

# Ferramentas Gráficas

*PET Estatística UFPR*



# Sumário

<b>1</b>	<b>Ferramentas gráficas</b>	<b>5</b>
1.1	Interfaces Git . . . . .	7
1.1.1	git-gui . . . . .	7
1.1.2	gitk . . . . .	11
1.1.3	Outras Interfaces . . . . .	15
1.2	Interfaces de comparação . . . . .	22



# Capítulo 1

## 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`, `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; e o procedimento de *commit* se torna trabalhoso, pois é necessário listar todos os arquivos que fazem parte de um *commit* no commando `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 as 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.

## 1.1 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.

### 1.1.1 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**

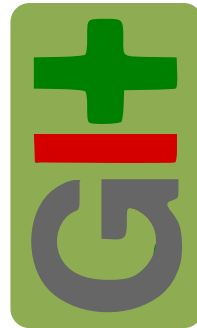


Figura 1.1: Logo usada para a `git-gui`, também é uma das logos do próprio GIT

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 as 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
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

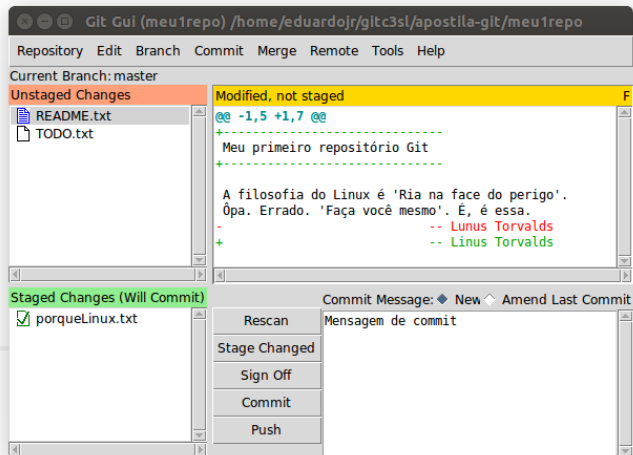


Figura 1.2: *Screenshot* da execução do programa git-gui

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 1.2 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 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 1.2 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.

### 1.1.2 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 GLit, 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.

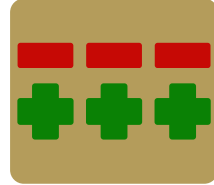


Figura 1.3: Logo da interface gitk, símbolos de supressão e adição são característicos das logos GIT

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 1.4 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

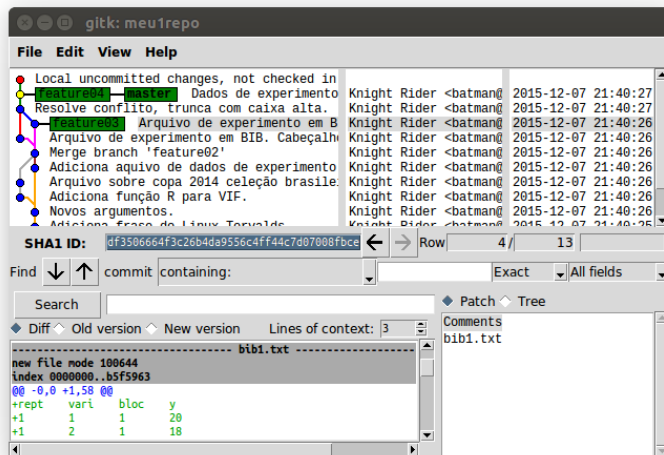


Figura 1.4: Screenshot da execução do programa gitk

ramo a partir do *commit* dentre outras opções possíveis através da interface.

