18
1	3	2	15

Ao encurtar o nome para quatro dígitos, fica assim.

```
## Substitui o conteúdo da primeira linha pelos nomes truncados em 4
## dígitos e separados por tabulação. Etapa que você pode fazer no seu
## editor de texto.
sed -i "ls/./rept\tvari\tbloc\ty/" bib1.txt
head -4 bib1.txt
```

rept	vari	bloc	y
1	1	1	20
1	2	1	18
1	3	2	15

```
git add bib1.txt
git commit -m "Arquivo de experimento em BIB. Trunca cabeçalho 4 dígitos."
```

```
[feature03 d8bab6c] Arquivo de experimento em BIB. Trunca cabeçalho 4 dígitos.
1 file changed, 58 insertions(+)
create mode 100644 bib1.txt
```

Baixamos e modificamos o arquivo. Adicionamos e fizemos o registro das modificações. Agora vamos voltar ao *master* e baixar o arquivo também, fazendo de conta que é outra pessoa trabalhando no mesmo projeto, mas essa pessoa vai passar a cabeçalho para caixa alta.

```
git checkout master
```

Switched to branch 'master'

```
## Baixa o arquivo.
wget 'http://www.leg.ufpr.br/~walmes/data/bib1.txt'
```

Ao encurtar o nome para quatro dígitos, fica assim.

```
## Substitui o conteúdo da primeira linha pelo mesmo em caixa alta. Faça
## isso no seu editor de texto de preferido.
sed -i '1s/.*/\U&/' bib1.txt
head -4 bib1.txt
```

REPT	VARI	BLOC	Y
1	1	1	20
1	2	1	18
1	3	2	15

```
git add bib1.txt
git commit -m "Arquivo de experimento em BIB. Cabeçalho em caixa alta."
```

```
[master 50dc261] Arquivo de experimento em BIB. Cabeçalho em caixa alta.
1 file changed, 58 insertions(+)
create mode 100644 bib1.txt
```

```
git diff master feature03
```

```
diff --git a/bib1.txt b/bib1.txt
index 1d05031..b5f5963 100644
--- a/bib1.txt
+++ b/bib1.txt
@@ -1,4 +1,4 @@
-REPT    VARI    BLOC    Y
+rept    vari    bloc    y
 1 1    1    20
 1 2    1    18
 1 3    2    15
```

```
git log --graph --oneline --decorate --date=relative --all
```

```
* d8bab6c (feature03) Arquivo de experimento em BIB. Trunca cabeçalho 4 dígitos.
| * 50dc261 (HEAD -> master) Arquivo de experimento em BIB. Cabeçalho em caixa alta.
|/
* 039b7b6 Merge branch 'feature02'
|\
| * f26d9b4 Adiciona arquivo de dados de experimento com rações.
* | 3044d9b Arquivo sobre copa 2014 seleção brasileira.
* | 826f940 Adiciona função R para VIF.
|/
* 095046c Novos argumentos.
* 1aa33c3 Adiciona frase do Linux Torvalds.
* 8acac99 Lista de inicial de o porquê usar o Linux.
* 612c338 Cria arquivo com título
```

```
## Dá mensagem de erro que informa o conflito.
```

```
git merge feature03 master
```

```
Auto-merging bib1.txt
CONFLICT (add/add): Merge conflict in bib1.txt
Automatic merge failed; fix conflicts and then commit the result.
```

```
git status
```

```
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")
```

```
Unmerged paths:
  (use "git add <file>..." to mark resolution)
```

```
    both added:      bib1.txt
```

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

```
## `less` printa o conteúdo do arquivo mas `head` limita para 10 linhas.
less bib1.txt | head -10
```

```
<<<<<<< HEAD
REPT    VARI    BLOC    Y
=====
rept    vari    bloc    y
>>>>>>> feature03
1  1    1    20
1  2    1    18
1  3    2    15
1  4    2    16
1  5    3    14
```

Então deu conflito e o Git informa que ele deve ser resolvido. Resolver o conflito aqui significa abrir os arquivos com problema listados no Git status e editar de tal forma a “desconflitar”. Nas regiões de conflito o Git sinaliza de forma especial, indicando por divisórias (<<<<<<<, ===== e >>>>>>>) as versões no HEAD (ramo *master*) e no ramos a ser incorporado (*feature03*). Então vamos resolver o conflito sem favorecer ninguém, ou seja, vamos encurtar para 4 dígitos e manter caixa alta. Dessa forma o arquivo fica assim.

```
## Arquivo depois da edição que resolve o conflito.
head -6 bib1.txt
```

```
REPT    VARI    BLOC    Y
1  1    1    20
1  2    1    18
1  3    2    15
1  4    2    16
1  5    3    14
```

```
git status
```

```
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")
```

```
Unmerged paths:
  (use "git add <file>..." to mark resolution)
```

```
    both added:      bib1.txt
```

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

```
git add bib1.txt
git commit -m "Resolve conflito, trunca com caixa alta."
```

```
[master 9536074] Resolve conflito, trunca com caixa alta.
```

```
git status
```

```
On branch master
nothing to commit, working directory clean
```

```
git log --graph --oneline --decorate --date=relative --all
```

```
* 9536074 (HEAD -> master) Resolve conflito, trunca com caixa alta.
|\
| * d8bab6c (feature03) Arquivo de experimento em BIB. Trunca cabeçalho 4 dígitos.
* | 50dc261 Arquivo de experimento em BIB. Cabeçalho em caixa alta.
|/
* 039b7b6 Merge branch 'feature02'
|\
| * f26d9b4 Adiciona arquivo de dados de experimento com razões.
* | 3044d9b Arquivo sobre copa 2014 seleção brasileira.
* | 826f940 Adiciona função R para VIF.
|/
* 095046c Novos argumentos.
* 1aa33c3 Adiciona frase do Linux Torvalds.
* 8acac99 Lista de inicial de o porquê usar o Linux.
* 612c338 Cria arquivo com título
```

```
git reflog
```

```
9536074 HEAD@{0}: commit (merge): Resolve conflito, trunca com caixa alta.
50dc261 HEAD@{1}: commit: Arquivo de experimento em BIB. Cabeçalho em caixa alta.
039b7b6 HEAD@{2}: checkout: moving from feature03 to master
d8bab6c HEAD@{3}: commit: Arquivo de experimento em BIB. Trunca cabeçalho 4 dígitos.
039b7b6 HEAD@{4}: checkout: moving from master to feature03
039b7b6 HEAD@{5}: merge feature02: Merge made by the 'recursive' strategy.
3044d9b HEAD@{6}: merge feature01: Fast-forward
826f940 HEAD@{7}: checkout: moving from feature01 to master
3044d9b HEAD@{8}: commit: Arquivo sobre copa 2014 seleção brasileira.
826f940 HEAD@{9}: checkout: moving from feature02 to feature01
f26d9b4 HEAD@{10}: checkout: moving from f26d9b4eaf6e2050d137009ab67310f8ecb4a5b1 to feature02
f26d9b4 HEAD@{11}: commit: Adiciona arquivo de dados de experimento com razões.
095046c HEAD@{12}: checkout: moving from master to HEAD@{6}
826f940 HEAD@{13}: checkout: moving from 095046c3927fe5fa9932d9d3d858e3128126b4fc to master
095046c HEAD@{14}: checkout: moving from master to HEAD@{4}
826f940 HEAD@{15}: merge feature01: Fast-forward
```

```
095046c HEAD@{16}: checkout: moving from feature01 to master
826f940 HEAD@{17}: commit: Adiciona função R para VIF.
095046c HEAD@{18}: checkout: moving from master to feature01
095046c HEAD@{19}: commit: Novos argumentos.
1aa33c3 HEAD@{20}: commit: Adiciona frase do Linux Torvalds.
8acac99 HEAD@{21}: commit: Lista de inicial de o porquê usar o Linux.
612c338 HEAD@{22}: commit (initial): Cria arquivo com título
```

Capítulo 4

Projetos Remotos

Nos capítulos anteriores descrevemos como instalar o Git e ter projetos versionados. No entanto, o uso do Git até então foi apenas local. Os arquivos eram mantidos na sua máquina de trabalho e disponíveis só para você.

Os recursos do Git, como o desenvolvimento em *branches*, permite que vários segmentos sejam conduzidos de forma independente e no futuro, quando apropriado, reunidos em um único *branch*. Isso é exatamente o que precisamos para trabalhar em equipe, certo? Se cada colaborador pudesse ter um ramo separado do projeto para trabalhar, todos poderiam trabalhar simultaneamente. Quando oportuno, bastaria fazer merges para reunir o trabalho. A questão é: como deixar o projeto disponível para os colaboradores?

A resposta é simples: mantenha o projeto em um servidor onde os colaboradores tenham acesso. Isso inclusive vai permitir que você acesse o projeto de várias outras máquinas, como o *notebook* de casa e o desktop do *escritório*.

4.1 Repositório remoto pessoal

O repositório remoto serve de centro do seu repositório Git. Como ele está em um servidor que você tem acesso, você pode compartilhar o repositório com outras máquinas, clonando de lá. Ele serve como *backup* do repositório.

Aqui não se trabalha em colaboração mas o processo permite acessar o repositório, transferir arquivos de várias máquinas suas.

```
## Autenticar no servidor (logar).  
ssh eu@servidor  
  
## Verificar se a máquina tem o Git, se não instalar.  
git --version  
  
## Criar um diretório para conter o projeto.  
mkdir -p ~/meusProjetos/meulrepo
```

```
cd ~/meusProjetos/meulrepo

## Começar um projeto Git remoto. Note a opção --bare.
git --bare init
```

Apenas isso precisa ser feito no servidor. Você não cria nenhum arquivo pelo servidor. Inclusive, muitos dos comandos Git, como `git status` não funcionam para repositório iniciados com a opção `git --bare init`.

Caso queira, você também pode usar `git init`. A diferença entre eles é só onde ficam os arquivos do versionamento. Com `git init`, um diretório oculto `.git/` é o repositório Git e os arquivos de trabalho, como o `README.md`, ficam ao lado dele na estrutura de diretório. Com `git --bare init` o conteúdo do repositório Git fica na raiz. Essa última opção é justamente para criar repositórios remotos que vão justamente manter a parte repositório e não os arquivos.

<code>git init</code>	<code>git --bare init</code>
<code>.</code>	<code>.</code>
<code> -- .git</code>	<code> -- branches</code>
<code> -- branches</code>	<code> -- config</code>
<code> -- config</code>	<code> -- description</code>
<code> -- description</code>	<code> -- HEAD</code>
<code> -- HEAD</code>	<code> -- hooks</code>
<code> -- hooks</code>	<code> -- info</code>
<code> -- info</code>	<code> -- objects</code>
<code> -- objects</code>	<code>+-- refs</code>
<code> +-- refs</code>	
<code>+-- README.md</code>	

Uma vez iniciado o repositório no servidor, todo trabalho passa ser local. É a vez de adicionar o endereço do diretório no servidor e transferir arquivos.

```
## Agora na sua máquina local, adicione o endereço do remoto.
git remote add eu@server:~/meusProjetos/meulrepo

## Exibir os endereços remotos.
git remote -v
```

Esse endereço pode ter IP, porque na realidade, todo servidor tem um IP. Por exemplo, o servidor do `github.com` tem o IP `192.30.252.129`. Para saber o IP é só dar um `ping` no endereço.

```
ping github.com
ping gitlab.com
ping google.com
ping cran.r-project.org
```


Normalmente, servidores de escritório não tem um endereço nominal, apenas o IP (numérico). É necessário registrar domínio para ter nominal.

```
## Agora na sua máquina local, adicione o endereço do remoto.  
git remote add eu@111.22.333.44:~/meusProjetos/meu1repo
```

Nesse processo, toda transferência de arquivos vai pedir senha do seu usuário no servidor. Para facilitar, pode-se trabalhar com chaves públicas.

O arquivo `id_rsa.pub` é a sua chave pública. O `id_rsa` é o seu par. RSA é o tipo de encriptação. O nome e caminho do arquivo e a encriptação podem ser outros, depende do que você escolher ao gerar o par de chaves. Normalmente usa-se RSA e as chaves são criadas no diretório `~/.ssh`.

```
## Exibir chaves públicas.  
cat ~/.ssh/id_rsa.pub  
  
## Caso não tenha, gerar.  
ssh-keygen -t rsa -C "eu@dominio.com"
```

No servidor, o arquivo `authorized_keys2` contém as chaves públicas das máquinas com acesso permitido sem senha. Esse arquivo nada mais é que uma coleção de chaves. O conteúdo dele são as chaves das suas máquinas, ou das máquinas de outros usuários, conteúdo do `id_rsa.pub`, uma embaixo da outra, conforme o exemplo abaixo¹.

```
## `authorized_keys` do servidor da Liga da Justiça.  
ssh-rsa IyBUrjvUdSMY... flash@justiceleague.org  
ssh-rsa AAAAB3NzaCly... batman@justiceleague.org  
ssh-rsa Mdju17IdXhSd... superman@justiceleague.org
```

```
## Logar no servidor  
ssh eu@111.22.333.44  
  
## Criar o diretório/arquivo para as chaves públicas.  
mkdir ~/.ssh  
  > authorized_keys2
```

Agora é preciso transferir o conteúdo do seu arquivo local `id_rsa.pub` para o `authorized_keys2` que fica no servidor. O jeito mais simples de fazer isso é com transferência `scp` mas a instrução `ssh` abaixo também resolve.

```
## Transfere arquivo para diretório temporário.  
scp ~/.ssh/id_rsa.pub eu@111.22.333.44:/tmp  
  
## Cola o conteúdo do *.pub no final do authorized_keys.  
ssh eu@111.22.333.44\
```

¹O Flash foi o primeiro a transferir as chaves para o servidor porque ele é mais rápido

```
"cat /tmp/id_rsa.pub ~/.ssh/authorized_keys"

## Faz os dois passos anteriores de uma vez só.
ssh eu@111.22.333.44\
"cat >> ~/.ssh/authorized_keys2" < ~/.ssh/id_rsa.pub
```

4.2 Repositório remoto coletivo

A única diferença é recomendamos a você criar um novo usuário e adicionar as chaves públicas de todos os membros. Evite adicionar chaves públicas para usuários na sua conta porque isso expõe seus documentos, alguma operação desastrosa por parte de alguém pode comprometê-los. Por isso, crie um usuário, por exemplo gitusers, para nesta conta manter o repositório remoto.

Solicite que os colaboradores gerem as chaves públicas e te enviem o arquivo id_rsa.pub. Depois você adiciona cada chave ao authorized_keys de conta gitusers. Com chaves autorizadas, os colaboradores podem transferir arquivos, podem logar no servidor mas não podem instalar nada, a menos que você passe a senha do usuário gitusers. Para criar usuários no servidor, você precisa de privilégios de *admin*.

```
## Logar na servidora.
ssh eu@servidor

## No servidor, criar uma conta para o projeto.
sudo adduser gitusers
```

Vamos assumir que você tem os arquivos *.pub dos colaboradores no diretório /chaves devidamente identificados pelo nome deles. O comando abaixo acrescenta as chaves deles uma embaixo da outra no authorized_keys.

```
## Entra no diretório com os arquivos *.pub.
## Existem várias: angela.pub, jhenifer.pub, ...
cd chaves

## Juntar as chaves em um único arquivo.
cat *.pub > todos.pub

## Copiar o conteúdo do arquivo pro authorized_keys2.
ssh gitusers@111.22.333.44\
"cat >> ~/.ssh/authorized_keys2" < todos.pub
```

4.3 Fluxo de trabalho com repositório remoto, do clone ao push

4.3.1 Git clone

Este comando é usado para clonar um repositório do servidor remoto para um servidor local, caso você queira copiar um repositório que já existe para realizar colaborações em um projeto que queira participar. Você terá acesso a todos os arquivos e poderá verificar as diferentes versões destes. Supondo que sua equipe de trabalho possui uma biblioteca Git **Teste Clone**, onde são armazenados todos os arquivos. Você pode clonar estes arquivos para o seu diretório de trabalho e assim modificá-los conforme deseja.

Exemplo:

```
git clone git@gitlab.c3sl.ufpr.br:pet-estatistica/TesteClone.git
```

Desta forma você terá um diretório TesteClone em seu computador, onde estarão todos os arquivos do projeto nele.

Você também terá a opção de clonar o repositório TesteClone em um diretório diferente do padrão Git, que no próximo exemplo denominaremos de DirTeste:

Exemplo:

```
git clone git@gitlab.c3sl.ufpr.br:pet-estatistica/TesteClone.git DirTeste
```

4.3.2 Git Push

Após clonar e realizar contribuições ao projeto, você pode enviá-los para o repositório remoto. Estes arquivos, após o Git push, estarão prontos para serem integrados ao projeto com o merge. Usado para transferência de arquivos entre repositório local e o servidor remoto. Como o nome já diz, o comando empurra os arquivos para o servidor remoto. No exemplo abaixo enviaremos a ramificação Branch Master para o servidor chamado origin:

Exemplo:

```
git push origin master
```

É importante ressaltar que se dois usuários clonarem ao mesmo tempo, realizarem modificações e enviarem os arquivos atualizados ao repositório utilizando o Git push, as modificações do usuário que realizou o push por último serão desconsideradas.

4.3.3 Git Pull

Para obter todos os arquivos presentes no projeto, você pode importar os arquivos do branch master. Toda vez que você utilizar o `Git pull` a última versão de todos os arquivos estarão no seu diretório. Também usado para transferência de arquivos, o comando puxa os arquivos do servidor remoto para o repositório local e faz o merge do mesmo, fundindo a última versão com a versão atualizada.

Exemplo:

```
git pull origin master
```

4.3.4 Git fetch

Assim como o comando `Git pull`, o `Git fetch` transfere arquivos do repositório remoto para o local, porém ele não realiza automaticamente o merge dos arquivos transferidos, o usuário deve fazer o merge manualmente.

Exemplo:

```
git fetch origin master
```

Para verificar as modificações realizadas entre versões de um arquivo basta utilizar o comando `git diff`:

Exemplo:

```
git diff master origin/master
```

4.4 Listar branches locais/remotos

O comando `git remote` é usado para verificar quais repositórios estão configurados.

Exemplo: para retornar a lista de repositórios:

```
git remote
```

No comando acima é possível visualizar o remoto padrão **origin** (URL SSH para onde será possível enviar os seus arquivos).

Exemplo: para retornar o nome dos repositórios com a URL onde foram armazenados:

```
git remote -v
```

4.5 Adicionar, renomear, deletar remote

4.5.1 Adicionando repositórios remotos

O comando `git remote add` adiciona um repositório remoto. No exemplo a seguir será adicionado um repositório chamado **MeuRepo** ao qual será vinculado a URL `git@gitlab.c3sl.ufpr.br:pet-estatistica/apostila-git.git`. Usaremos como exemplo o projeto **Apostila-git**.

Exemplo:

```
git remote add MeuRepo \
  git@gitlab.c3sl.ufpr.br:pet-estatistica/apostila-git.git

## A lista de repositórios agora é:
git remote -v
```

Para acessar localmente o branch master do projeto **Apostila-git** será usado `MeuRepo/master`.

4.5.2 Obtendo informações de um Remoto

Você pode acessar as informações de qualquer repositório remoto com o comando `git remote show`, que retornará a URL e os branches.

Exemplo:

```
git remote show origin
```

4.5.3 Renomeado Remotos

O comando `git remote rename` pode modificar o nome de um repositório remoto. A seguir o repositório `MeuRepo` será renomeado para `RenameRepo`.

Exemplo:

```
git remote rename MeuRepo RenameRepo
```

4.5.4 Removendo Remotos

Para remover remotos é utilizado o comando `git remote rm`, agora será removido o repositório renomeado anteriormente `RenameRepo`.

Exemplo:

```
git remote rm RenameRepo
```

4.5.5 Deletar ramos no servidor

Quando houver um branch que não está sendo utilizado ou está concluído, há a opção de excluí-lo do servidor. Se for necessário apagar branches remotos, podemos utilizar o comando a seguir:

Exemplo:

```
git push origin --delete <branch>
```

4.5.6 Clonar apenas um *branch*, *commit* ou *tag*.

É possível clonar apenas um branch e não o repositório Git completo. Supondo que no repositório há um branch chamado MeuBranch dentro do repositório MeuRepo, clonaremos o branch.

Exemplo:

```
git clone -b MeuBranch --single-branch git://sub.domain.com/MeuRepo.git
```

O Git ainda permite clonar um commit ou tag.