### 1.1.3 Outras Interfaces

#### 1.1.3.1 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*.

#### 1.1.3.2 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.

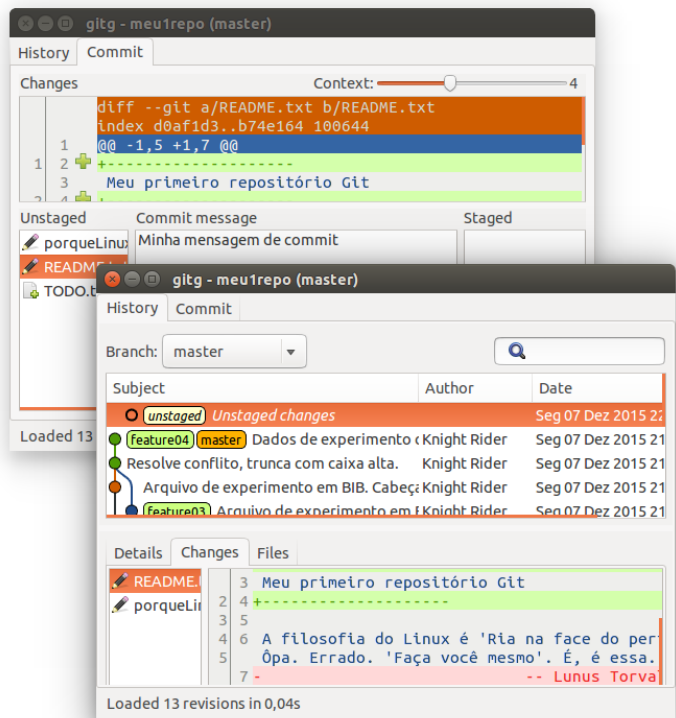


Figura 1.5: Screenshot da execução do programa gitg



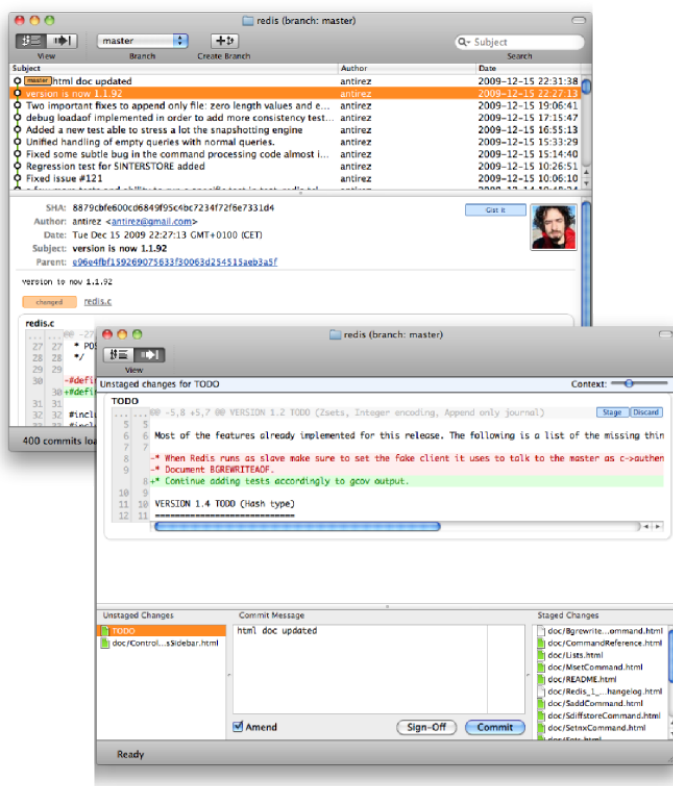


Figura 1.6: Screenshot do programa gitx

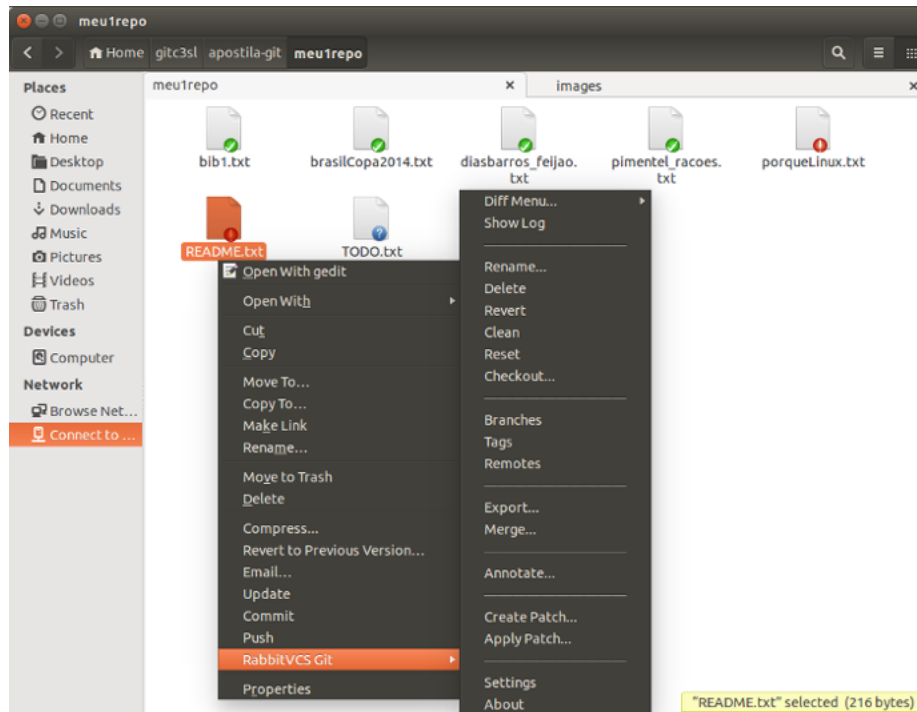


Figura 1.7: Screenshot do Navegador Nautilus com uso do RabbitVCS

Na figura 1.7 temos o *screenshot* do repositório `meu1repo` 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.

### 1.1.3.3 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 1.8 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 destaca em implementação de funcionalidade Git, a `git-cola` se

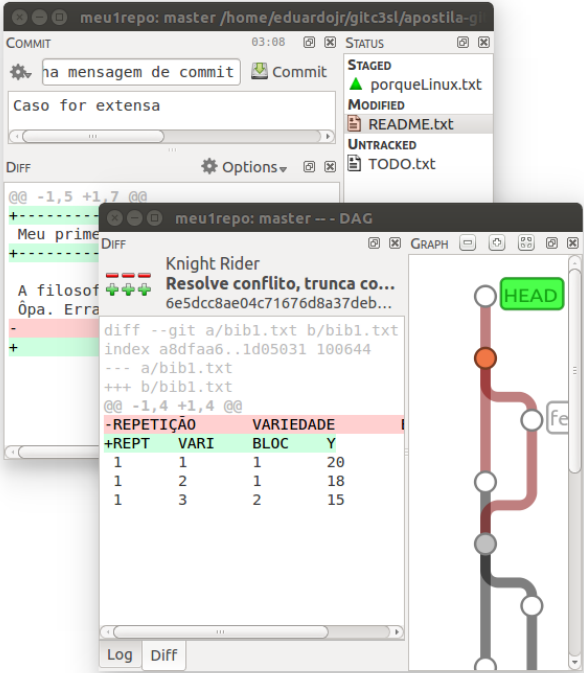


Figura 1.8: Screenshot dos programas git-cola e git-dag

sobressai com relação à `git gui` na possibilidade de execução do comando `git rebase` via interface.

### 1.1.3.4 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** integrado 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*<sup>1</sup> 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

---

<sup>1</sup>Informações em <http://magit.vc/>

onde as diversas ferramentas serão exploradas, com exemplos e ilustrações voltadas para a comunidade estatística.

## 1.2 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,



Figura 1.9: Logo do aplicativo **meld**, não há descrições sobre o seu significado, mas nos parece representar

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 1.1, porém é permitido e será usado para comparar versões de arquivos ou repositórios, mas vamos começar apresentando o `meld` como programa independente.

Inicializando o programa, sua tela inicial deverá ser similar a apresentada na figura 1.10, 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 1.11 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