Exemplo:

```
## Para uma tag:
```

```
git -e: //git.myproject.org/MyProject.git@v1.0#egg=MyProject
```

```
## Para um commit:
```

```
git -e: //git.myproject.org/MyProject.git@da39a3ee5e6b4b0d3255bfef95601890afd80709#egg=
```

4.6 Criando um Repositório Git

Primeiramente é necessário ter acesso a um servidor Linux com chave SSH, no qual você poderá ter seus repositórios. É definido um diretório no qual será armazenado o repositório remoto. No próximo exemplo é preciso criar um repositório remoto chamado MeuRepo e o armazenar em um diretório ~/git:

Exemplo:

```
# Para criar um diretório git na sua home:
```

```
mkdir ~/git
```

```
# Para criar um repositório git:
```

```
mkdir MeuRepo.git
```

```
# Para definir MeuRepo como um repositório remoto:
```

```
git --bare init
```

As configurações do servidor estão completas. A partir de agora você pode realizar os primeiros comandos para iniciar o repositório criado.

4.7 Git no servidor

Primeiramente, para configurar o Git no Servidor e configurar os protocolos, clonaremos o repositório existente em um repositório limpo. **Observação:** você poderá colocar um repositório no Servidor se este não contém um diretório de trabalho.

Exemplo:

```
git clone --bare MeuRepo MeuRepo.git
```

Acima foi criado um repositório limpo `MeuRepo.git`, no qual está armazenada a cópia de todos os arquivos do diretório Git.

Após este primeiro passo o repositório limpo será colocado no Servidor e configurado os protocolos. No exemplo abaixo, supondo que você tem configurado um servidor `git.servidor.com`, e um diretório `/dir/git` no qual você quer armazenar seus repositórios. Ao copiar o seu repositório limpo, você pode configurar seu novo repositório.

Exemplo:

```
scp -r MeuRepo.git usuario@git.example.com:/dir/git
```

Agora o repositório pode ser clonado por outros usuários, que podem ter acesso de escrita e de envio de arquivos push no diretório.

```
git clone usuario@git.example.com:/dir/git/MeuRepo.git
```

4.8 Configuração de Conexão SSH com Servidor

O Git possibilita ao usuário realizar uma chave SSH que fará uma conexão segura da sua máquina com o servidor. Para isso começamos com o seguinte comando no terminal:

Exemplo:

```
## Gerando uma chave ssh  
ssh-keygen -t rsa -C "usuario@email.com"
```

A partir deste comando, será possível alterar o diretório onde será salva a chave SSH. O usuário tem a opção de permanecer com o diretório padrão, para isso basta apertar Enter. A partir disso, são criados dois arquivos no diretório, o `id_rsa` e o `id_rsa.pub`. Após escolher o diretório onde serão

salvos os arquivos, você terá a opção de digitar uma senha ou deixar o espaço em branco.

Para visualizar a chave basta digitar o seguinte comando:

Exemplo:

```
cat ~/.ssh/id_rsa.pub
```

A chave está no arquivo `id_rsa.pub`. O usuário deve copiar o texto deste arquivo na íntegra. Para gerar a conexão ssh com o servidor, deve visitar o site <https://gitlab.c3sl.ufpr.br/profile/keys> e clicar em **Add SSH Key**. É necessário escrever um título para a sua nova chave, no campo key colar o texto copiado do arquivo `id_rsa.pub` e adicionar sua nova chave.

Para checar a configuração da sua máquina com o servidor basta realizar o seguinte comando:

Exemplo:

```
ssh -T git@gitlab.c3sl.ufpr.br
```

Configurando o servidor

Agora será abordado como configurar o acesso SSH do ponto de vista do servidor. Você precisa criar um usuário Git e um diretório `.ssh` para este usuário.

Exemplo: criar usuário e diretório.

```
sudo adduser git
su git
cd
mkdir .ssh
```

Agora, você terá um arquivo chamado `authorized_keys` onde será adicionado uma chave pública de algum desenvolvedor. Após obter chaves de alguns usuários, você pode salvá-las no arquivo `authorized_keys`, como no exemplo a seguir.

Exemplo:

```
## Chave do primeiro usuário.
cat /tmp/id_rsa1.pub >> ~/.ssh/authorized_keys

## Chave do segundo usuário.
cat /tmp/id_rsa2.pub >> ~/.ssh/authorized_keys
...
```

Depois de armazenar as chaves dos usuários, basta criar um repositório limpo (sem um diretório de trabalho) para eles. Como visto anteriormente:

Exemplo:


```
cd/dir/git  
mkdir NovoProjeto.git  
cd NovoProjeto.git  
git -bare init
```

Agora os usuários, cujas chaves foram salvas no arquivo `authorized_keys` podem compartilhar arquivos no repositório com os comando `git init`, `git add`, `git commit`, `git remote add` e `git push origin master`.

Capítulo 5

Serviços Web para Projetos Git

5.1 Serviços Web para Git

No capítulo anterior vimos como configurar um repositório remoto em um servidor. Esse procedimento possibilita trabalho em equipe pois centraliza o repositório remoto em um servidor que todos têm acesso. Assim, todos podem clonar, criar branches, subir as contribuições, etc. Apesar do servidor centralizar o trabalho de todos os usuários, estes terão que se comunicar e fazer a gestão de tarefas sobre o projeto de outra forma, como por e-mail direto, lista de emails/discussão ou chats coletivos. Para que um desenvolvedor veja o que os outros fizeram, ele terá que periodicamente dar `fetch`, ver os branches do repositório, navegar no histórico de commits, ver *diffs* entre arquivos. Se por um lado existe recurso para um fluxo de desenvolvimento orientado dessa forma, também existem recursos para tornar o trabalho coletivo mais simples e centralizado. Eles são os serviços web para projetos Git.

O Git tem vários serviços web voltados para ter um local que centralize o projeto bem como ofereça recursos administrativos e colaborativos. Esses serviços possuem contas *free*, alguns planos pagos e diversos recursos para todo tipo de projeto e equipe.

Os objetivos desse capítulo são: apresentar os serviços web para repositórios Git, descrever suas principais características, descrever como criar e configurar uma conta e conectar-se a um repositório local. Além disso, o *workflow* básico que considera serviços web será discutido, enfatizando os principais recursos desses serviços voltados à colaboração.

5.1.1 GitHub

O GitHub¹ é um serviço web para hospedagem, gestão e compartilhamento de repositórios Git que oferece recursos para desenvolvimento e colaboração. “*Build software better, together.*” é o principal slogan do GitHub, justamente para enfatizar o seu principal compromisso: dar suporte ao desenvolvimento colaborativo.



Figura 5.1: Octocat é o mascote do GitHub.

O GitHub foi fundado em 8 de Fevereiro de 2008, em São Francisco, por quatro pessoas: Tom Preston-Werner, Chris Wanstrath, PJ Hyett e Scott Chacon. Antes de terminar 2015, o GitHub já havia ultrapassado a marca de 10 milhões de usuários. De acordo com o <http://github.info/>, no quarto trimestre de 2014 haviam 22 milhões de repositórios. A linguagem JavaScript teve o maior número de repositórios ativos (>320 mil) e total de *pushes*, enquanto que a linguagem R teve o maior número de novas cópias por repositório (6.45).

Diferente da forma tradicional de usar o Git, por linha de comando, o GitHub é um serviço web com interface gráfica repleta de funções para o desenvolvimento e acompanhamento de um projeto Git. Tais recursos vão desde administrar tarefas até permitir a colaboração de outras pessoas, mesmo sendo desconhecidas. Dentre os principais recursos disponíveis, têm-se:

- **README:** é um arquivo que funciona como capa do seu repositório. O seu conteúdo é texto escrito em linguagem de marcação (Markdown, RST, Textile, Org, etc) renderizada para exibição pelo GitHub. Como capa, o conteúdo do README está na *home* do projeto e serve para informar o visitante dos objetivos do repositório, seus desenvolvedores, instruções de instalação/uso e formas de colaboração.
- **Wiki:** a Wiki de cada repositório é um conjunto de páginas que serve para divulgação e documentação do repositório. Estas também são escritas em linguagem de marcação, tornando fácil e rápida a escrita pelo desenvolvedor e simples para o visitante ler e navegar. Como a Wiki é também um repositório Git, ela pode inclusive ser editada por meios dos recursos de edição do próprio GitHub, além de ser versionada. Com isso, não diferente do restante, a edição da Wiki também é colaborativa.

¹<https://github.com/>

- *Issues*: pelos *issues* são feitas as notificações de *bug* e gerenciamento de tarefas. Os usuários criam *issues* para notificar um *bug* encontrado de forma que ele possa ser rapidamente corrigido. Criar *issues* também serve como ferramenta de administração de tarefas nas quais eles descrevem algo a ser feito por alguém.
- *Milestones*: são pedras de milha, ou seja, marcam um ponto a ser alcançado. São usadas para descrever o que precisa ser desenvolvido para que ocorra uma mudança de versão e estabelecer um prazo para conclusão, por exemplo. As *milestones* agrupam *issues* que indicam as etapas a serem percorridas.
- *Pull request* ou *merge request* (MR): é uma requisição de fusão. Os membros da equipe fazem suas contribuições em ramos de desenvolvimento e ao concluir pedem um MR pela interface. O responsável por avaliar o MR pode ver os *diffs* nos arquivos e fazer o merge direto pela interface, de dentro do serviço sem precisar baixar (*fetch*) o ramo, aplicar o merge (*merge*) e subi-lo (*push*). Então os desenvolvedores não precisam interromper o trabalho local para fazer um merge já que ele pode ser feito no serviço sem modificações no *workspace*.
- *Fork*: é uma forma de se fazer uma cópia do projeto de alguém para livremente experimentar modificações sem afetar o projeto original. A cópia vem para a sua conta e funciona como qualquer outro repositório seu. A ideia do *fork* é dar liberdade de contribuição (não supervisionada) a qualquer pessoa interessada de modo que esta possa submeter as contribuições para a origem (por MR) ou até mesmo usar como ponto de partida para um novo projeto.

De acordo com Klint Finley², *fork* e MR são os recursos que tornam o GitHub tão poderoso. Quando o mantenedor recebe um MR ele pode ver o perfil do contribuidor onde estão listados todos os projetos no qual este contribuiu. Ao aceitar o MR, é acrescentado mais uma colaboração a reputação do colaborador. Esse mecanismo, então, beneficia as duas partes.

Além dessas características chave, o GitHub permite que você acompanhe (*watch*) e favorite (*star*) repositórios. Também dá para seguir pessoas e participar de organizações (grupo de usuários) que podem ter repositórios próprios, ideal para projetos em equipe.

O perfil de cada usuário registra suas atividades em todos os projetos e em cada um pode-se acompanhar as contribuições de cada colaborador. Nos repositórios pode-se navegar pelo histórico de *commits*, filtrá-los por colaborador, ver as modificações no código (*diffs*) comparando *commits* e *branches*.

O GitHub não hospeda apenas código fonte mas sim todo e qualquer arquivo que você tenha sob versionamento. É possível hospedar dados, por exemplo, em formato texto (csv, txt), e disponibilizá-los por meio da URL para download ou leitura direta. Para usuários de R, essa é uma característica que permite não apenas disponibilizar dados, mas também coleções de funções que podem ser carregadas com um `source()`.

²<http://techcrunch.com/2012/07/14/what-exactly-is-github-anyway/>

Com o plano *free* do GitHub, você pode ter inúmeros repositórios públicos, inúmeros colaboradores, ter o *fork* de quantos repositórios quiser e participar de quantas organizações precisar. Para ter repositórios privados, o plano mais básico custa U\$ 7 por mês e dá direito à 5 repositórios. Existem outros planos individuais, e também planos organizacionais, para todos os tamanhos de projeto e equipe. Além dessas opções, pode-se ter o GitHub em um servidor próprio, o GitHub Enterprise³, com recursos e vantagens além das já mencionadas

Para muitos programadores, o GitHub é uma fonte de conhecimento. Lá você encontra *scripts* de todas as linguagens de programação. Você pode livremente estudar o código dos repositórios, ver como o código evoluiu *commit* após *commit* e acompanhar como um *bug* foi resolvido.

Qualquer pessoa, mesmo sem perfil no GitHub, pode clonar um repositório público. O GitHub reconhece 382 linguagens que compreendem as de programação (293: C++, Python, R, JavaScript), as de *markup* (34: HTML, TeX, Markdown), as de dados (40: JSON, SQL, XML, csv) e aplica os realces de código (highlights) que facilitam a sua compreensão.

O GitHub é o serviço web para Git mais popular quanto ao número de projetos hospedados. No entanto, existem serviços com as mesmas funcionalidades e até com algumas que o GitHub não oferece no plano básico. O GitLab e o Bitbucket estão entre os 5 mais populares e permitem, por exemplo, ter alguns repositórios privados com a conta *free*.

Como hospedamos nossos projetos coletivos do PET-Estatística no GitLab do C3SL⁴, não podemos deixar de falar sobre ele.

5.1.2 GitLab

O GitLab⁵, assim como o GitHub, é um serviço web para repositórios Git. O GitLab é um projeto *open source* desenvolvido em Ruby que teve início em 2011 pelo ucraniano Dmitriy Zaporozhets. Em 2013, a Companhia GitLab tinha uma equipe de 11 funcionários e mais de 10 mil organizações usando o serviço.

O GitLab, além de ser um serviço gratuito (com planos extras), é também um programa que você pode instalar em servidor local para ter seus repositórios na intraweb. Esse é o caso do GitLab do C3SL e do GitLab do LEG⁶.

O GitLab oferece todos os recursos do GitHub⁷. No entanto, com uma conta gratuita no <http://gitlab.com>, você pode ter além de repositórios públicos e colaboradores, ilimitados repositórios privados sem pagar por nada. Isso faz do GitLab.com o lugar certo para pequenas equipes com pouco recurso ou que desenvolvem trabalhos sem fins lucrativos, como é o caso de colaboração em código para análise de dados para publicação científica. Atualmente existem mais de 69 mil projetos públicos e 5840 grupos abertos no GitLab.com.

³<https://enterprise.github.com>

⁴<https://gitlab.c3sl.ufpr.br/explore>

⁵<https://about.gitlab.com/>

⁶<http://git.leg.ufpr.br/explore>

⁷<http://technologyconversations.com/2015/10/16/github-vs-gitlabs-vs-bitbucket-server-formerly-stash>



Figura 5.2: O guaxinim (*raccoon* em inglês) é o mascote do GitLab.

A versão paga do GitLab, *Enterprise Edition* (EE), tem um preço menor que a equivalente do GitHub.

A versão gratuita do GitLab para servidores, a *Community Edition* (CE), pode ser instalada em servidores Linux, Windows, máquinas virtuais e servidores na nuvem, além de outras opções. Os endereços <https://gitlab.c3sl.ufpr.br/explore> e <http://git.leg.ufpr.br/explore> são o GitLab CE para servidores rodando no C3SL (Centro de Computação Científica e Software Livre) e LEG (Laboratório de Estatística e Geoinformação). Estes GitLabs dão suporte à colaboração em código dentro dos departamentos de Informática e Estatística para alunos e professores (C3SL) e para a equipe e colaboradores do LEG.

Em termos financeiros, custa menos um servidor na nuvem da Digital Ocean⁸ com o GitLab CE (U\$ 5/mês) do que ter uma conta para repositórios privados no GitHub (U\$ 7/mês) por 5 repositórios privados). No entanto, conforme já mencionamos, pequenas equipes podem ter repositórios privados na conta gratuita do GitLab.com.

Além das características essenciais e comuns aos demais serviços, como *issues*, *fork*, *merge requests*, *wiki* e *snippets*, o GitLab tem 1) 5 níveis de acesso aos repositórios (*owner*, *master*, *developer*, *report* e *guess*), 2) revisão comentada de código nos *diffs*, 3) importação repositórios de outros serviços, 4) adição de *web hooks* e 5) integração contínua nativa a partir da versão 8.0.

5.2 Criar um perfil

Criar uma conta no Github é tão simples como uma conta de e-mail ou numa rede social. Acesse o endereço <https://github.com/join> para preencher seus dados pessoais e escolher um plano. Nenhum dos planos tem limitação quanto ao número de repositórios ou colaboradores. O que muda é a quantidade de repositórios privados. No plano *free*, só são criados repositórios públicos enquanto que nos planos pagos, o menor deles permite 5 repositórios privados

⁸<https://www.digitalocean.com/community/tutorials/how-to-use-the-gitlab-one-click-install-image-to-manage-git-repositories>

por um custo de US\$ 7 por mês. Acesse <https://github.com/pricing> para mais detalhes.

Ao preencher o formulário de criação de conta, você receberá um e-mail com uma url de ativação. Se optar por planos pagos, terá que informar o número do cartão de crédito para que seja feito o pagamento mensalmente.

Para criar uma conta gratuita no GitLab, acesse https://gitlab.com/users/sign_in. Os serviços GitLab que usamos são instalações em servidoras próprias (do C3SL e do LEG), no entanto, a interface do usuário é a mesma, com exceção de alguns privilégios administrativos.

5.2.1 Habilitar comunicação

Uma vez criada uma conta, é necessário habilitar a comunicação entre sua máquina e o (servidor do) GitHub. A comunicação é feita com protocolo SSH (*Secure Shell*), o qual já usamos no capítulo anterior para hospedar o repositório em um servidor remoto.

Para lembrar, a maioria dos servidores suporta a autenticação por SSH. Para que a conexão entre máquinas seja automática, ou seja, sem precisar fornecer usuário e senha a cada acesso/transferência, usamos o recurso de pares de chaves. Este serve para fazer a máquina remota (servidor) reconhecer a máquina local (sua máquina) via autenticação do par de chaves. É como se o servidor fosse um cadeado e a sua máquina local tem a chave que o abre. Uma vez que as chaves são pareadas e compatíveis, ocorre o acesso ou transferência de arquivos.

Para gerar as chaves públicas, você precisa executar:

```
## Gera chaves públicas.
ssh-keygen -t rsa -C "seu_email@seu.provedor"
```

```
## Batman gerando chaves públicas.
ssh-keygen -t rsa -C "batman@justiceleague.org"
```

Generating public/private rsa key pair.

Enter file in which to save the key (/home/batman/.ssh/id_rsa):

Enter passphrase (empty for no passphrase):

Enter same passphrase again:

Your identification has been saved in /home/batman/.ssh/id_rsa.

Your public key has been saved in /home/batman/.ssh/id_rsa.pub.

The key fingerprint is:

66:c1:0b:3a:94:25:83:00:81:9f:40:26:f7:aa:af:3a batman@justiceleague.org

The key's randomart image is:

```

+---[ RSA 2048]-----+
  |                     |
~MMMMMMMMMMMMMM      MMMMMMMMMMMMM~
 .MMMMMMMMM.   MMM   .MMMMMMMMMM.
  MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
```

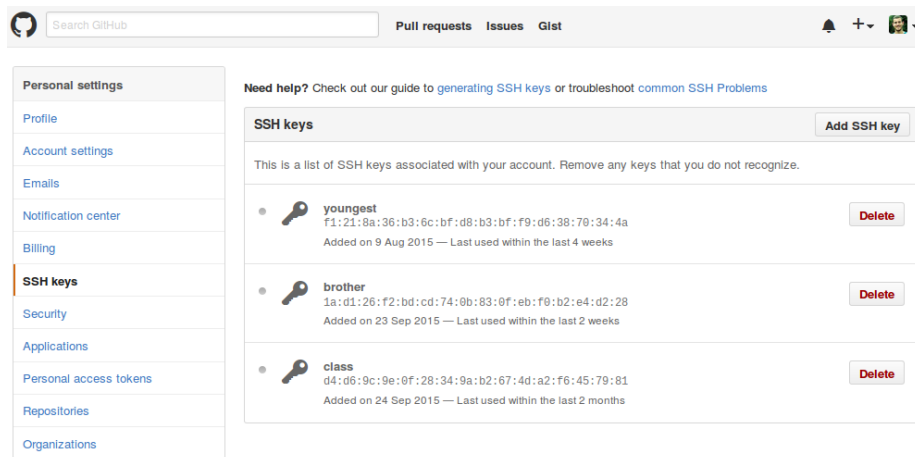



Figura 5.3: *Printscreen* da página de configurações pessoais do GitHub. No menu SSH Keys pode-se ver e adicionar chaves públicas.

```

MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
.MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM.
      .MMMMMMMMMM.
        .MMM.
          M
      |         |
      +-----+

```

O endereço padrão para os arquivos criados é o diretório `~/ .ssh/`. Os arquivos serão reescritos caso já existam arquivos de chaves públicas lá. Todo novo par de chaves é único. Então, se você reescreveu os arquivos anteriores, terá que atualizar as chaves públicas em todos os serviços web que fazem uso desse recurso e com todos os servidores com o qual você tem autenticação por chaves.

Acesse <https://github.com/settings/ssh> para então adicionar chaves públicas (Figura 5.3) ao seu perfil. Clique em `Add SSH key`, cole o conteúdo copiado do arquivo `*.pub` no campo `key`. No campo `Title` identifique a máquina correspondente àquela chave. Use, por exemplo, `laptop` ou `trabalho` para facilitar o reconhecimento. É comum trabalhar-se com mais de um máquina, como uma em casa e outra no trabalho.

Para testar a comunicação entre o GitHub e a sua máquina, execute:

```

## Testa comunicação. Retorna um "Welcome!" em caso positivo.
ssh -T git@github.com

## Se falhar, habilite o modo verbose para rastrear o erro.
ssh -vT git@github.com

```

No GitLab, o cadastro de chaves públicas é um processo semelhante. Uma

vez autenticado, acesse <https://gitlab.c3sl.ufpr.br/profile/keys> para adicionar chaves públicas. Para testar a comunicação entre o GitLab e a sua máquina, execute:


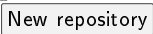
```
## Testa comunicação. Retorna um "Welcome!" em caso positivo.  
ssh -T git@gitlab.c3sl.ufpr.br  
  
## Se falhar, habilite o modo verbose para rastrear o erro.  
ssh -vT git@gitlab.c3sl.ufpr.br
```

Lembre-se de que o endereço `gitlab.c3sl.ufpr.br` corresponde ao serviço GitLab disponibilizado pelo C3SL. Caso você esteja fazendo a conta no GitLab.com, o endereço muda de acordo.

5.2.2 Gerenciar repositórios

GitHub

A comunicação com o GitHub acabou de ser estabelecida. Agora podemos criar repositórios e começar a mostrar nosso trabalho para o mundo e colaborar de forma eficiente.

No canto superior direito das páginas do GitHub existe um ícone  que permite criar um novo repositório ou uma nova organização. Clique em  ou acesse o endereço <https://github.com/new>. Na janela que abrir, dê um nome para o seu projeto e adicione uma breve descrição à ele (Figura 5.4). Na etapa seguinte, defina o nível de visibilidade: público ou privado. Lembre-se que os planos *free* só permitem repositórios públicos. Ao clicar em privado você passará pelos formulários para efetuar pagamento.

Para criar o projeto dentro de uma Organização, selecione a Organização na *drop list* que fica no campo *Owner*, a esquerda do campo para o nome do repositório.

Você pode inicializar o repositório com um arquivo `README.md`, o que é altamente recomendado. Como já mencionamos, esse arquivo é a capa do seu repositório e serve para documentar o seu objetivo, formas de colaboração, colaboradores, formas de instalação/uso.

Você pode editar o arquivo `README.md` (ou qualquer outro) no próprio GitHub. As modificações que fizer devem ser *commitadas* para serem salvas. O arquivo `README.md`, que é linguagem de marcação Markdown, é automaticamente renderizado pelo GitHub fazendo com que *urls* sejam clicáveis e códigos estejam em ambientes de fonte monoespaço, além de ter títulos com tamanho de fonte apropriado.


Depois de criar o repositório, você já pode cloná-lo para trabalhar localmente. O endereço do repositório é composto pelo seu nome de usuário (ou organização) e nome do repositório. O repositório bat-rangue da conta do batman pode ser clonado com:

Create a new repository

A repository contains all the files for your project, including the revision history.

Owner

Repository name

 walmes ▾


 /

ola-mundo ✓


Great repository names are short and memorable. Need inspiration? How about **crispy-octo-meow**.

Description (optional)

Meu primeiro projeto

☒  **Public**

Anyone can see this repository. You choose who can commit.

☐  **Private**

You choose who can see and commit to this repository.

☒ **Initialize this repository with a README**

This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: **None** ▾

 |

Add a license: **None** ▾ ⓘ

Create repository

Figura 5.4: Janela para a criação de um novo repositório no GitHub.

```
## Clone pelo protocolo ssh. Requer chaves públicas.  
git clone git@github.com:batman/bat-rangue.git
```

Pode-se clonar repositórios pelo protocolo http (*Hypertext Transfer Protocol*). Em geral, para clonar repositórios de outros usuários e fazer testes, usa-se http. Embora você possa usar http nos seus repositórios, prefira o SSH. Com http, o endereço passa a ser:

```
git clone https://github.com/batman/bat-rangue.git
```

Por padrão, ao clonar o repositório, um diretório de mesmo nome é criado com o projeto em seu interior. Em caso de preferir outro nome para esse diretório, por exemplo, bat-bumerangue, use:

```
git clone https://github.com/batman/bat-rangue.git \  
    bat-bumerangue
```

Existe outra situação que é quando você já tem repositório Git no qual já está trabalhando e quer tê-lo no GitHub. Nesse caso, você vai criar um novo repositório mas não irá cloná-lo, apenas copie a url que usaria para cloná-lo. No repositório local, adicione essa *url* como endereço do servidor remoto e faça um *push*. Vamos supor que o repositório seja um artigo científico de nome Artigo. Ao criar o repositório com esse nome no GitHub, o endereço fica `git@github.com:fulano/Artigo.git`. Então é só adicionar esse endereço como origin do projeto Git:

```
## Adiciona endereço de "origin" ao repositório.  
git remote add origin git@github.com:fulano/Artigo.git  
  
## Sobe o conteúdo do repositório.  
git push -u origin master
```

O seu projeto é configurado em `Settings`. Para renomear, deletar ou transferir o projeto para outro usuário ou organização, vá em `Options`. Em `Collaborators` você administra os colaboradores do projeto. Para gerenciar os ramos de trabalho, como proteger ou remover ramos, vá em `Branches`. O uso de serviços web é configurado no `Webhooks & services`. O `Deploy keys` permite adicionar chaves públicas.

Vamos considerar um repositório bem ativo para conhecermos um pouco mais dos recursos do GitHub. O repositório <https://github.com/yihui/knitr> (Figura 5.5) é o sítio de desenvolvimento do pacote `knitr` do R, o principal pacote para a geração de relatórios dinâmicos. Nesse repositório tem-se acesso aos fontes.

Na figura 5.5, logo abaixo do título, tem-se quatro quantificadores:

- *commits*: ao clicar neste item, tem-se o histórico de *commits* com autor, mensagem e SHA1. É possível comparar estados dos arquivos (*diff*) ao clicar no SHA1.

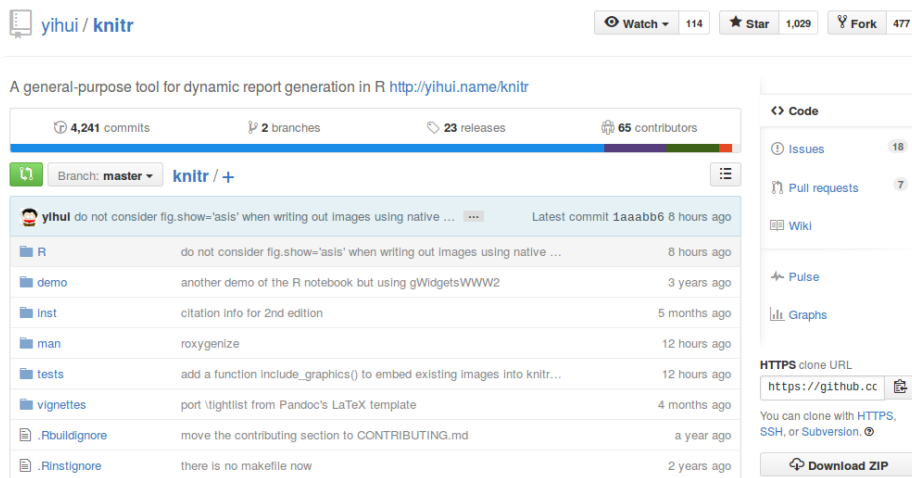


Figura 5.5: Home de um repositório no GitHub.

- *branches*: este lista os *branches* do projeto, permite comparar ramos.
- *releases*: documenta as modificações de uma versão para outra e lista os *commits* que tem *tags*. Essa é uma fonte importante para saber as maiores mudanças entre versões.
- *contributors*: dá um resumo das atividades dos colaboradores do projeto. Na aba *members* tem-se uma lista de todos os usuários que fizeram o *fork* do seu projeto. Falaremos de *fork* adiante.

No menu da direita existem links para acessos a mais informações:

- *Code*: estão visíveis os diretórios e arquivos do projeto. Pode-se navegar entre eles. Ao visualizar um arquivo, ele é exibido com realces de acordo com a linguagem para facilitar a compreensão. Ao abrir um arquivo, no topo aparecem botões de exibição/ação: *Raw* é para ver o arquivo cru, sem renderização e *highlights*. A *url* dessa exibição pode ser usada para ler/baixar arquivos de dados ou scripts direto da web. *Blame* mostra o arquivo com autor para cada porção do código. O *History* mostra o histórico de *commits*. Os dos ícones seguintes permitem editar o arquivo ou deletar.
- *Issues*: permite criação e acesso aos *issues* do projeto. Dentro dessa página tem-se acesso às *Milestones*, que são as coleções de *issues* por tema.
- *Pull requests*: é o termo que o GitHub usa para requisição de merge. Nesta página pode-se submeter uma requisição e navegar nas existentes.
- *Wiki*: na Wiki de um repositório normalmente é feita uma documentação mais detalhada do projeto. Páginas Wiki de um repositório também são repositórios Git, portanto, versionáveis.
- *Pulse*: dá um resumo sobre a quantidade de *issues* presentes, fechados e abertos bem como para as requisições de mescla. Mostra também um resumo da atividade recente do repositório.

- *Graphics*: exibe gráficos sobre a atividade no repositório, como frequência de *commits*, total e por autor, instantes do dia de maior colaboração, etc.

Existem ainda mais coisas sobre o GitHub que não podemos deixar de comentar: os recursos disponíveis para colaboração. Nas sessões à frente, trataremos dos recursos de *issue*, *fork* e requisições de merge. Antes, no entanto, vamos conhecer um pouco do GitLab, um serviço web para projetos Git que pode ser instalado no seu servidor.

O GitLab é o serviço que nós do PET usamos para colaboração em nossos projetos. Além disso, é o que os alunos usam para fazerem seus trabalhos e desenvolverem o TCC. O uso do Git como ferramenta de trabalho passou ser estimulado no segundo semestre de 2015. Além de reunir os alunos e professores numa plataforma de colaboração eficiente, o conhecimento de Git passa ser um diferencial no currículo.

GitLab

O GitLab concentra os acessos à outras páginas em um menu do lado esquerdo. Do lado direito pode haver informações extras. O botão para criar um novo repositório fica no canto superior direito. Ao lado deste tem botões para ir para página de configurações, sair, ajuda e explorar o GitLab.

No menu da direita tem-se acesso aos projetos do usuário ([Yout Projects](#)) e [Starred Projects](#)) e aos grupos do qual participa ([Groups](#)). As entradas [Milestones](#), [Issues](#) e [Merge requests](#) reúnem as informações sobre todos os projetos do qual o usuário participa.

Assim como acontece com outros serviços web, na página inicial do projeto é exibido o arquivo de capa, o README. Centralizado na página encontra-se o endereço para clonar o repositório por SSH ou HTTP(S) e as principais entradas estão no menu da direita:

- *Project*: é a página inicial;
- *Activity* e *Commits*: listam a atividade (predominantemente *commits*);
- *Files*: apresenta diretórios e arquivos, com opção de mudar o ramo;
- *Network*: o estado de desenvolvimento dos ramos do projeto com uma lista de todos os *commits*;
- *Graphs*: contém a atividade de cada membro do projeto;
- *Milestones* reúne as marcas de milhas do projeto com progresso de cada uma;
- *Issues*: pode-se gerenciar os *issues* dos projetos;
- *Merge requests*: permite criação e acompanhamento das requisições de mescla;
- *Members*: faz a gestão da equipe de trabalho como adição e redefinição dos níveis de acesso;
- *Labels* são usados para classificar os *issues* - é comum usar *bug*, *fix*, *review*.
- *Wiki*: vai para a página Wiki do repositório, onde é possível editá-la;
- *Settings*: são feitas as configurações gerais do projeto como definição de nome, descrição, nível de visibilidade e adição e *Web Hooks*. Nessa mesma página pode-se renomear, transferir ou remover o projeto.

Agora que conhecemos um pouco da interface dos serviços Git, podemos descrever o procedimento de trabalho considerando tais interfaces.

5.3 Fluxo de trabalho

O fluxo de trabalho de um projeto Git local usando um servidor remoto foram apresentados no capítulo anterior. O fluxo com um repositório Git mantido em um serviço, como o GitHub ou GitLab, não muda muito. A maior diferença não é sobre o uso de novas instruções Git mas sim a adoção de uma estratégia de trabalho (*workflow*) baseada nos recursos desses serviços, como as *milestones* e *issues*. Esse modelos de trabalho serão descritos em um capítulo à frente.

O fluxo de trabalho com repositórios remotos, em termos de comandos, acrescenta àqueles necessários para se comunicar com o repositório. Alguns desses comandos, senão todos, já foram apresentados no capítulo anterior a esse.

Ao considerar um serviço web, você pode começar um repositório novo de duas formas: localmente ou pelo serviço.

Pelo serviço, qualquer que seja, crie um novo repositório. GitHub e GitLab dão instruções resumidas de como proceder logo que você cria um novo repositório. Eles inclusive, incluem opções como criar um repositório com arquivo de README. Assim que criar, seu repositório terá um endereço (SSH e HTTP). Na sessão anterior, descremos o processo de criar e *clonar* o repositório e também de criar local e adicionar a *url* do repositório remoto (*origin*).

Localmente o repositório segue o ciclo normal de desenvolvimento que minimamente contém as instruções `git add` e `git commit`. Os fluxos de trabalho, em geral, preconizam o desenvolvimento a partir de *branches* de demanda cujo conteúdo será incorporado aos ramos permanentes (*master*, *devel*) quando forem concluídos. Abaixo ilustramos a sequência de criar um ramo, desenvolvê-lo e por fim subir o seu conteúdo para o repositório remoto, que nesse caso é mantido em um servidor web.

```
## Cria um ramo chamado issue#10.  
git branch issue#10  
  
## Muda do ramo atual para o recém criado.  
git checkout issue#10  
  
## Segue a rotina.  
git add <arquivos>  
git commit -m <mensagem>  
  
## Sempre que quiser, suba o trabalho.  
git push origin issue#10
```

Nessa rotina, consideramos *issue#10* um *issue* criado na interface web para atender uma demanda, como por exemplo, corrigir um *bug*. Da mesma forma que o ramo default do Git é o *master*, o repositório remoto é por default

chamado de origin, então subimos o conteúdo desse ramo para o servidor que mantém o repositório sob o serviço web.

Não há limites para o número de ramos. Ramos podem ser locais, remotos ou ambos. Quando você cria um ramo, como nos comandos acima, ele é um ramo local. Quando você sobre esse ramo, ele passa ter sua parte remota também. E quando você clona um repositório, você traz todos os ramos que ele possui, que são ramos remotos. Ao escolher um ramo desse para trabalhar, ele passa a ser também um ramo local. Segue abaixo, formas de listar os ramos do projeto.

```
## Lista os ramos locais.  
git branch -l  
  
## Lista os remotos.  
git branch -r  
  
## Lista todos (all).  
git branch -a
```

Você pode notar que ramos locais não tem prefixo e que ramos remotos tem o prefixo origin/. Você pode remover um ramo local mas manter sua parte remota e pode também excluir esse ramo por completo de todo o sistema, localmente e no servidor, conforme ilustram os códigos abaixo.

```
## Remove um ramo local (não mais útil).  
git branch -d issue#10  
  
## Remove sua cópia remota (não mais útil também).  
git branch -dr origin/issue#10  
  
## Remove um ramo remoto na origem (duas formas).  
git push origin --delete issue#10  
git push origin :issue#10
```

Até aqui nos referimos ao repositório remoto como origin que é o nome default. No entanto, um projeto Git pode ter mais de um repositório remoto. Considere o caso simples de um repositório para arquivos de uma disciplina. Esse repositório contém tanto lista de exercícios para os alunos como também as provas com as soluções. Para não entregar as provas de bandeja, o diretório de provas existe no ramo secret e é enviado somente para um repositório privado chamado provas. Já o conteúdo restante (lista de exercícios, notas de aula, scripts), disponibilizado para os alunos, fica no ramo master, mantido em um repositório aberto chamado de origin.

Dois repositórios remotos tem endereços distintos pois são projetos distintos no serviço web. Você pode ter a parte pública do projeto (origin) no GitHub e a parte privada, as provas (secret), no GitLab. Abaixo tem-se uma série de comandos para listar, renomear, remover e adicionar endereços para remotos.


```
## Lista os remotos.
git remote -v

## Renomeia para um nome melhor
git remote rename origin profs

## Remove.
git remote rm profs

## Adiciona um endereço de remoto (origin)
git remote add origin <url>
git remote add profs <url>

## Adiciona/remove URLs ao origin.
git remote set-url origin --add <url>
git remote set-url origin --delete <url>
```

Quando você trabalha em colaboração ou em mais de uma máquina, vai sempre precisar atualizar os repositórios com o conteúdo existente no remoto. Existem duas instruções Git para isso: `git fetch` e `git pull`.

As duas traduções mais frequentes do verbo *to fetch* são buscar e trazer. A documentação oficial do comando `git fetch`⁹ indica que esse comando serve para trazer objetos e referências dos repositórios remotos. Abaixo o comando padrão considerando um remoto de nome `origin` e algumas variações.

```
## Traz todos os ramos do remoto origin.
git fetch origin

## Traz só do ramo devel.
git fetch origin devel

## Traz de todos os remotos: origin, profs.
git fetch --all
```

O comando `fetch` traz os ramos e atualiza a parte remota, por exemplo, o `origin/devel`. O ramo local `devel` não tem seu conteúdo modificado após o `fetch`. Para transferir o conteúdo `origin/devel` para o `devel` tem-se que usar o `git merge`. No entanto, sempre que houver necessidade, antes do *merge* pode-se verificar as diferenças que existem entre local e remoto, principalmente quando esse remoto traz contribuições de algum colaborador. Os comandos abaixo exibem as diferenças entre os ramos e aplicam o `merge` entre eles.

```
## Muda para o ramo devel.
git checkout devel

## Mostra as diferenças entre devel local e remoto no terminal.
```

⁹<https://git-scm.com/docs/git-fetch>

```
git diff devel origin/devel

## Faz o merge do devel remoto no devel local.
git merge origin/devel
```

Em resumo, para passar o conteúdo de um ramo remoto para um local temos que fazer `git fetch` e em seguida `git merge`. O comando `git pull` é esses dois, em um. A documentação do `git pull`¹⁰ dá uma descrição detalhada do seu uso enquanto que os exemplos abaixo ilustram o uso mais simples possível.

```
## Traz e junta todos os ramos do remoto com os locais.
git pull origin

## Traz e junta só do ramo devel.
git pull origin devel
```

Por fim, para subir o trabalho feito em um ramo, usa-se a instrução `git push`. Enviar o seu trabalho com frequência é a melhor forma de ter *backups*, exibir e disponibilizar suas contribuições para os seus colaboradores. A documentação do `git push`¹¹ é rica em variações mas a maneira mais simples de usá-lo é para subir um ramo para o repositório remoto.

```
## Sobe o ramo issue#10 para o repositório remoto origin.
git push origin issue#10

## Sobe todos os ramos.
git push --all origin
```

Quando você sobe pela primeira vez um ramo local, a sua parte remota é criada. Assim, a instrução que colocamos acima criou um ramo remoto chamado `origin/issue#10`. Essa é a forma mais natural, e até despercebida, de criar ramos locais com cópias remotas, mas existem outras maneiras. Abaixo fazemos a criação do remoto a partir do local sem ser usando o *push*. Esses comandos precisam ser usados uma vez apenas.

```
## Conectar o local e remoto.
git branch --set-upstream issue#10 origin/issue#10

## Cria o ramo issue#11 a partir e vinculado ao devel remoto.
git checkout -b issue#11 origin/devel

## Cria ramo local issue#12 a partir do remoto com vínculo.
git checkout --track origin/issue#12

## Mostra as ligações entre pares de ramos: locais e remotos.
git branch -vv
```

¹⁰<https://git-scm.com/docs/git-pull>

¹¹<https://git-scm.com/docs/git-push>

5.4 Macanismos de colaboração

Os serviços web para Git, mesmo que você trabalhe sozinho, já são interessantes para ter uma cópia (*backup*) do seu projeto e disponibilizar seu trabalho. No entanto, um dos principais objetivos dos serviços web é permitir a colaboração entre pessoas. Já mencionamos o básico sobre recursos de colaboração quando falamos de *issue*, *fork* e *merge request*. Nas sessões a seguir, vamos nos aprofundar nesses três mecanismos chave para a colaboração via serviços web para Git.