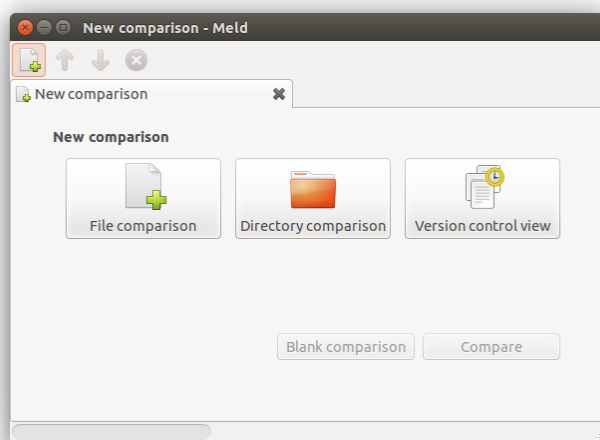


Figura 1.10: *Screenshot* do tela inicial do programa *meld* RabbitVCS



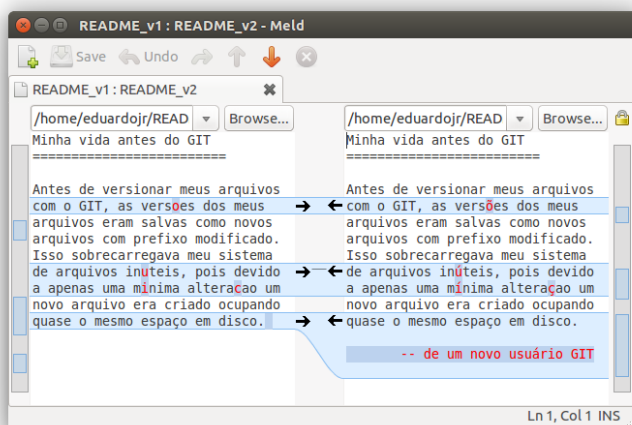


Figura 1.11: *Screenshot* de comparação de arquivos com programa *meld* RabbitVCS

fluxo de trabalho de um projeto sob versionamento. Vamos então voltar ao nosso projeto `meu1repo`, iniciado no capítulo 3 e alterado na seção 1.1 (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`, 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'
```

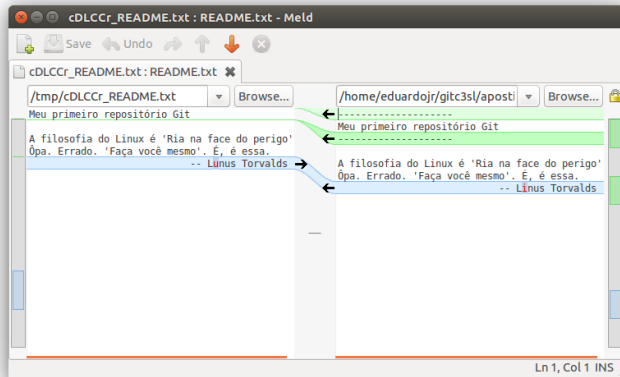


Figura 1.12: *Screenshot* do programa *meld* utilizado como difftool para o arquivo `README.txt` RabbitVCS

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 1.12 temos o *screenshot* do programa após executado

o `diff`tool. 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 `diff`tools, 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.

```
## 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 08034e6] Adiciona TODO e corrige README
 3 files changed, 11 insertions(+), 1 deletion(-)
 create mode 100644 TODO.txt
Switched to branch 'feature05'
[feature05 121c456] 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
```

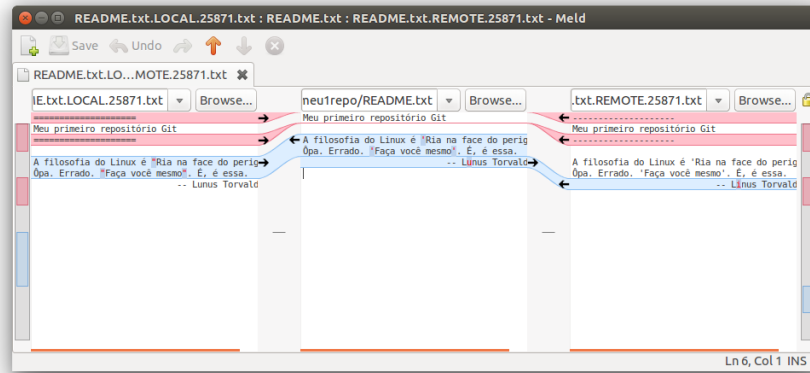


Figura 1.13: Screenshot do programa meld utilizado como difftool para o arquivo README.txt RabbitVCS

Na figura 1.13 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 1.12, em que o meld foi utilizado como diff tool, 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 3b8599c] 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 a opção -t (ou --tool). Porém podemos definir o meld como ferramenta padrão para resolução de merge, assim como foi feito com 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 -
```

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 1.13 e ainda a opção --output=\$MERGED informa que o arquivo a ser editado será sobrescrito sob a versão \$MERGED, que é criada pelo automaticamente 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 programada `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 `kdiff3`<sup>2</sup> e `P4Merge`<sup>3</sup>, 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 conjunto facilita o fluxo de trabalho adotado em um projeto sob versionamento.

---

<sup>2</sup>Disponível para *download* em <http://kdiff3.sourceforge.net/>

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