5.4.1 Issues

De acordo com o dicionário Macmillan¹², *issue* significa um problema que precisa ser considerado. Também significa um assunto que as pessoas debatem ou o volume de uma revista. Cabe aqui a primeira definição.

Nos serviços web para Git, *issue* é um recurso da interface que permite que as pessoas abram um assunto/tópico referente ao projeto. Em geral, com os *issues* as pessoas externas ao projeto indicam um *bug* - uma falha a ser corrigida - ou sugerem que o projeto incorpore determinada característica desejável. O dono do projeto avalia a proposta e pode discuti-la logo em seguida, mantendo as mensagens reunidas e disponíveis para outros usuários que acompanham o projeto. Quando a proposta do *issue* for implementada, o *issue* é fechado. Mesmo assim, o *issue* continua disponível e pode ser referenciado no futuro, tanto para novos usuários quanto para incluir no *change log* do projeto (essa versão corrige o bug descrito no *issue*#43, por exemplo).

Quando um projeto é coletivo, como são alguns dos projetos no PET Estatística¹³, o recurso de *issue* pode ser usado de outra forma: como gerenciador de tarefas. Combinado com as *milestones*, cada membro cria um *issue* correspondente ao que vai fazer no projeto no período de uma semana. Com isso, todos os demais membros são notificados de que alguém já vai trabalhar na tarefa A do projeto, eliminando foco duplicado. O *issue* é parte do fluxo de trabalho do PET-Estatística que será descrito em outro capítulo.

As *milestones* são uma espécie de coleção de *issues* que tenham algo em comum. Considere por exemplo o projeto de um livro. As *milestones* podem ser os capítulos a serem escritos e os *issues* podem ser as seções. Se for um projeto de software, as *milestones* podem separar os *issues* referentes ao aperfeiçoamento da documentação e ao desenvolvimento do mesmo (escrita de código fonte).

Criar um *issue* é muito simples. Tanto no GitHub quanto no GitLab, existe um fácil acesso as *issues* do projeto. Na página dos *issues* você pode criar um novo *issue*, comentar em um já existente e até reabrir um *issue* que já foi fechado (comum quando acredita-se ter resolvido um *bug* mas que ainda não foi). Os *issues* são numerados sequencialmente e isso faz com que cada *issue* seja único dentro do projeto. Os serviços web até tem formas de fazer *hyperlinks* para os

¹²<http://www.macmillandictionary.com>

¹³<https://gitlab.c3sl.ufpr.br/groups/pet-estatistica>

issues dentro das mensagens e *commits*. No GitLab, usa-se um hash seguido do número identificador (e.g. #45) para fazer um *hyperlink* para um *issue*.

5.4.2 Fork

A palavra *fork*, como substantivo, representa forquilha, bifurcação. Esse recurso está presente nas interfaces web para permitir que usuários façam cópias livres de projetos de outras pessoas. Como essa cópia é do projeto em determinado instante, há grande chance de divergência entre cópia e original a partir desse instante, por isso é apropriado a palavra bifurcação.

As finalidades de um *fork* são duas: 1) ter uma cópia na qual se possa acrescentar modificações e enviar para o dono as contribuições no projeto original ou 2) usar a cópia como ponto de partida para um outro projeto, sem intenção de voltar ao projeto original.

No GitHub, o acesso ao *fork* é por um botão que fica mais à direita na linha de título do projeto. No GitLab, o botão de *fork* fica na página inicial do projeto logo acima do endereço para clonar. Em qualquer um desses serviços, uma vez logado, ao pedir um *fork*, uma cópia do projeto é criada no seu perfil.

Com o *fork* você pode colaborar como um terceiro em um projeto, ou seja, sem ser colaborador adicionado pelo dono. Nessa cópia você tem permissões de *owner* pois na realidade, embora seja uma cópia, ela é toda sua. Você faz sua cópia livre, trabalha como quiser e no prazo que quiser, e submete ao projeto original o desenvolvido na sua cópia. O dono do projeto não tem como saber por que você fez o *fork* (mas você pode criar um *issue*) nem quando irá concluir o que almeja. No entanto, quando concluir, para que seu trabalho seja incorporado ao original, você terá que fazer um *merge request* (isso só é possível entre usuários de um mesmo serviço web).

Uma vez com a cópia, você pode fazer um clone e começar a desenvolver os projetos. Se a sua intenção é submeter modificações ao original, seja ainda mais cuidadoso com as mensagens de *issue*. Escreva sempre no idioma original do projeto. Muitas vezes, antecipando esse tipo de colaboração, os usuários disponibilizam guias de contribuição (*contribution guide*) com instruções de como proceder para maior agilidade.

Os *branches* que criar serão na sua cópia e os *pushs* que fizer irão para o seu perfil. Quando o seu trabalho estiver concluído, você pode fazer um *merge request* que descreveremos a seguir. Se a sua intenção foi usar o *fork* como ponto de partida para um projeto independente, não se esqueça de dar os devidos créditos ao dono da cópia, mesmo que lá no futuro, o seu projeto e o original sejam completamente diferentes.

5.4.3 Merge Request

O *merge request* (requisição de mescla/fusão) é o recurso de colaboração chave. Ele serve para que pessoas da equipe (segundos) peçam para incorporar *branches* ao ramo principal (em geral o *master* ou o *devel*) e terceiros peçam para

incorporar o que foi desenvolvido no *fork*. O GitHub usa o termo *pull request* ao invés de *merge request* embora não exista diferença alguma.

Os trabalhos coletivos em projetos Git, para serem bem sucedidos, consideram algum esquema de trabalho. A maioria dos esquemas considera o desenvolvimento por *branches*. Nada mais justo, já que uma é uma característica do Git. Existem ramos permanentes, como o *master*, que recebem o desenvolvimento feito em *branches* auxiliares ou de demanda. Esses ramos de demanda, como o nome sugere, são criados para incorporar algo ao projeto e, portanto, não são permanentes - uma vez incorporados, são removidos.

Nos esquemas de trabalho, os membros são instruídos a fazerem o desenvolvimento nos ramos de demanda e jamais nos ramos permanentes. Ao concluir essa unidade de trabalho, esse *branch* é enviado para o servidor com um *push*

```
## Envia o desenvolvido em um ramo.  
git push origin issue#33
```

Na interface web, o membro faz um *merge request* desse ramo para um ramo permanente, no qual em projetos simples é o *master* mas em projetos maiores é usualmente o *devel*. Ao clicar em *merge request*, uma caixa de diálogo abre para que você liste as colaborações que o *branch* leva.

Em ambos os serviços, o *merge request* leva para uma página na qual você escolhe que ramo de demanda (doador) será incorporado a um ramo permanente (receptor). Ao confirmar os ramos envolvidos, tem-se uma caixa de texto destinada as informações sobre quais as modificações devem ser incorporadas. Elas servem para justificar, esclarecer e informar o responsável pelo *merge* sobre a incorporação. Quando o projeto é coletivo e não individual, um membro pode ser indicado como responsável pelo *merge*. O responsável pelo *merge* avalia as contribuições, se sentir necessidade vê as *diffs* nos arquivos, e pode colocar uma mensagem embaixo da sua com questionamentos e até adequações que precisarem ser feitas para aceitar o *merge request*.

Quando trata-se de *fork*, o processo ocorre de forma semelhante. A sequência de etapas é: fazer o *fork* do projeto para a sua conta, 2) clonar o projeto para sua máquina, 3) criar um *branch* e fazer o desenvolvimento nele, 4) subir o *branch* (*push*) para a sua conta e 5) fazer um *merge request* para incorporar as modificações. Na hora de escolher o ramo permanente (receptor) tem-se duas opções: 1) incorporar ao *master* (ou outro) da sua cópia ou 2) incorporar ao *master* (ou outro) do original. Outra opção é incorporar ao *master* da sua cópia e depois pedir um *merge request* do seu *master* para o *master* do original. Essa última é útil quando a quantidade de modificações é maior e portanto, o trabalho vai levar mais tempo.

Os próprios serviços web fazem o *merge* diretamente quando não existe conflito (*merge* do tipo *fast forward*). Isso facilita bastante o trabalho. Porém, não haver conflito de *merge* não significa que as modificações incorporadas estão funcionais, ou seja, as modificações feitas precisam ser testadas localmente para verificar se surtiram efeito. É possível habilitar o serviço Git para checar se o projeto é executável ou funcional. Esse recurso se chama integração contínua e veremos na próxima sessão.

Em caso de conflito de *merge*, tem-se que baixar os ramos (`git fetch`). Localmente pode-se comparar as diferenças entre eles para entender as fontes de conflito. Para isso são recomendáveis as interfaces para Git que serão discutidas no próximo Capítulo. Uma vez que o *merge* foi resolvido, deve-se fazer o *push* do ramo permanente (receptor) para o serviço web que já reconhece que o *merge* foi feito e fecha a requisição automaticamente.

Recomenda-se que os ramos de demanda sejam removidos após sua incorporação nos ramos permanentes. Isso torna o projeto mais claro e concentrado em ramos definitivos colhedores de desenvolvimento. Pode-se excluir o ramo de demanda incorporado direto pela interface, marcando uma caixa de confirmação sobre remover o ramo após o *merge*. Por linha de comando também é possível.

```
## Deleta o branch issue#33 no servidor (origin).
git push origin :issue#33

## Deleta o branch issue#33 localmente.
git branch -d issue#33

## Deleta a cópia do issue#33 que veio do origin.
git branch -dr origin/issue#33
```

Ao incorporar uma contribuição grande, é interessante marcar o *commit* vinculado à esse *merge* de uma forma destacável, como por exemplo, com uma *tag*.

```
## Lista as tags existentes.
git tag

## Adiciona uma tag anotada ao último commit.
git tag -a v1.0 -m "Versão 1.0 do software"
```

5.5 Integração contínua

Quando você está trabalhando em um projeto Git, cada *commit*, cada *branch*, contém algum desenvolvimento. Solucionar conflitos de *merge* não é uma tarefa complicada, principalmente se forem consideradas as ferramentas certas para isso, como será visto no próximo capítulo. No entanto, para cada *commit* tem-se a preocupação: será que o projeto está funcional?

Reprodutibilidade é uma questão de fundamental importância em projetos coletivos ou públicos. Existe a preocupação constante de que seu código (programa) seja executado (instalado) sem erros nos ambientes dos seus colaboradores ou usuários. É claro que o desenvolvimento do projeto visa aperfeiçoá-lo mas o tempo todo estamos sujeitos a fazer modificações que induzem erros e alguém encontra erros não esperados (*bugs*).

Monitorar erros no projeto em desenvolvimento não é uma tarefa fácil, principalmente em trabalhos coletivos nos quais cada colaborador tem um ambiente

de trabalho. Em uma situação ideal, alguém teria que testar se o projeto corre sem erros (código executa, software instala) em uma máquina cliente (com os requisitos mínimos exigidos), toda vez que um *commit* fosse feito, já que o *commit* indica que modificações foram feitas. Então, se correu sem erros, avisar à todos que podem prosseguir, mas se falhou, informar a equipe, encontrar e corrigir a falha.

Não é raro algo ser bem sucedido no ambiente em que foi desenvolvido e apresentar falhas no ambiente de outra pessoa. O melhor jeito de antecipar erros é testar em um ambiente virgem, uma máquina cliente que contenha apenas os requisitos mínimos necessários ao projeto.

A Integração Contínua (IC) é a solução desse problema. A ideia é manter o repositório Git continuamente integrado à um ambiente cliente que faz verificações no projeto a cada novo *push*.

Fazer integração contínua no GitHub e GitLab, embora o conceito seja o mesmo, tem algumas diferenças. No GitHub existem diversos serviços de IC, sendo eles terceirizados. Já o GitLab CE têm o serviço próprio de IC a partir da versão 8.0. Apresentaremos cada um deles na sessão seguinte.

Independente do serviço web, as principais vantagens da IC¹⁴ são:

- 1) Economia de tempo com supervisão de código e procura de falhas. Com a verificação mais frequente, a cada *commit/push*, menos tempo é gasto na correção de bug, que devem ser também menores, assim deixando mais tempo para desenvolver o que interessa.
- 2) Verificação em um ambiente virgem sem efeito de variáveis de ambiente locais.
- 3) Verificação em vários ambientes (sistemas operacionais, arquiteturas, dependências e versões);
- 4) Informação coletiva e imediata à equipe da causa de falha na hora que ela ocorre, o que aumenta a visibilidade, facilita a comunicação e direcionamento de esforços.
- 5) Indicação de sucesso ou não na home do projeto e nos *branches* disponível antes do merge;
- 6) Entrega de versões estáveis, documentação e arquivos acessórios com *continuous deployment*, o que facilita o empacotamento e disponibilização do projeto.
- 7) Custo computacional reduzido já que é feito em um servidor dedicado.
- 8) Acaba saindo mais barato do que parar o projeto para corrigir um erro.

O que deve ficar claro é que a integração contínua não elimina erros, mas faz com que eles sejam mais fáceis de identificar e corrigir.

Sem a automação, os espaços entre verificações podem ficar longos. Encontrar um *bug* em tantos *commits* é mais difícil, encontrar no código é mais ainda. Pode atrasar a entrega do projeto e comprometer a receita e popularidade. Integrações periódicas são mais fáceis e leves.

O fluxo de trabalho de um repositório com IC é basicamente assim:

¹⁴<https://about.gitlab.com/gitlab-ci/>

- 1) Os desenvolvedores trabalham no projeto em seu *workspace* privado/local;
- 2) Periodicamente fazem *commits* e *pushs* para o repositório remoto;
- 3) O serviço de IC monitora o repositório toda vez que modificações ocorrem;
- 4) O serviço de IC corre todos os testes de inspeção que foram configurados (instala dependências, executa scripts, cria/transfere arquivos, etc);
- 5) O serviço de IC disponibiliza produtos da verificação, como binários de instalação e documentação (no caso de sucesso) ou log de execução (no caso de falha);
- 6) O serviço de IC assinala no repositório Git o *status* da verificação: sucesso/fracasso;
- 7) O serviço de IC informa a equipe dos resultados por mensagem de e-mail ou *web hooks*, como o Slack;
- 8) A equipe toma das providências cabíveis na primeira oportunidade, como corrigir a falha ou anunciar o sucesso;
- 9) O serviço aguarda por mais modificações;

5.5.1 GitHub

Embora exista uma lista grande de serviços de integração contínua disponíveis para repositórios GitHub, um dos mais usados é o Travis CI¹⁵. Travis CI (*continuous integration*) é um serviço *free* e *open source* destinado à integração contínua para projetos **públicos** no GitHub.

Para vincular um projeto no GitHub com IC no Travis CI, você precisa logar no <https://travis-ci.org/> com a sua conta do GitHub. Assim que você logar, dê autorização para acessar seus repositórios e em uma lista você deve marcar os que usarão o serviço. A próxima etapa é criar um arquivo `.travis.yml` na raiz do seu repositório Git. Esse arquivo oculto especifica todas as instruções que devem ser feitas a fim de verificar seu repositório. Se seu repositório é um pacote R, por exemplo, esse arquivo vai ter instruções de instalação do pacote.

Cada projeto, cada linguagem, têm uma forma particular de ser testada. Para pacotes R, o Travis CI tem uma documentação de orientação: <https://docs.travis-ci.com/user/languages/r/>. Além disso, uma boa prática em ver exemplos em uso, como o `.travis.yml` dos pacotes R `knitr`¹⁶ e `devtools`¹⁷ que estão na lista dos mais utilizados.

Para todas as linguagens e objetivos têm-se exemplos de arquivos `.travis.yml`. Para o Emacs e bibliotecas lisp visite <https://github.com/rolandwalker/emacs-travis>, <https://github.com/Malabarba/elisp-bug-hunter> e <https://github.com/abo-abo/tiny>. Para projetos em C++ e python, assim como o R, o Travis CI tem uma documentação introdutória. Visite <https://docs.travis-ci.com/user/language-specific/> para ver a lista de documentações.

¹⁵<https://travis-ci.org/>

¹⁶<https://github.com/yihui/knitr>

¹⁷<https://github.com/hadley/devtools>

Além das linguagens de programação, é possível testar inclusive a compilação de um documento LaTeX: <http://harshjv.github.io/blog/setup-latex-pdf-build-using-travis-ci/>.

5.5.2 GitLab

A Integração Contínua passou a fazer parte do GitLab CE na versão 8.0¹⁸, lançada em setembro de 2015. Diferente do GitHub, essa versão do GitLab tem o IC de forma nativa. Você pode configurar servidores externos para executarem as verificações ou pode fazê-las no mesmo servidor que roda o GitLab.

Alan Monger¹⁹ descreve como configurar um servidor Ubuntu no Digital Ocean²⁰ para verificar repositórios hospedados no <https://gitlab.com/>.

Segundo ele, o primeiro passo é acessar <https://gitlab.com/ci/> para adicionar o projeto à integração contínua. Na sua matéria, Alan descreve como instalar um *Virtual Private Server* com Ubuntu 14.04 no Digital Ocean. Em seguida instala o *runner* (`gitlab-ci-multi-runner`) o configura e habilita. Por fim, o arquivo `.gitlab-ci.yml`, que especifica o que deve ser executado, é criado na home do repositório.

O GitLab do LEG²¹ tem o CI embutido pois é a versão mais recente do serviço. Essa servidora é na realidade um *desktop* com Ubuntu Server 14.04. É patrimônio da UFPR de responsabilidade do LEG mas tem uso compartilhado com o PET Estatística e colaboradores do Laboratório. Desde a disponibilização do GitLab, em julho de 2015, mantemos artigos, materiais de ensino (aulas, provas, scripts), tutoriais e matérias de blog em devolvimento colaborativo, além de pacotes R.

Nessa servidora, criamos um usuário chamado `gitlab-runner` com o qual fazemos as integrações contínuas. Toda vez que um projeto com o arquivo `.gitlab-ci.yml` recebe um *push*, as instruções nesse arquivo executam a IC com o usuário `gitlab-runner`. Tecnicamente podemos correr todo tipo de comando ou programa disponível no servidor em questão. Até então, os repositórios com IC são só dois: `legTools`²² e `mcglm`²³, dois pacotes R.

5.5.2.1 O arquivo de configuração `.gitlab-ci.yml`

A documentação oficial sobre como usar o arquivo `.gitlab-ci.yml` encontra-se em <http://doc.gitlab.com/ce/ci/yaml/README.html>.

O arquivo `.gitlab-ci.yml` fica na raiz do projeto. Seu conteúdo define todo o processo de verificação do seu repositório a partir de uma série de instruções, como execução de comandos diretos ou execução de scripts. Abaixo tem-se um exemplo simples de `.gitlab-ci.yml`.

¹⁸<https://about.gitlab.com/2015/09/22/gitlab-8-0-released/>

¹⁹<http://alanmonger.co.uk/php/continuous/integration/gitlab/ci/docker/2015/08/13/continuous-integration-with-gitlab-ci.html>

²⁰<https://www.digitalocean.com/>

²¹<http://git.leg.ufpr.br/>

²²<http://git.leg.ufpr.br/leg/legTools>

²³<http://git.leg.ufpr.br/wbonat/mcglm>

```
job1:
  script: "teste_instalacao.sh"
```

```
job2:
  script:
    - pwd
    - ls -a
```

Neste exemplo existem dois *jobs* (tarefas). Cada um deles corre independente e podem ser executados simultaneamente. O primeiro executa um *script* shell e o segundo comandos *shell* em uma lista. Porém, tais arquivos podem ser bem mais complexos, com campos além do `script:`. Os campos a seguir são todos opcionais:

- `image`: para especificar uma imagem *docker*²⁴. O tutorial de Alan Monger considera esse campo.
- `services`: também refere ao *docker*. Tais campos são de uso menos frequente, porém existe uma série de vantagens neles. A documentação oficial sobre isso encontra-se em <http://doc.gitlab.com/ce/ci/docker/README.html>.
- `before_script`: define comandos/scripts a serem executados antes dos comandos verificadores. São normalmente instruções de preparação das quais não se espera falhas pois não devem depender do projeto.
- `stages`: define a ordem de execução dos *jobs* para uma cadeia de execução condicional. *Jobs* de mesma ordem ou do mesmo estágio são executados paralelamente mas àqueles à frente só são executados se houver sucesso dos predecessores.

```
stages:
  - construcao
  - teste
  - entrega

job_construcao:
  script: "construcao.sh"
  stage: construcao

job_test:
  script: "teste_codigofonte.sh"
  script: "teste_documentacao.sh"
  stage: teste

job_entrega:
  script: "compacta_transfere.sh"
  stage: entrega
```

²⁴<https://www.docker.com/>

- `variables`: serve para criar variáveis de ambiente que podem ser usados por todos os comandos e scripts. Tais variáveis podem armazenar senhas de acesso, necessárias por exemplo para instalação de componentes e acesso a bancos de dados.
- `cache`: indica os diretórios e arquivos que serão mantidos entre os *jobs* (builds), possivelmente porque são aproveitados futuramente.

Com exceção dos nomes listados acima, um *job* pode ter qualquer nome, desde que seja exclusivo. Dentro de um *job* tem-se também uma lista de campos disponíveis para configurá-lo:

- `script`: especifica script *shell* ou uma lista de comandos a serem executados.
- `stage`: já mencionado anteriormente, serve para dar a ordem de execução dos *jobs*. Vários *jobs* podem ser do mesmo estágio e nesse caso são executados paralelamente.
- `only` e `except`: servem para restringir a execução do *job*, incluindo ou excluindo, para uma lista de referências Git, como *branches* ou *tags*. Esse campo permite expressões regulares, úteis para incluir (excluir) ramos representáveis por *regex*, como são os ramos de desenvolvimento do nosso workflow, iniciados com *issue*.

```
job_test:
  script: "teste_codigofonte.sh"
  script: "teste_documentacao.sh"
  stage: teste
  only:
    - /^issue.*$/ ## Filtra os que começam com issue.
```

- `tags`: são usadas para indicar o *runner* da lista de disponíveis. Na configuração dos *runners* pode-se atribuir *tags* que servem justamente para esse tipo de pareamento.
- `allow_failure`: indica *jobs* que, mesmo que falhem, não serão considerados para dar estampa de sucesso ou falha, ou seja, é um teste mas não crucial.
- `when`: é um comando que dispara excussões condicionais ao sucesso do *job* anterior
 - `on_failure`: são instruções executadas quando algum *job* do estágio anterior falhou.
 - `on_success`: são instruções executadas quando todos os *jobs* do estágio anterior foram bem sucedidos.
 - `always`: executados sempre.
- `cache`: especifica arquivos e diretórios mantidos entre um *job* e outro.

No caso do pacote `legTools`, o arquivo `.gitlab-ci.yml` do repositório tem o seguinte conteúdo:

```
job_R:
  script:
    - echo $HOME
    - Rscript -e 'getwd(); .libPaths(); sessionInfo()'
    - Rscript -e 'library(devtools); check()'
    - Rscript -e 'library(devtools);\n
      .libPaths(path.expand("~/R-tests/legTools"));\n
      install(local = FALSE)'
```

Estas são instruções em *shell* que chamam o R com expressões passadas para `Rscript -e` para fazer a instalação do pacote. Na ocorrência da primeira falha, o processo é logicamente interrompido e os colaboradores podem ver a causa em <http://git.leg.ufpr.br/leg/legTools/builds>. No caso do *build* correr sem falhas, tem-se uma estampa de *success*. Essa estampa também vai aparecer no `README.md` na página de rosto do projeto, basta para isso colocar o seguinte conteúdo no arquivo.

```



```

Caso queira a estampa para outros ramos, é só acrescentá-los.

5.5.2.2 Runners

Em poucas palavras, um *runner* é uma máquina que executa o *build* por meio do GitLab CI. Um *runner* pode ser específico de um projeto ou pode ser um *runner* compartilhado, disponível à todos os projetos. Estes últimos são úteis à projetos que tem necessidades similares, como todos àqueles projetos que são pacotes R, por exemplo. Isso facilita a administração e atualização. Os específicos, por outro lado, atendem projetos com demandas particulares.

Apenas usuários com permissões de *admin* podem criar *runners* compartilhados.

Em   marque o campo *active*. Isso vai criar 6 novas entradas no menu da esquerda:

- *Runners*: contém instruções de como usar os *runners* específicos e compartilhados.
- *Variables*: define variáveis que são omitidas no *log* do *build*, útil para passar senhas.
- *Triggers*: servem para TODO
- *CI Web Hooks*: esse *web hook* pode ser usado para criar eventos quando o *build* é concluído.
- *CI Settings*: configurações gerais.
- *CI Services*: permite integrar o CI com outros serviços, como e-mail e Slack.

Para habilitar um *runner*, é necessário instalar o `gitlab-ci-multi-runner`. O repositório oficial do *GitLab Runner*²⁵ contém as instruções de instalação e configuração e a documentação oficial *runners*²⁶ indicando como tirar melhor proveito do recurso.

²⁵<https://gitlab.com/gitlab-org/gitlab-ci-multi-runner>

²⁶<http://doc.gitlab.com/ce/ci/runners/README.html>

Capítulo 6

Ferramentas gráficas

No Git, todo o gerenciamento do projeto é realizado via *CLI* (*Command line interface*), linhas de comando interpretadas, geralmente pelo *bash*. Isso confere um maior controle e segurança nas ações realizadas, mas em muitas situações os comandos e *outputs* Git não se apresentam de forma tão amigável, seja pela difícil memorização ou pela interatividade limitada.

Os comandos mais usuais como `git add` e `git commit` se tornam simples, pois mesmo para um usuário iniciante eles fazem parte do cotidiano em um projeto sob versionamento Git. Porém, algumas situações não ocorrem com frequência, como por exemplo voltar a versão de um arquivo ou do repositório requerem comandos que são pouco utilizados e para realizá-las é necessário a consulta de algum material. Outra situação em que a utilização dos comandos é dificultada, ocorre em projetos grandes, uma vez que muitos arquivos são alterados simultaneamente. Neste caso o procedimento de *commit* se torna trabalhoso, pois é necessário listar todos os arquivos que fazem parte de um *commit* no comando `git add`. Uma última situação exemplo em que o uso de *CLI* não parece satisfatório é na comparação de arquivos, já usamos o comando `git diff` no capítulo 3 e o *output* deste comando foi de simples visualização, mas em arquivos grandes (com muitas linhas) a navegação para verificar as alterações do arquivo não é tão amigável. Para facilitar essas e outras situações surgem as *GUI's* (*Graphical User Interfaces*), interfaces gráficas para o usuário incorporar comandos Git em *widgets* (botões, caixas de texto etc.) dispostos em uma janela gráfica de seu sistema operacional.

Neste capítulo apresentamos as principais *GUI's* para projetos Git em diferentes plataformas, sistemas UNIX, Mac OS X e Windows. Seccionamos em dois conjuntos de interfaces. O primeiro chamado de **Interfaces Git** refere-se às ferramentas para alterações e visualizações de arquivos no repositório a fim de facilitar as atividades cotidianas. Já o segundo, **Interfaces de comparação**, representam as que objetivam facilitar a visualização e edição de arquivos com base em suas diferentes versões. Detalhes de download, instalação e exemplos da utilização destas interfaces no fluxo de trabalho de um projeto são descritos.

6.1 Interfaces Git

Neste material chamaremos de **Interfaces GIT** as *GUI's* para gestão de um repositório. Estas facilitam a utilização das principais instruções **Git** (`git add`, `git commit`, `git push`, `git pull`), visualização dos arquivos e alterações no repositório.

6.1.1 git-gui

Baseada em *Tcl/Tk*, a *GUI* chamada `git gui` é mantida como projeto independente do Git, mas as versões estáveis são distribuídas junto com o programa principal, portanto não é necessário o download e instalação. A interface é voltada para realizar alterações no repositório, desde as mais simples como *commitar* arquivos até as mais específicas como voltar estágios ou reescrever o último *commit* (muito útil quando notamos erros de gramática logo após a submissão). Nesta seção abordaremos apenas as alterações mais comuns no repositório.

A `git gui` no Windows, pode ser aberta pelo menu iniciar. Nesta plataforma, ao instalar o Git (conforme visto no capítulo 2), optamos pelos componentes **git BASH** e **git GUI**. Assim estas aplicações ficam disponíveis para uso. Em sistemas UNIX, a interface pode ser instalada via terminal, também apresentada no capítulo 2:

```
## Instalando a git gui
sudo apt-get install git-gui
```

Ainda em sistemas Unix podemos criar um *alias* (criar ou editar adequadamente um arquivo em `/usr/share/applications`) para que a `git gui` fique listada junto às aplicações do sistema. Porém, de forma geral, independente da plataforma de trabalho, a `git gui` pode ser iniciada a partir de um terminal bash, com o comando:

```
git gui
```

Para exemplificar a utilização desta interface vamos alterar alguns arquivos do repositório `meu1repo` criado no capítulo 3.

```
## Destaca título no README
sed -i "2i\-----" README.txt
sed -i "1i\-----" README.txt

## Destaca título no porqueLinux
```

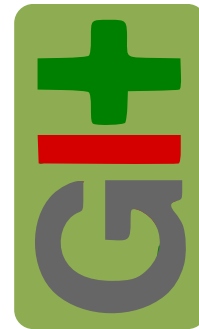


Figura 6.1: Logo usada para a `git-gui`, também é uma das logos do próprio GIT


```

sed -i "2i\-----" porqueLinux.txt
sed -i "1i\-----" porqueLinux.txt

## Corrige nome do autor citado
sed -i "s/Lunus/Linus/g" README.txt

## Cria um novo arquivo TODO.txt
echo "
Lista de afazeres:
-----
* tarefa 1
* tarefa 2
* tarefa 3" > TODO.txt

```

Agora visualizando o estado do repositório após nossas modificações, ainda via terminal:

```
git status
```

On branch master

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: README.txt

modified: porqueLinux.txt

Untracked files:

(use "git add <file>..." to include in what will be committed)

TODO.txt

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

A partir daqui poderíamos seguir o fluxo de gestão de versões via terminal, apresentado no capítulo 3. Mas faremos isso agora via interface `git gui`.

A interface `git gui` se apresenta de forma simples, o que facilita sua utilização. Na figura 6.2 destacamos as quatro áreas que compreendem a interface. Na primeira porção temos listados os arquivos presentes no *working directory*, os arquivos criados aparecem com ícone em branco e os modificados com linhas em azul, aqui a interface implementa interativamente o comando `git add`, pois ao clicar no ícone de um arquivo ele é automaticamente adicionado a *staging area*. Na segunda parte são listados os arquivos na *staging area* com ícone de *check mark*. Na terceira parte temos a implementação do comando `git diff` para qualquer arquivo selecionado. Com destaque de cores, a interface apresenta em vermelho as deleções e em verde as adições. Por fim temos no canto inferior direito a área para escrever *commits* com botões para submissão

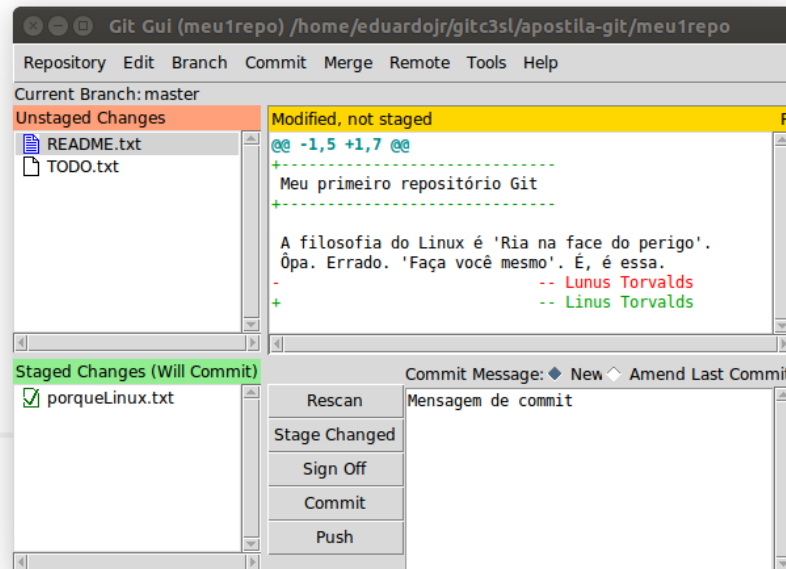


Figura 6.2: Screenshot da execução do programa git-gui

de ação. Um detalhe importante do `git gui` é que o idioma do sistema operacional é verificado para sua construção, ou seja, os botões da interface na figura 6.2 são *push*, *commit*, *sign off*, etc, pois o idioma do sistema operacional em que essa interface foi executada é o inglês. Para outros idiomas as mensagens podem sofrer alterações.

Além das quatro áreas principais da interface, que facilitam interativamente atividades como `git status`, `git diff`, `git add`, `git commit` e `git push`, temos mais implementações no menu da interface para procedimentos não cotidianos. Essas implementações podem ser acessadas com um simples clique e são auto-explicativas.

6.1.2 gitk

Pioneira dentre as interfaces gráficas, `gitk` foi a primeira *GUI* implementada. Também escrita em *Tcl/Tk*, esta *GUI* tem como objetivo a apresentação do histórico de um projeto. A `gitk` é incorporada ao principal repositório do Git, portanto nas instalações completas, esta interface fica disponível sem ser necessário download e instalação. Nesta seção apresentamos a `gitk`, detalhando a disposição dos elementos nesta interface que se mostra muito útil na visualização de projetos.

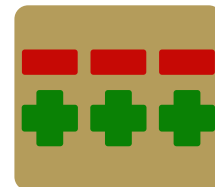


Figura 6.3: Logo da interface `gitk`, símbolos de supressão e adição são característicos dos logos GIT

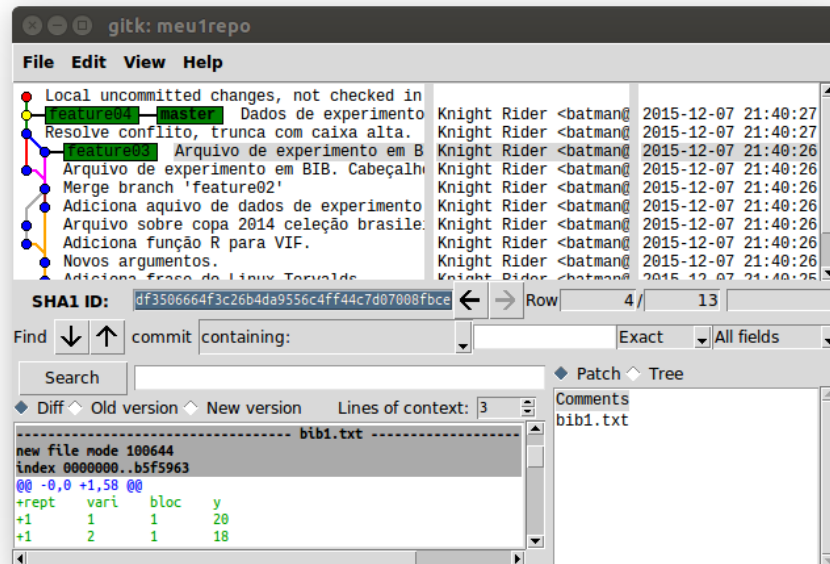


Figura 6.4: Screenshot da execução do programa gitk

A gitk trabalha em conjunto com a git gui. Em git gui podemos fazer alterações de forma rápida e visual nos arquivos que estão na *staging area* e *working directory*, porém para visualizar o histórico completo de *commits* com ramificações, marcações e demais detalhes, recorreremos à gitk, uma prova disso é que no menu da git gui temos um atalho para a gitk `Repository >> Visualize History`. Essa interface se mostra muito útil também como ferramenta de aprendizagem Git, uma vez que visualizar de forma gráfica as alterações que os comandos realizados causam no projeto, torna mais fácil a compreensão dos mesmos.

gitk, assim como a git gui pode ser chamada através da linha de comando:

```
gitk
```

Para exemplificar a disposição dos elementos nesta interface, seguimos com as alterações feitas na seção anterior, lembrando que temos todas as alterações já realizadas no capítulo 3 e ainda duas modificações e uma inclusão de arquivo não *commitados*. Visualizando a interface gitk chamada neste estado do repositório temos:

Perceba na figura 6.4 que esta interface é mais completa do que a git gui no que diz respeito à informação. Dividida em apenas duas partes, a gitk apresenta na primeira todo o histórico do projeto, contempla uma implementação visual e agradável do comando `git log --graph`. No gráfico apresentado

na parte superior, as bolinhas em azul representam *commits* passados, a de amarelo indica o estado atual do repositório e em vermelho são as modificações no *working directory*. Ao lado estão os autores dos respectivos *commits* e o momento em que foram feitos. Na parte inferior da interface temos o detalhamento do *commit* selecionado na parte superior. As informações contidas aqui vão desde identificador do *commit* (*SHA1 ID*), diferença das modificações referenciadas com relação ao estado anterior do repositório até a listagem dos arquivos atingidos pelo *commit* selecionado.

Além da excelente apresentação visual do repositório Git, a interface *gitk* também permite algumas alterações. Clicando com o botão direito de seu *mouse* em qualquer *commit* listado, podemos criar *tags*, reverter o repositório neste estado, criar um ramo a partir do *commit* dentre outras opções possíveis através da interface.

6.1.3 Outras Interfaces

6.1.3.1 *gitg* e *gitx*

Estas duas interfaces tentam juntar em uma única as opções proporcionadas pela *git gui* e pela *gitk*. Os layouts e as propostas são similares, a diferença está na portabilidade. A *gitg* é implementada em *GTK+* e está disponível para sistemas UNIX e a *gitx* foi implementada para Mac OS seguindo o estilo de aplicativos deste sistema operacional. De forma geral não há detalhes a serem repassados sobre estas interfaces uma vez que as possibilidades já foram listadas nas seções sobre *git gui* e *gitk*.

6.1.3.2 *RabbitVCS*

RabbitVCS é uma coleção de ferramentas gráficas para navegadores de arquivos do sistema LINUX que permitem o acesso simples e direto aos sistemas de controle de versão Git e/ou Subversion. Não se caracteriza como interface, porém altera a visualização no navegador de arquivos de diretórios sob versionamento, além de dispor de ações implementadas nas opções do menu quando pressionado o botão direito do mouse.

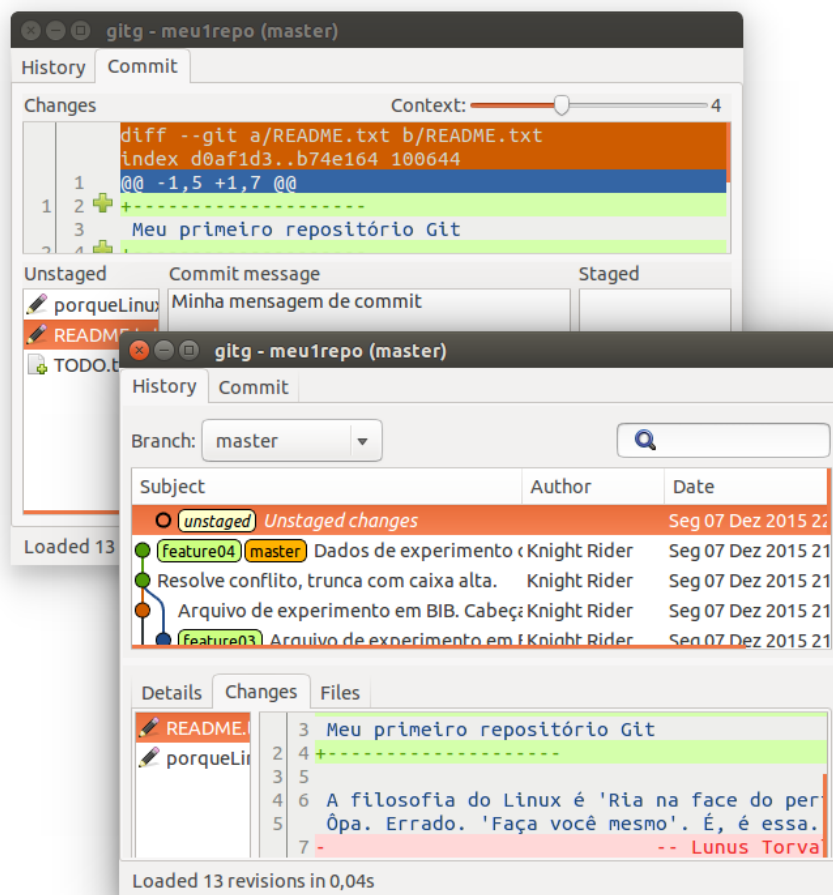


Figura 6.5: *Screenshot* da execução do programa `gitg`

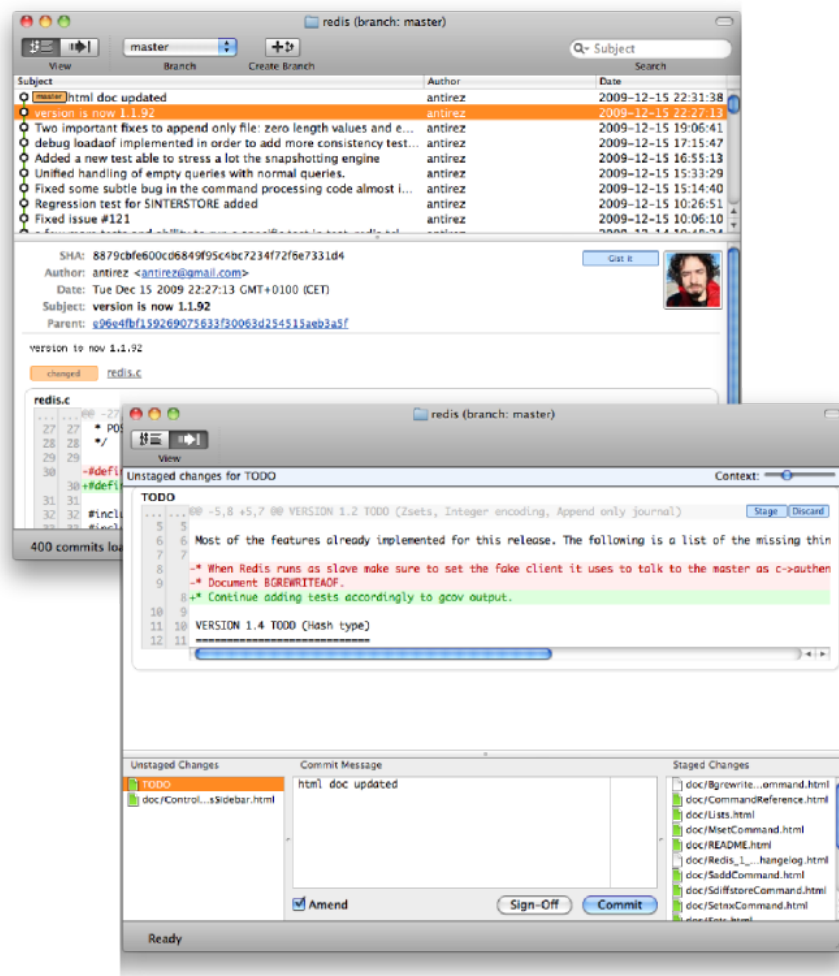


Figura 6.6: Screenshot do programa gitx

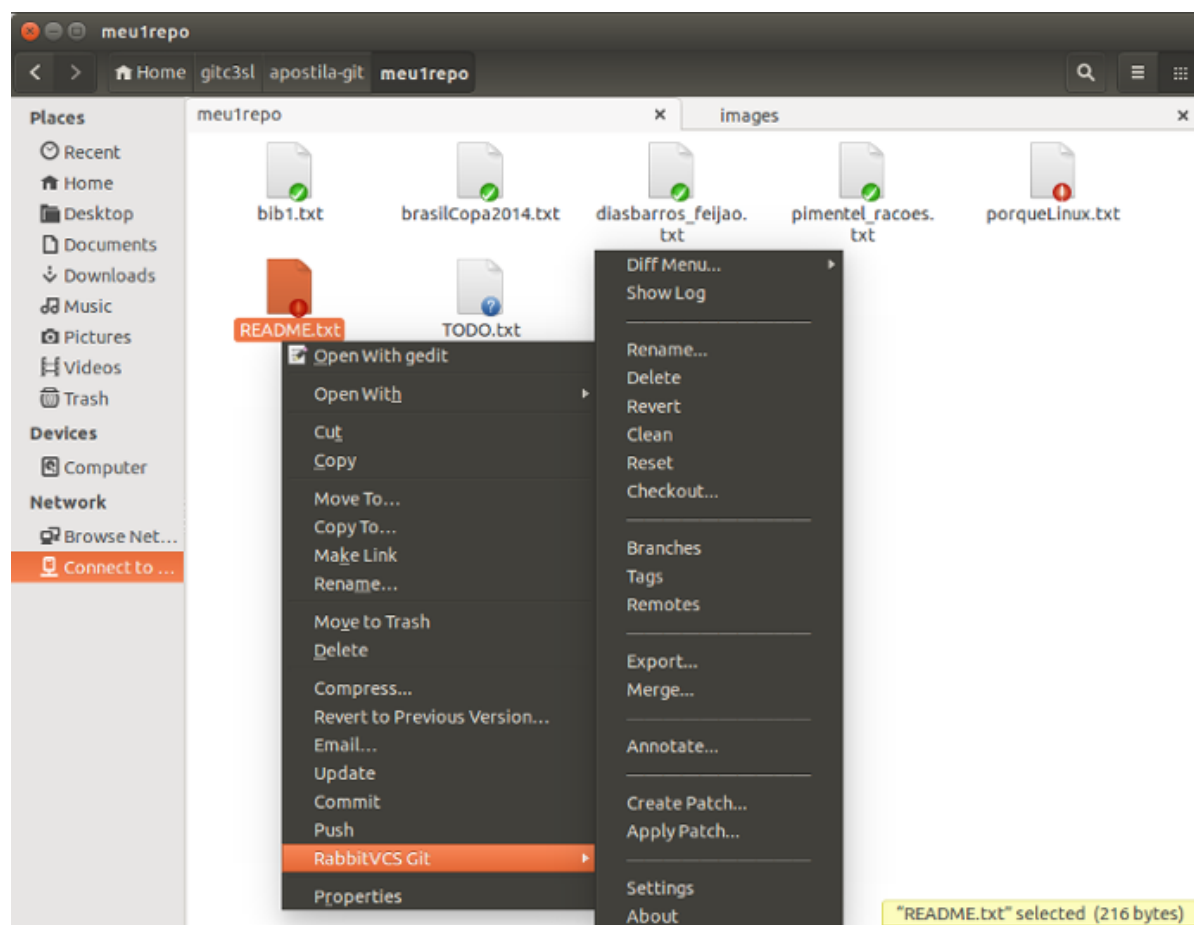


Figura 6.7: Screenshot do Navegador Nautilus com uso do RabbitVCS

Na figura 6.7 temos o *screenshot* do repositório `meurepo` no navegador de arquivos `nautilus` (padrão do sistema Ubuntu 14.04). Perceba que com essa interface os ícones de arquivos e pastas no navegador ganham destaque com um outro pequeno ícone na parte inferior. Estes pequenos ícones indicam o estado do arquivo sem precisar recorrer ao terminal, ou seja, temos um `git status` no próprio navegador de arquivos. Além disso `RabbitVCS` complementa o menu de opções acessados com o botão direito do mouse. Essas opções são completas, vão desde *commits*, criação de *branches* e *tags*, reverter o estado do repositório, até atualizar com a versão remota, entre outras.

6.1.3.3 git-cola

Esta também é uma interface alternativa que se destaca por ser completa e portátil (disponível para sistema LINUX, Windows e Mac). Implementada em *python*, a `git-cola` é uma alternativa à `git gui` e contém praticamente os mesmos elementos para alterações no repositório. Como a `git gui` se auxilia da `gtk` para visualização, a `git-cola` também tem uma interface de apoio, chamada de `git-dag` que vem instalado junto ao `git-cola`.

Perceba pela figura 6.8 que as opções das interfaces são similares as apresentadas em `git gui` e `gtk`. As interfaces `git-cola` e `git-dag` se destacam pela fácil manipulação do layout exibido, além de deixar a interface mais intuitiva possível. Como destaque em implementação de funcionalidade `Git`, a `git-cola` se sobressai com relação à `git gui` na possibilidade de execução do comando `git rebase` via interface.

6.1.3.4 Plugins e extensões para editores

Muitas vezes é inconveniente trabalhar com códigos fonte em um editor e ter que abrir um terminal *bash* em outra janela do sistema operacional para verificar o sistema de versionamento, realizar *commits*, etc. Felizmente alguns editores possuem um sistema **Git** integrado, seja por meio de *plugins* adicionais instalados ou pela opção nativa do editor.

Destacamos aqui dois editores, comumente utilizados pela comunidade estatística, que possuem os comandos **Git** integrados à sua interface. São eles, o `emacs`, o qual temos as opções de *buffers* no editor onde podemos abrir uma instância *shell* e executar os comandos **Git** junto com o desenvolvimento do código fonte. Além disso uma extensão poderosa chamada `magit`¹ está disponível e em desenvolvimento para o uso no `emacs`, esta extensão proporciona opções de comandos e visualização em um *buffer* do editor que facilita o trabalho de versionamento. Outro editor também muito utilizado em Estatística, talvez o mais utilizado pela comunidade, é o `RStudio` que também implementa em sua interface vários comandos, assim como as interfaces anteriormente descritas e tarefas não triviais, uma chamada do terminal *Shell* é possível dentro do aplicativo. Devido ao seu grande uso, o `RStudio` terá uma seção específica onde as diversas ferramentas serão exploradas, com exemplos e ilustrações voltadas para a comunidade estatística.

¹Informações em <http://magit.vc/>

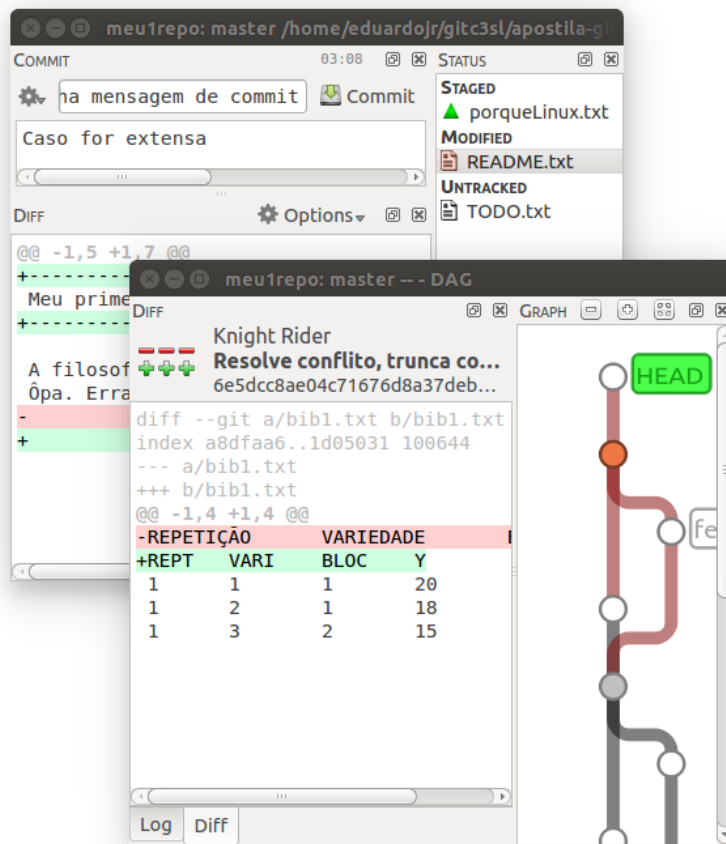


Figura 6.8: Screenshot dos programas git-cola e git-dag

6.2 Interfaces de comparação

Uma das principais vantagens do **Git** é a possibilidade de trabalho paralelo por dois ou mais usuários ou por ramos de desenvolvimento. E como qualquer desenvolvimento paralelo, desejamos ao final do trabalho, mesclar as contribuições realizadas lado a lado. Como vimos no capítulo 3 isso é feito através do comando `git merge ramo_desenvolvimento` para ramos locais e `git push origin` quando estamos trabalhando em equipe e as contribuições são enviadas para um servidor remoto, capítulo 4. Porém, quando a mesma porção de um mesmo arquivo é alterada em duas instâncias distintas (ramos diferentes, usuários diferentes etc.) ocorrem conflitos e vimos como o **Git** os sinaliza para que possamos resolvê-los. Nesta seção mostraremos como as interfaces gráficas dedicadas à resolução de conflitos na mesclagem e à visualização da diferença de arquivos em estados diferentes do repositório podem nos auxiliar.

Há vários programas que objetiva a comparação visualmente agradável de arquivos. Aqui iremos abordar o programa `meld`, que é multiplataforma *open source* e tem várias facilidades implementadas, porém outras alternativas serão indicadas, e devido à equidade de objetivos todos os comentários feitos para o `meld` podem ser adotadas para os demais.



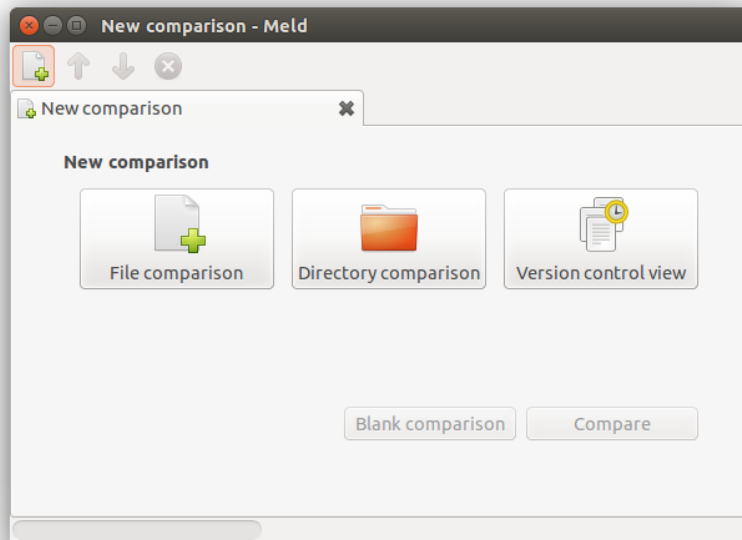
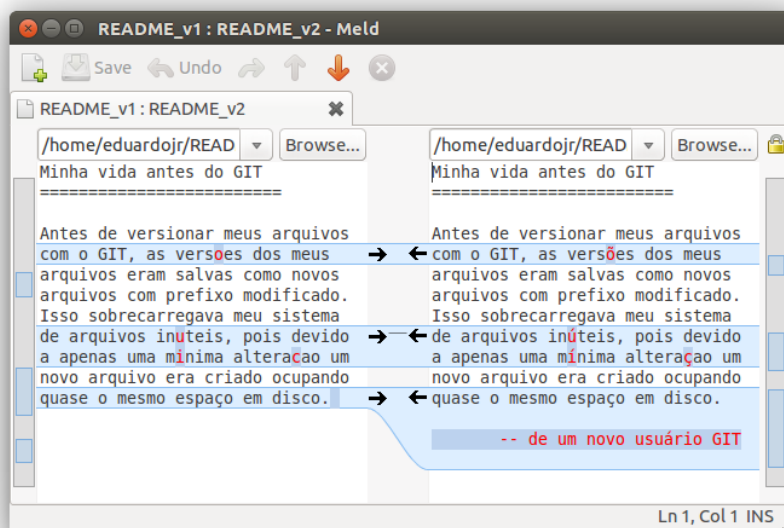
O programa `meld` é implementado em *python* e se denomina como “uma ferramenta de diferenciação e mesclagem voltada para desenvolvedores”, o programa pode ser baixado para as plataformas UNIX, Mac OS X e Windows através do endereço <http://meldmerge.org/>. O `meld` não é uma ferramenta específica para o **Git**, como as apresentadas na seção 6.1, porém é permitido e será usado para comparar versões de arquivos ou repositórios, mas vamos começar apresentando o `meld` como programa independente.

Figura 6.9: Logo do aplicativo `meld`, não há descrições sobre o seu significado, mas nos parece representar mistura, como mistura de arquivos

Inicializando o programa, sua tela inicial deverá ser similar a apresentada na figura 6.10, aqui estamos utilizando um sistema operacional Ubuntu 14.04. A partir daí podemos escolher quaisquer dois (ou três) arquivos ou diretórios para comparação.

A partir da escolha, o `meld` apresenta os arquivos ou diretórios lado a lado para comparação e destaca as porções dentro dos arquivos que estão discordantes. A figura 6.11 apresenta a comparação de dois arquivos, que salvamos como `README_v1` e `README_v2` (relembrando os velhos tempos antes de conhecermos o **Git**).

Com isso já podemos notar onde utilizar esta ferramenta no fluxo de trabalho de um projeto sob versionamento. Vamos então voltar ao nosso projeto `meu1repo`, iniciado no capítulo 3 e alterado na seção 6.1 (Interfaces **Git**). As alterações realizadas não foram salvas, então podemos visualizá-las no `meld`. A inicialização do programa pode ser feita via linha de comando `git difftool`,

Figura 6.10: Screenshot do tela inicial do programa *meld* RabbitVCSFigura 6.11: Screenshot de comparação de arquivos com programa *meld* RabbitVCS

só temos que informar o programa a ser utilizado com a opção `-t` (abreviação de `--tool`). Nos sistemas UNIX, o programa pode ser lançado apenas através do nome `meld`.

```
## Compara o arquivo README (UNIX)
git difftool -t meld README.md
```

Para utilização em sistemas Windows, programas externos ao Git devem ser informados no arquivo de configuração (`.gitconfig`). Abaixo configuramos, via linha de comando, este arquivo para usarmos o `meld` como ferramenta de comparação - `difftool`, tanto para usuários Unix como usuários Windows:

```
## Define localmente o meld como ferramenta padrão de diff
##-----
## Unix.
git config diff.tool meld

##-----
### Windows.
git config diff.tool meld
git config difftool.meld.cmd '"path/Meld.exe" $LOCAL $REMOTE'
```

onde `path` é o caminho para o arquivo executável do programa `meld`. `$LOCAL` representa o arquivo na sua versão local e `$REMOTE` na sua versão remota. Assim o programa pode ser lançado apenas com o comando:

```
## Compara o arquivo README (WINDOWS)
git difftool README.md
```

Na figura 6.12 temos o *screenshot* do programa após executado o `difftool`. Nesta figura temos a mesma informação trazida pelas interfaces onde implementam o comando `git diff`, porém aqui podemos alterar os arquivos exibidos através das flechas nas bordas (levando ou trazendo as contribuições) ou mesmo editando pela interface. Isso pode ser útil caso necessite desfazer parcialmente um *commit*, ou seja, parte das alterações de uma versão anterior seria mantida e parte alterada.

Contudo, a maior necessidade das ferramentas de comparação não está no seu uso como `difftools`, mas sim como `mergetools`. Já vimos no capítulo 3 que há momentos em que a mesclagem de ramos gera conflitos e estes eram resolvidos abrindo e editando os arquivos conflitantes. Porém com o `meld` ou outras interfaces de comparação, podemos realizar a resolução de conflitos via interface.

Para exemplificar a utilidade na resolução de conflitos na mesclagem, vamos marcar as alterações já feitas no `meu1repo` e criar um novo *branch* alterando os mesmos arquivos a fim de gerar conflito.

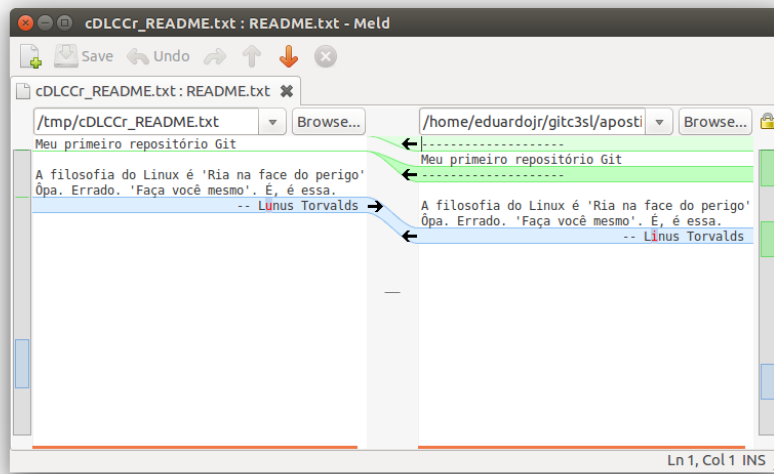


Figura 6.12: Screenshot do programa *meld* utilizado como difftool para o arquivo README.txt RabbitVCS

```
## Criando um novo ramo para desenvolvimento
git branch feature05

## Registrando as alterações no ramo master
git add .
git commit -m "Adiciona TODO e corrige README"

## Indo para o ramo feature05
git checkout feature05

##-----
## Alterando README para induzir conflito

## Destaca título no README
sed -i "2i\#####\" README.txt
sed -i "1i\#####\" README.txt

## Corrige citações, de "'" para ""
sed -i "s/'/\"/g" README.txt

##-----

## Registrando as alterações no ramo feature05
git add .
git commit -m "Adiciona lista de coisas a se fazer"
```

```
[master 7e9c4ab] Adiciona TODO e corrige README
 3 files changed, 11 insertions(+), 1 deletion(-)
 create mode 100644 TODO.txt
Switched to branch 'feature05'
[feature05 dfe7bba] Adiciona lista de coisas a se fazer
 1 file changed, 4 insertions(+), 2 deletions(-)
```

Tentando incorporar as contribuições realizadas acima, do ramo changes para o ramo master obtemos:

```
## Retorna ao ramo principal
git checkout master

## Tentativa de mesclagem
git merge feature05
```

```
Auto-merging README.txt
CONFLICT (content): Merge conflict in README.txt
Automatic merge failed; fix conflicts and then commit the result.
```

E agora, ao invés de editarmos o arquivo em conflito, vamos utilizar a ferramenta meld para resolver os conflitos. Para isso, execute o seguinte comando no terminal:

```
## Lançando a interface `meld` para resolução de conflitos
git mergetool -t meld
```

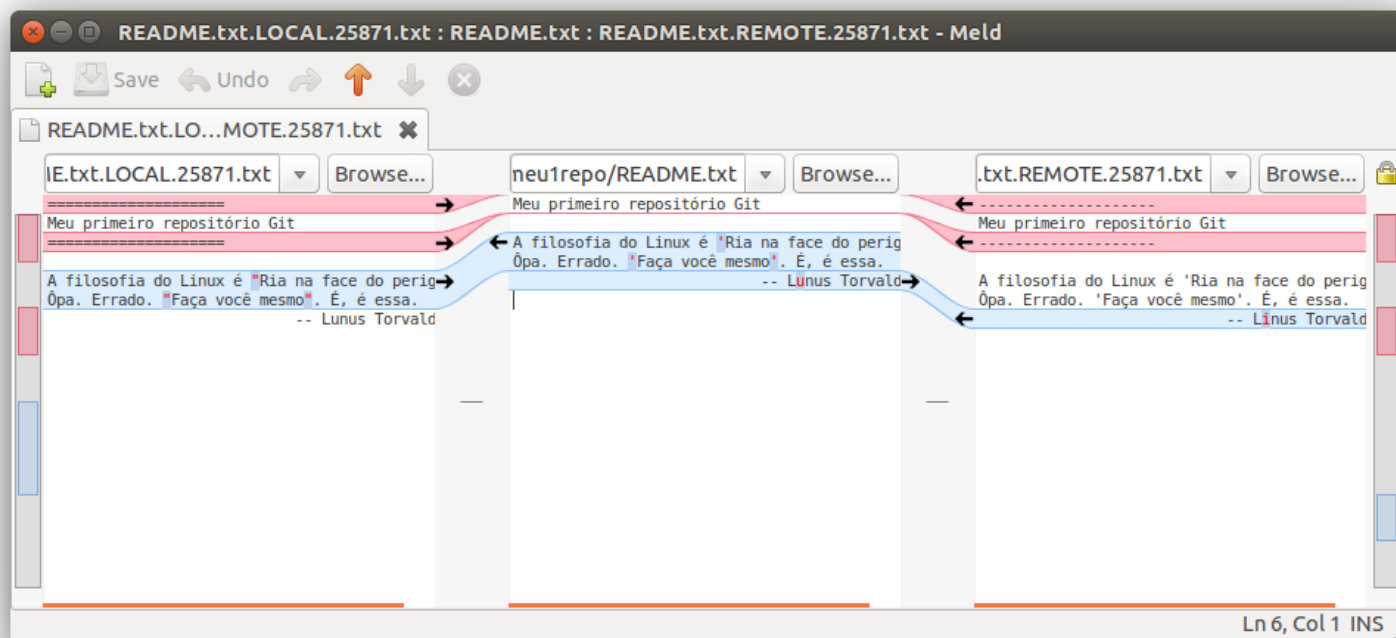


Figura 6.13: Screenshot do programa meld utilizado como difftool para o arquivo README.txt RabbitVCS

Na figura 6.13 temos a janela do programa meld quando usado para resolução de conflito, conforme comando descrito anteriormente. São apresentados três versões lado a lado de cada arquivo em conflito: à direita temos a versão LOCAL, com o estado do arquivo no ramo atual; à esquerda o REMOTE, que representa a versão com as alterações a serem mescladas e; finalmente na porção central temos o BASE, com o conteúdo de uma versão anterior comum a ambos. Assim como apresentado na figura 6.12, em que o meld foi utilizado como difftool, podemos (e neste caso devemos) editar um arquivo o BASE, exibido na porção central do aplicativo. Este arquivo será o definitivo ao fim da mesclagem, nele podemos incluir as contribuições apresentadas no que batizamos de LOCAL e REMOTE. Isso facilita a resolução de conflitos, pois podemos ver as contribuições lado a lado e decidir como deverá ficar o arquivo definitivo.

Após a edição do arquivo, o processo de mesclagem pode continuar normalmente. Abaixo concluímos o processo via linha de comando:

```
## Verificando o estado do repositório
git status
```

```
On branch master
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)
```

```
Changes to be committed:
```

```
    modified:   README.txt
```

```
Untracked files:
```

```
  (use "git add <file>..." to include in what will be committed)
```

```
    README.txt.orig
```

```
## Conclui a mesclagem com a mensagem de commit curta
git commit -m "Resolve conflito via meld"
```

```
[master 112a71b] Resolve conflito via meld
```

Para resolução de conflitos via alguma ferramenta gráfica com o comando `git mergetool`, o **Git** gera arquivos de *backup* com extensão `.orig`. Perceba no *output* gerado pelo `git status` que estes armazenam o conteúdo de cada arquivo em conflito com as porções conflitantes. É recomendável não versionar estes arquivos de *backup*. Podemos então simplesmente excluí-los ou ignorá-los após a mesclagem adicionando arquivos com esta extensão no `.gitignore`. Outra forma de manter seu repositório sem os arquivos *backup* é configurando sua *mergetool* para não armazená-los, ou seja, que a própria ferramenta os descarte quando a mesclagem for bem sucedida. Isso pode ser configurado com:


```
## Configura a ferramenta de merge para não criar os backups  
git config --global mergetool.keepBackup false
```

O procedimento de conflito simulado acima foi resolvido utilizando o programa `meld`, com os comandos definidos para os sistemas baseados no kernel LINUX. Ainda a chamada do programa foi realizada através da opção `-t` (ou `--tool`). Porém podemos definir o `meld` como ferramenta padrão para resolução de merge, assim como foi feito para a ferramenta de comparação (`difftool`). Abaixo configuramos o `meld` também como `mergetool`, para sistemas Unix e Windows.

```
## Configura localmente meld como ferramenta padrão de merge  
## -----  
## Unix  
git config merge.tool meld  
  
## -----  
## Windows  
git config merge.tool meld  
git config merge.meld.cmd '"path/Meld.exe" $LOCAL $BASE $REMOTE --output=$MERGED'
```

Para Windows deve-se informar o caminho para o arquivo executável, `path`, além de definir as três versões que serão exibidas `$LOCAL`, `$BASE` e `$REMOTE` conforme vimos na figura 6.13. Ainda a opção `--output=$MERGED`, para informar que o arquivo a ser editado será sobrescrito sob a versão `$MERGED`, que é criada automaticamente pelo Git quando há conflitos.

Alternativamente pode-se editar o arquivo `.gitconfig` com as mesmas informações passadas ao comando `git config`. Para verificar como altera-se esse arquivo, abra-o após avaliar os comandos descritos. Ainda, se desejado que essas opções sejam válidas para todos os projetos Git de seu computador, a opção `--global` em `git config` pode ser utilizada. Assim, quando avaliados os comandos `git mergetool` e `git difftool`, o programa `meld` será lançado automaticamente.

Com isso, já temos nosso **Git** devidamente configurado para utilizar o programa `meld` e já observamos sua relevância quando se trabalha com arquivos versionados. Mas ainda, apresentamos somente uma das várias interfaces que se dispõem a facilitar a visualização de diferenças e mesclagem de arquivos. Podemos citar as interfaces `kdifff3`² e `P4Merge`³, como outras interfaces de comparação bastante utilizadas em projetos versionados. Em geral, todos estes programas seguem o mesmo estilo de exibição de arquivos que o `meld` e as configurações para torná-los programas de `mergetool` e `difftool` padrão são as mesmas.

É importante salientar que as ferramentas gráficas apresentadas neste capítulo não substituem totalmente os comandos via terminal, mas seu uso em

²Disponível para download em <http://kdifff3.sourceforge.net/>

³Disponível para download em <https://www.perforce.com/product/components/perforce-visual-merge-and-diff-tools>

conjunto com estes facilitam o fluxo de trabalho adotado em um projeto sob versionamento.

Capítulo 7

Trabalhando em equipe

O Git é uma ferramenta que aliada a outros serviços web, como GitLab ou GitHub, oferece funcionalidade e autonomia para se trabalhar. Contudo, com tantos recursos disponíveis, só serão bem aplicados quando todos os membros do grupo, além de conhecê-los, trabalham em harmonia.

7.1 Boas práticas de colaboração

Repositório é onde são armazenados os arquivos de um projeto. Existem três níveis de acesso permitidos:

- **Private:** é o repositório fechado, onde apenas o criador (Owner) tem permissão de leitura e escrita. Se um repositório privado for criado dentro de um grupo, todos do grupo terão permissão de leitura e escrita.
- **Internal** repositório fechado para usuários externos ao grupo, mas qualquer usuário cadastrado no grupo terá permissão de leitura e escrita no repositório.
- **Public:** repositório aberto, visível para qualquer pessoa (usuário do grupo ou não). Usuários do grupo tem permissão de leitura e escrita no repositório. Usuários sem conta no grupo podem clonar o repositório, mas não tem permissão para alterar o repositório (enviar merge requests por exemplo).

É possível adicionar usuários para colaborar em um repositório. Cada usuário pode ter um nível de acesso diferente: **Guest, Reporter, Developer, Master**. Em permissões ¹ é possível visualizar as habilidades concedidas para cada nível.

Logo após criar um novo repositório, é recomendável que se crie um arquivo README.md. Independente da forma como o repositório foi configurado, é sempre fundamental que ele contenha o arquivo README.md. Este arquivo é

¹<https://gitlab.c3sl.ufpr.br/help/permissions/permissions>

sempre o primeiro a ser mostrado na página inicial de todo repositório. Por esse motivo, é importante que o `README.md` contenha no mínimo:

- Uma descrição geral do projeto;
- Os nomes dos autores do projeto;
- Instruções de instalação, no caso de softwares;
- A licença do projeto (especialmente para projetos públicos), ou uma orientação sobre o uso do projeto (permissão, citação, entre outros). Opcionalmente pode-se criar um arquivo `LICENSE` com a licença. Esse arquivo ficará disponível também em uma aba na página inicial do projeto.
- **(Opcional):** um guia de contribuição, se o (Owner) do projeto pretende que usuários externos colaborem, é possível apresentar algumas orientações básicas sobre como colaborar. Criando um arquivo `CONTRIBUTING.md` com este guia, ele será automaticamente colocado em uma aba na página inicial do projeto.
- **(Opcional):** um *changelog* para que sejam registradas as modificações realizadas entre uma versão e outra (principalmente para softwares). Criando esse arquivo com estas informações, ele aparecerá automaticamente em uma aba na página inicial do projeto.

Outra parte fundamental do Git, são os **commits**. Além de salvarem as alterações realizadas nos arquivos, também são responsáveis por documentar as alterações feitas por qualquer usuário e em qualquer arquivo. Os commits agilizam o processo de revisão do projeto, e poderá ajudar futuros mantenedores do projeto a desvendar o motivo de algum acréscimo ou modificação no código. Por causa dessas importâncias, uma mensagem bem escrita é a melhor forma de se comunicar a alteração para os demais membros do grupo e para você mesmo. Essas mensagens também aparecerão no `git log` do projeto, por isso é essencial que sejam bem escritas, de forma clara e sigam um padrão.

Algumas **regras de ouro**, que são convenções gerais, para que um projeto versionado com Git seja bem sucedido são:

- **Faça commits regularmente:** isso faz com que as mudanças de código entre um commit e outro sejam menores, tornando mais fácil para todos acompanhar as alterações;
- **Não faça commits de trabalhos pela metade:** faça um commit apenas quando tiver finalizado o que estava propondo. Isso irá forçar você a deixar o trabalho em pedaços menores, e por consequência realizar commits regularmente;
- **Teste antes de fazer um commit:** resista à tentação de fazer um commit que você pensa que está completo. Teste toda a sua realização para ter certeza de que não causará um efeito colateral no projeto;
- **Escreva boas mensagens de commit:** seja claro e objetivo ao escrever as mensagens de commit. No entanto, tome cuidado para não ser vago, ou escrever apenas mudança, mais mudanças, etc. Se uma mensagem curta for suficiente, use `git commit -m 'Mensagem'`, mas lembre-se de ser informativo sobre a alteração realizada, para ser útil para todos do projeto.

Existem outras convenções estabelecidas sobre como escrever mensagens de commit contextualizadas, baseadas nas mensagens geradas por mensagens de funções do próprio Git. Estas convenções podem resumidas nas 7 regras que são convenções globais:

1. **Separe o título do corpo do texto com uma linha em branco:** por padrão, a primeira linha é o título do commit, e deve ser uma mensagem curta. Ao deixar uma linha em branco, é permitido escrever uma mensagem de qualquer tamanho, detalhando melhor as modificações feitas. Dessa forma, quando `git log` for executado, toda a mensagem de commit aparecerá, enquanto que `git log --oneline` mostrará apenas o título do commit.
2. **Limite a linha de título em 50 caracteres:** isso faz com que o colaborador pense mais para escrever uma mensagem mais informativa. Se a mensagem for uma única linha (`git commit -m`), então esse limite pode se estender para 72 caracteres.
3. **Capitalize a mensagem:** em todas as mensagens de commit comece com letra maiúscula, tanto se for título, corpo da mensagem, ou apenas uma mensagem de uma única linha.
4. **Não termine os commits com ponto:** principalmente se for o título de uma mensagem de commit mais longa. Espaço é valioso quando dispomos apenas de 50 ou 72 caracteres.
5. **Use o modo imperativo:** no título de commits longos ou em mensagens de commits únicas. O modo imperativo significa escrever como se estivesse dando um comando a alguém. Seja direto e objetivo, e escreva no presente. Exemplos de mensagens no imperativo:

- Adiciona versão final
- Altera parágrafo da introdução
- Remove funções precipitadas

Algumas mensagens no modo **não** imperativo são:

- Corrigindo o erro
- Mudando a função
- Mais correções para mais funções

6. **Limite o corpo da mensagem em 72 caracteres:** ao escrever uma mensagem de commit mais longa, devemos manter o corpo da mensagem com no máximo 72 caracteres.
7. **Use o corpo da mensagem para explicar “o que” e “porque”, e não “como”:** contextualize o que você fez e o motivo. Na maioria dos casos você pode deixar de fora como você fez as modificações, pois o código alterado será auto-explicativo.

7.2 Modelos de fluxos de trabalho

A escolha do *workflow* (fluxo de trabalho) depende de cada projeto e das preferências pessoais. Podemos utilizar as informações sobre cada *workflow*, e decidir qual é mais adequado para cada projeto. Existem quatro maneiras principais de trabalhar em colaboração com o Git e o GitLab:

7.2.1 Centralized workflow

Recomendado para projetos pequenos, e/ou que não necessitam de muitas alterações. Nesse workflow, o repositório possui apenas um branch (master) e as alterações são feitas nesse branch. As revisões só poderão ser realizadas depois que tudo foi enviado para o servidor remoto. Com isso, há uma grande chance de ocorrerem conflitos.

Exemplo

Após iniciar um repositório central, os colaboradores devem clonar o repositório.

Depois de um colaborador terminar seu trabalho remotamente, ele publica as modificações para o repositório central, para que os demais membros possam ter acesso.

Outro membro também termina seu trabalho e resolve publicar no repositório central, porém não conseguirá. O repositório central está diferente do seu repositório local.

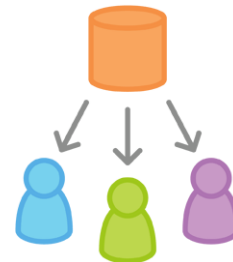


Figura 7.1: Colaboradores clonando o repositório central.

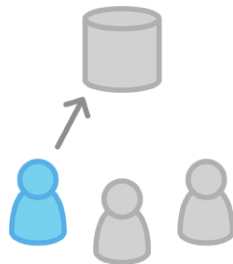


Figura 7.2: Colaborador publicando as modificações no repositório central.

Para conseguir enviar as modificações realizadas, o colaborador precisa puxar as atualizações feitas para o seu repositório, integrá-las com as suas alterações locais, e em seguida, tentar novamente.

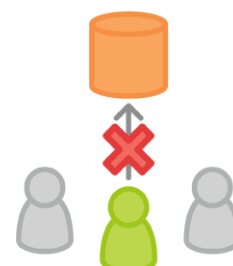


Figura 7.3: Colaborador não conseguindo publicar as modificações no repositório central.

Após feito isso, será possível o colaborador fazer as modificações.

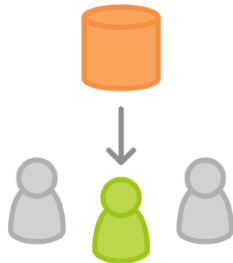


Figura 7.4: Colaborador publicando as modificações do repositório central.

7.2.2 Feature branch workflow

Recomendado para projetos pequenos e grandes, que envolvam mais de um colaborador. O projeto principal é mantido no branch master. Se um membro quiser realizar alguma alteração, deverá criar um novo branch feature, e fazer as alterações nesse branch e sugerir um merge request. Com isso, os demais colaboradores poderão revisar as alterações sugeridas e discutir as modificações, até que uma pessoa habilitada faça o merge desse branch feature para o branch master.

Dessa forma, evita-se os possíveis conflitos e garante que tais alterações não causaram algum problema. Esse workflow é altamente recomendado por ser simples de gerenciar, evitar grandes conflitos, e ser relativamente fácil para usuários novos do Git.

Exemplo

Antes de começar a desenvolver um recurso, é preciso criar um ramo isolado para trabalhar.

Com isso, o membro poderá iniciar o seu trabalho, e realizar o que for necessário nesse ramo (branch). O colaborador, após finalizar o projeto, irá requerir um merge request para que as alterações feitas nesse branch, sejam incorporadas no master. Os demais

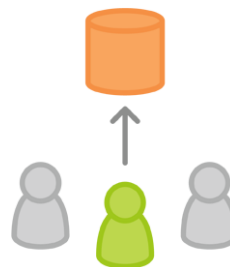


Figura 7.5: Colaborador publica modificações do repositório central.



Figura 7.6: Criando um novo ramo para o trabalho.

membros, poderão avaliar se tais modificações são pertinentes para o projeto.

Quando as alterações sugeridas para o colaborador forem incorporadas o branch poderá ser movido para o master.

7.2.3 Gitflow workflow

Indicado para projetos maiores e/ou com um grande número de colaboradores. Esse workflow envolve a criação de alguns branches com funções específicas. Todo o desenvolvimento é realizado no branch develop. Quando uma versão está pronta, ela é movida para o branch release, onde é testada e finalmente incorporada ao ramo master, que contém apenas versões finais (estáveis). É extremamente recomendado esse workflow para desenvolvimento de softwares, porém exige de mais familiaridade com o Git. Permite, por exemplo, que os usuários de um software, instalem tanto uma versão estável (do branch master) quanto uma versão em desenvolvimento (do branch develop).

Exemplo

São criados branches com funções específicas, como no exemplo Hotfix, Release e Develop.

Develop é semelhante ao master do feature branch workflow. *Release* serve para “lançar” possíveis bugs gerados no código. *Hotfix* contém as correções dos bugs do release que não podem aguardar o lançamento do mesmo.

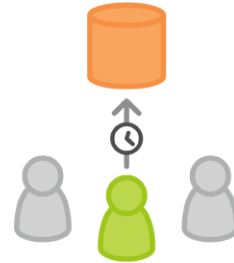


Figura 7.7: Colaborador solicita merge, e aguarda revisão dos colaboradores.

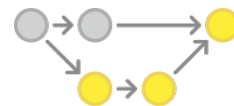


Figura 7.8: Movendo ramo para o master.

7.2.4 Forking workflow

Recomendado para projetos abertos, onde se espera que usuários externos façam contribuições. Esse workflow consiste em um repositório oficial, de onde os colaboradores fazem um fork desse repositório, e passam a desenvolver o projeto de maneira independente. Assim, cada colaborador poderá adotar o workflow de preferência, e não precisará ter acesso ao repositório oficial, apenas colaborar enviando merge.

Exemplo



Figura 7.9: Ilustração dos branches específicos.

Criado o repositório central, os colaboradores fazem um fork e poderão trabalhar de maneira independente.

Independente da escolha do workflow para cada projeto é importante sempre informar o método que está sendo utilizado para seus colaboradores, para que eles possam seguir o mesmo padrão.

Essas informações poderão ser descritas em `README.md` ou no `CONTRIBUTING.md`.

7.3 Fluxo de trabalho PET no GitLab

O PET-Estatística UFPR possui um grupo no Git para o desenvolvimento de projetos. Utilizaremos a seguinte ilustração para entender o fluxo do trabalho do PET.

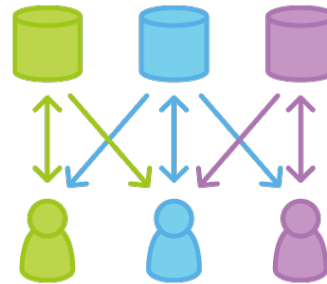


Figura 7.10: Ilustração dos forks de um projeto.

Conforme a demanda de projetos, é criado o repositório para armazená-lo. Após isso, são criados as milestones - marcadores de classificação dos arquivos. Esses passos são feitos no Owner.

Indo para o master, temos os seguintes passos:

- Conforme a demanda do projeto, é criado um issue para adição de contribuições;
- Atualiza o ramo devel;
- Após isso, é necessário criar um branch (ramo) para incluir as contribuições;

Entrando no developer, teremos o ciclo de trabalho em que adiciona as modificações (git add), registra as mesmas (git commit) e após realizar todo o trabalho, é feito o git push enviando ao servidor remoto.

A próxima etapa é a requisição de merge. Com esse merge, é feita as discussões a respeito da contribuição, assim podendo retornar ao ciclo do developer para as devidas correções e sugestões. Após a certeza dessa contribuição, é movida para o ramo devel e fechado o issue referente ao trabalho feito.

Depois de terminar todas etapas do projeto, completa-se as milestones, realiza o merge do devel no master, e cria a tag de versão.

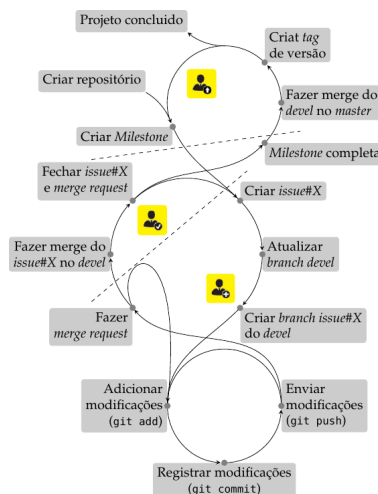


Figura 7.11: Ilustração do fluxo de trabalho do PET.

Apêndice A

Exemplos de rotinas

Neste apêndice são descritas brevemente algumas das rotinas mais usuais em projetos Git. A maioria dos exemplos estão detalhados na apostila e foram resumidos nesta coletânea. Abaixo temos a lista de rotinas aqui presentes.

Rotinas

Rotina 1:	Configure usuário e e-mail.	123
Rotina 2:	Inicie um projeto Git local.	124
Rotina 3:	Trabalhe com ramos.	124
Rotina 4:	Visualize diferenças.	125
Rotina 5:	Resolva conflitos de merge.	126
Rotina 6:	Visualize seu projeto.	126
Rotina 7:	Volte versões anteriores.	126
Rotina 8:	Reescreva commits.	127
Rotina 9:	Cria chaves públicas.	127
Rotina 10:	Trabalhe remotamente.	128
Rotina 11:	Manipule ramos remotos.	128
Rotina 12:	Adicione endereço remoto.	130

Rotina 1: Configure usuário e e-mail.

```
## Configurando localmente
## - válido para o repositório atual
git config user.name "Name Lastname"
git config user.email "namelastname@servidor"
```

```
## Configurando globalmente
## - válido para todos os repositórios do computador
git config --global user.name "Name Lastname"
git config --global user.email "namelastname@servidor"

## Obs.: As configurações locais se sobrepõem as
## globais, quando as duas forem avaliadas.
```

Rotina 2: Inicie um projeto Git local.

```
## Em um diretório que deseja-se versionar

## Inicia o versionamento Git
git init

## Verifica o estado do repositório
git status

## Adicione os arquivos para receber o commit
git add file1.txt file2.R file3.txt file4.Rmd

## Registre a versão do repositório com uma mensagem
## informativa
git commit -m "Inicia repositório"

## Verifica o histórico de versões
git log
```

Rotina 3: Trabalhe com ramos.

```
## Verifica os ramos existentes
git branch

## Cria um ramo, para trabalho específico como:
## - Corrigir bugs
## - Desenvolver features
## - Tarefas em geral
git branch bugfix

## Altera versionamento para o ramo criado
git checkout bugfix

## Adiciona as alterações que corrigem o bug
git add app.R
```

```
## Registra as alterações
git commit -m "Altera delimitadores da função"

## Retorna para o ramo principal
git checkout master

## Incorpora ao ramo principal as alterações
## realizados no ramo bugfix
git merge bugfix

## Deleta o ramo responsável pela correção do bug
git branch -d bugfix
```

Rotina 4: Visualize diferenças.

```
## -----
## Diferenças não commitadas

## Lista as diferenças entre o último commit e o
## estado do repositório no working directory
git diff

## Diferença para um único arquivo
git diff file.Rmd

## -----
## Diferenças entre versões commitadas

## Verifica os registros com referência
git reflog

## Lista as diferenças entre o último e o
## antepenúltimo registro
git diff HEAD~0 HEAD~2 ## ou pelo SHA1 ID
git diff 7204daa bela9cc

## -----
## Diferenças entre ramos

## Lista diferenças entre os ramos master e
## feature1
git diff master feature1

## Obs.: Programas externos podem ser utilizados para
## visualizar diferenças, use difftool no lugar de
```

```
## diff, ver cap. 06
```

Rotina 5: Resolva conflitos de merge.

```
## Incorpora modificações realizadas no ramo feature
git merge feature

## Edite e salve o(s) arquivo(s) em conflito, as
## porções em conflito são destacadas por <<< === >>>

## Finaliza merge com o commit:
git commit -a -m "Merge do ramo feature"
```

Rotina 6: Visualize seu projeto.

```
## Histórico de registros
git log

## Histórico de registros em uma linha com trajetória
## dos ramos
git log --graph --oneline

## Histórico de registro com referência
git reflog

## Histórico de registro via interface gráfica padrão
gitk
```

Rotina 7: Volte versões anteriores.

```
## Verifica o histórico de versões do repositório
## - Guarde o SHA1 ID do registro desejado: ec3650c8
git log --oneline

## -----
## Descartar todas as alterações até um commit
git reset --hard ec3650c8

## Obs.: Reescreve a história do repositório, não
```

```
## é recomendável reescrever a linha do tempo quando
## se está em um projeto colaborativo remoto.

## -----
## Reverte o estado atual para um registro específico
git revert ec3650c8
git commit -am "Retorna projeto para versão funcional"

## Obs.: Faz um merge da versão atual com a versão do
## SHA1 ID informado. Podem haver conflitos que devem
## ser resolvidos para concluir a reversão.

## -----
## Cria um ramo provisório a partir de um SHA1 ID
git checkout ec3650c8

## Visualiza os ramos existentes
git branch

## Cria um ramo definitivo com o estado no SHA1 ID
git checkout -b ramo_teste

## Obs.: O ramo provisório é removido assim que se
## fizer um checkout para qualquer outro ramo
```

Rotina 8: Reescreva commits.

```
## Verifica o histórico de versões do repositório
git log --oneline

## -----
## Reescreve a última mensagem de commit
git commit --amend -m "Correção de Commit"

## Obs1.: Arquivos na staging area também são
## incorporados ao último commit
## Obs2.: Reescreve a história do repositório, não
## é recomendável reescrever a linha do tempo quando
## se está em um projeto colaborativo remoto.
```

Rotina 9: Cria chaves públicas.

```
## Cria uma chave pública.  
ssh-keygen -t rsa -C "namelastname@servidor"  
  
## Exibe as chaves públicas.  
cat ~/.ssh/id_rsa.pub  
  
## Adicione o conteúdo a um servidor remoto, como:  
## - git@github.com  
## - git@gitlab.com  
## - git@gitlab.c3sl.ufpr.br  
  
## Verifica conexão com o servidor  
ssh -T endereço ## endereço = git@github.com, ...  
  
## Obs.: Todos os comandos ssh são provenientes do  
## pacote de função ssh para shell, portanto para  
## utilizar instale este pacote.
```

Rotina 10: Trabalhe remotamente.

```
## Clona um projeto remoto:  
## e.g. git@github.com:pet-estatistica/apostila-git.git  
git clone endereço:namespace/project.git  
  
## Realiza modificações e/ou inclusões de um ou  
## vários arquivos  
  
## Adiciona todas as alterações para commit  
git add .  
  
## Registra suas alterações  
git commit -a -m "Modifica compilação do projeto"  
  
## Envia as alterações para o repositório remoto (origin)  
git push origin  
  
## Traz estado do repositório remoto  
git pull origin
```

Rotina 11: Manipule ramos remotos.

```
## -----  
## Realizando contribuições em um ramo remoto
```



```
## Lista todos os ramos, locais e remotos
git branch -a

## Altera versionamento para o ramo issue01
git checkout issue01

## Traz estado do ramo remoto issue01
git pull origin issue01

## Realiza modificações e/ou inclusões de um ou
## vários arquivos

## Adiciona todas as alterações para commit
git add .

## Registra suas alterações
git commit -a -m "Modifica laço iterativo condicional"

## Envia as alterações no ramo para a versão remota
git push origin issue01

## -----
## Realizando contribuições em um ramo remoto e
## enviando o ramo principal mesclado

## Traz estado do ramo remoto issue01
git pull origin bugfix

## Altera versionamento para o ramo issue01
git checkout bugfix

## Realiza modificações e/ou inclusões de em um ou
## vários arquivos

## Adiciona todas as alterações para commit
git add .

## Registra suas alterações
git commit -a -m "Altera classe do objeto retornado"

## Retorna ao ramo principal
git checkout master

## Incorpora modificações realizadas no ramo bugfix
git merge bugfix

## Envia as alterações ao repositório remoto
git push origin master
```

```
## Deleta o ramo bugfix
git branch -d bugfix ## Local
git push :bugfix      ## Remoto
```

Rotina 12: Adicione endereço remoto.

```
## Lista os servidores remotos, com endereço
git remote -v

## -----
## Adicionando local para trazer contribuições

## Adiciona local remoto com nome gitlab:
git remote add gitlab git@gitlab.com:user/project.git

## Adiciona local remoto com nome github:
git remote add github git@github.com:user/project.git

## Atualiza arquivos locais, baseado no local remoto:
git pull gitlab ## gitlab.com:namespace/project.git
git pull github ## github.com:namespace/project.git

## -----
## Adicionando local para enviar contribuições

## Lista os locais de origem
git remote show origin

## Adiciona novo local de origem:
## e.g. git@github.com:pet-estatistica/apostila-git.git
git remote set-url origin --push --add endereço_remoto

## Envia as contribuições para os locais remotos
git push origin
```

Apêndice B

Dicionário de termos

B.0.1 Config

O `config` é um comando usado para ajustar as configurações padrão do Git. Há duas configurações básicas a serem feitas: a inclusão do e-mail e do nome do usuário Git. Todas as configurações definidas como globais ficam armazenadas em um arquivo chamado `.gitconfig`, que fica localizado no diretório padrão do usuário.

Exemplo:

```
# Configurando o usuário Ezio Auditore:  
git config --global user.name "Ezio Auditore"  
# Configurando o e-mail:  
git config --global user.email ezio.auditore@exemple.com
```

B.0.2 SSH Key

É uma chave de autenticação baseada em criptografia de chave pública (chave assimétrica). Essa criptografia torna o processo de transferência de arquivos entre o cliente e o servidor mais segura.

Para que o Git local consiga se conectar a um servidor Git remoto, é necessário que esta chave seja gerada e que as devidas configurações sejam feitas tanto localmente quanto no servidor remoto, caso contrário, é exibido um erro e a conexão é interrompida.

Exemplo:

```
# Gerando chave SSH:  
ssh-keygen
```

B.0.3 Help

Todo comando Git tem um manual de ajuda que pode ser exibido na tela com o comando `--help`.

Exemplo:

```
# Exibir ajuda do comando status:  
git status --help
```

B.0.4 Repositório

Um repositório é uma pasta gerenciada pelo Git. A partir da criação desta, usufruímos do sistema de versionamento, sendo possível transitar entre as diferentes versões a medida que necessário.

Exemplo:

```
# Iniciar repositório na pasta atual:  
git init
```

B.0.5 Stagin Area

A stagin area é um espaço temporário na pasta gerenciada pelo Git. É o local em que ficam os arquivos antes de serem marcados como uma versão definitiva. Em tradução livre, stagin area é área de estágio, podemos assim imaginar que o arquivo está estagiando antes de ser promovido a um arquivo definitivo.

B.0.6 Remote

O remote mostra o servidor remoto onde os arquivos Git estão hospedados. O remote padrão geralmente é criado com o nome de `origin`, mas é possível adicioná-lo utilizando outros nomes e até mesmo adicionar outros servidores remotos juntamente ao `origin`.

Exemplo:

```
# Adicionando um servidor remoto com nome origin:  
git remote add origin "git@gitlab.c3sl.ufpr.br:pet-estatistica/apostila-git.git"
```

B.0.7 Clone

O clone é usado quando deseja-se clonar um repositório que está disponível em um servidor remoto para o servidor local. Depois da clonagem, estará disponível todos os arquivos e todo o histórico de controle de versões sem a necessidade de uso da internet.

É importante salientar que quando se é usado o clone, o servidor remoto é adicionado automaticamente, podendo ser acessado através do comando origin.

Exemplo:

```
# Clonando o projeto desta apostila:
git clone git@gitlab.c3sl.ufpr.br:pet-estatistica/apostila-git.git
# Exibindo os servidores remotos:
git remote
```

B.0.8 Status

O status exibe a diferença entre o estado atual dos arquivos e o estado do último commit do mesmo branch. São três estados possíveis: consolidado (committed), modificado (modified) e preparado (staged).

Exemplo:

```
# Pedindo o status:
git status
```

B.0.9 Add

O add adiciona (envia) os arquivos para a stagin area, para que possa ser marcado no tempo por um commit.

Exemplo:

```
# Adicionar todos os arquivos a stagin area:
git add *
```

B.0.10 Commit

O commit marca os arquivos da stagin area como uma versão definitiva, para que posteriormente, caso algum erro ocorra, possamos voltar nos commits anteriores onde o código está em pleno funcionamento.

Exemplo:

```
git commit -m "Meu primeiro commit"
```

B.0.11 Branch

Os branches são como uma cópia dos arquivos do ultimo commit para um ambiente de desenvolvimento paralelo, o que permite que as modificações em

um branch não afete os arquivos em outro. Os branches também são chamados de ramos de desenvolvimento. Veja com mais detalhes nos capítulos 3 e 7.

Exemplo:

```
# Cria um branch chamado novoBranch  
git branch novoBranch
```

B.0.12 Checkout

O checkout serve para transitar entre branches e commits. Usando o checkout é possível voltar a commits anteriores.

Exemplo:

```
# Mudar do branch atual para o branch teste:  
git checkout teste
```

B.0.13 Merge

Com o merge é possível a fusão de dois ramos em um.

Quando se trabalha em ramos diferentes (diferentes branches) e precisa-se posteriormente juntar o trabalho, o merge (fundir) é usado, permitindo que o trabalho seja centralizado novamente. A fusão é feita de forma automática, mas conflitos podem ocorrer, como por exemplo, quando duas ou mais pessoas modificam a mesma parte do código. Estes conflitos devem ser resolvidos manualmente, deixando a cargo do gerente de projetos decidir que parte do código deve permanecer.

Exemplo:

```
# Faz merge do branch chamado novoBranch com o branch atual:  
git merge novoBranch
```

B.0.14 Rm

O `git rm`, na sua forma mais comum, serve para remover um arquivo de forma que ele deixe de ser gerenciado pelo Git e seja excluído do disco rígido. Também é possível que o arquivo deixe de ser gerenciado e permaneça no disco.

Uma das vantagens é que, quando o arquivo é removido pelo `git rm`, já aparece como preparado (staged), precisando somente que a exclusão seja commitada.

Exemplo:

```
# Remover arquivo teste.tex do gerenciamento e do disco:  
git rm teste.tex  
  
# Remover arquivo teste.tex apenas do gerenciamento:  
git rm --cached teste.tex
```

B.0.15 Mv

O `git mv` move ou renomeia arquivos informando ao Git.

Caso a mudança seja feita sem esse comando, o Git entende que o arquivo foi deletado e que um novo arquivo foi criado, deixando de fora, por exemplo, a ligação existente entre o arquivo e seus commits.

Exemplo:

```
# Renomeando o arquivo teste.tex para arquivo1.tex:  
git mv teste.tex arquivo1.tex
```

B.0.16 Push

O `push` é usado para “empurrar” os arquivos do repositório local para o servidor remoto.

Exemplo:

```
# Atualizado o branch master (remoto), com o branch atual (local):  
git push origin master
```

B.0.17 Fetch

O `fetch` atualiza o repositório local com as alterações do remoto, porém não realiza o merge dos arquivos, deixando isso para ser feito manualmente.

Exemplo:

```
# Buscando arquivos no servidor remoto origin:  
git fetch origin
```

B.0.18 Pull

O `pull` é semelhante ao comando `fetch`, porém, puxa os arquivos do servidor remoto fazendo merge. Caso haja algum conflito de merge, estes deverão ser resolvidos manualmente.

Exemplo:

```
# Puxando arquivos no servidor remoto origin:  
git pull origin
```

B.0.19 HEAD

HEAD é um arquivo que contém um apontador para o branch atual. Quando o checkout é executado para a mudança do branch, esse arquivo é automaticamente modificado, apontando agora para o novo local, e assim permitindo que, quando o computador for desligado e depois de reiniciado o Git ainda esteja trabalhando com o mesmo branch.

B.0.20 Tag

As tags são usadas para marcar pontos específicos do desenvolvimento. Geralmente são usadas para marcar versões definitivas, como a v1.0, v2.0 e assim por diante.

Elas são divididas em dois tipos: leve e anotada. A tag leve simplesmente aponta para um commit específico. Já a tag anotada é guardada como objetos inteiros, possuindo algumas informações, como o nome da pessoa que criou, a data, uma mensagem semelhante a de commit, entre outras.

Exemplo:

```
# Criando tag leve:  
git tag -l "v1.0.0"  
  
# Criando tag anotada:  
git tag -a v1.0 -m "Minha primeira tag anotada."
```

B.0.21 Stash

Com este comando não é necessário fazer um commit para mudar de branch. Ao executá-lo, os arquivos modificados ficam salvos em uma pilha de modificações inacabadas, sendo possível transitar entre branches e voltar ao trabalho inacabado quando necessário.

Exemplo:

```
# Fazendo o stash:  
git stash  
# Listando os stash criados:  
git stash list
```


B.0.22 Reset

Enquanto o `git checkout` somente transita entre os commits, o `reset` pode também alterar o histórico, fazendo commits serem apagados de maneira irreversível (`--hard`) ou serem apenas retornados ao estado de não commitado (`--soft`).

Exemplo:

```
# Apagando o último commit (voltando ao anterior):  
git reset --hard HEAD~1
```

B.0.23 Rebase

O `rebase` é usado para modificar commits antigos. Ele refaz a árvore de commits, sendo assim não é uma boa ideia fazer um `push` da alteração, pois modificará a árvore do servidor afetando todos os desenvolvedores.

A ideia geral é `rebase` apaga os commits de um ramo e “muda de base”, passando-os para novos commits do ramo atual, formando uma árvore com fluxo de trabalho linear.

Exemplo:

```
# Fazer o rebase do branch teste para o atual:  
git rebase teste
```

B.0.24 Blame

Pesquisa qual commit modificou determinado arquivo, com o objetivo de encontrar quem e quando um erro foi inserido. Esse método também é chamado de anotação de arquivo.

Exemplo:

```
# Fazer
```

B.0.25 Bisect

O `bisect` realiza uma pesquisa binária (binary search) a procura de erros. Para que a pesquisa ocorra, é necessário um ponto no tempo em que o código esteja funcionando e outro que não esteja.

Exemplo:

```
# Pesquisa Binária:  
# Iniciando a pesquisa.  
git bisect start  
# Marcando o commit atual como não funcionando.  
git bisect bad  
# Marcando o commit com nome commit1 como funcionando:  
git bisect good commit1
```