Manual de Edição do Manual d* Bix*

Centro Acadêmico da Computação da Unicamp – CACo
Rafael Sartori Martins dos Santos Henrique Noronha Facioli

2018

Sumário

Prefácio

Como você já viu, o Manual d* Bix* é uma das tradições do CACo que tem um grande impacto, em especial por conter diversas informações úteis para quem está chegando na Unicamp, além disso, é um ótimo jeito de apresentar o centro acadêmico. Nosso manual é, de certa forma, uma referência na Unicamp por ter um conteúdo bem atualizado e completo, mas isso tem um custo: temos que revisá-lo todo ano. Nossa tática é concentrar o trabalho de edição no fim do ano, começando entre outubro e novembro (na calmaria, antes do caos das provas finais) e dando um gás em dezembro, logo após o fim das aulas.

É fácil perceber que isso não é um trabalho simples e requer a participação em peso da gestão, mas, ao mesmo tempo, por utilizar `git` e LaTeX, isso pode ser um tanto quanto impeditivo para a participação de novas pessoas que não tem tanta familiaridade com essas ferramentas. Por isso, aqui tento deixar um guia de como começar a colaborar. Tornando assim mais a fácil a participação de outros, além de ter uma forma de registrar as coisas que funcionaram bem no passado.

Consideração importante: `git` é uma ferramenta muito poderosa e, como sabemos, com grande poder, vem grandes responsabilidades. Fazer caquinha é **MUITO** fácil em `git`, por isso tenha cuidado e pesquise tudo antes de executar um comando. A parte ruim dessa ferramenta é que os tutoriais e a documentação, apesar de rigorosos e claros, não são didáticos, então boa sorte! Espero que este manual ajude a entender ao menos o que você deve fazer e sirva como um guia para os vários nomes que eu tive que descobrir com várias horas de pesquisas.

Justamente por ser uma ferramenta muito poderosa, há muitos caminhos para fazer diversas coisas. O caminho mostrado aqui foi o que eu conheci e achei mais didático. Muito provavelmente existe um caminho mais fácil para fazer o que fiz e, com certeza, há um caminho mais difícil, então use o manual para aprender a colaborar, mas não use como guia definitivo de `git`.

Nós, membros do CACo, agradecemos a colaboração! Espero que goste do manual.

Capítulo 1

Preparando para edição

1.1 Instalação de ferramentas

No caso do Linux, além do seu editor de texto preferido, precisará de alguns pacotes para compilar o manual (e a maioria dos projetos) em LaTeX:

- `git`
- `scons`
- `texlive-bin`
- `texlive-core`
- `texlive-fontsextra`
- `texlive-pictures`
- `texlive-latexextra`
- `texlive-formatextra`
- `texlive-bibtexextra`

Crie, se não possuir, uma conta no GitHub (lembre-se de que, se você adicionar algum e-mail da Unicamp – seja como secundário ou principal –, poderá adquirir uma conta de desenvolvedor).

1.2 Configuração de projeto

Em GitHub, a rede social de compartilhamento de código com suporte a ferramenta `git`, suporta a clonagem de projetos para a sua conta, assim você pode fazer coisas que para você talvez pareçam complexas, como pedir para seu código ser anexado ao projeto principal ou ainda criar um projeto paralelo com base em outro. Este processo de clonagem chama-se *forking*.

Abra a página do manual no GitHub e crie uma *fork* do manual na sua conta (clique em *"Fork"* no topo da página). Após ser redirecionado ao repositório – nome dado ao conjunto de arquivos do seu projeto, seja local ou remoto – na sua conta, copie o endereço dado em *"Clone or download"*. Abra, então, o terminal, navegue até a pasta que gostaria de possuir a pasta do manual e utilize o comando:

```
git clone https://github.com/(seu usuário)/Manual-do-Bixo.git
```

O download do repositório irá começar (e irá demorar um pouco). Por ter feito um clone, o seu projeto no `git` terá a configuração de *"remote"*, que é um *link* para o repositório na nuvem, ou seja, é a ligação que `git` usará para enviar/receber informações, sincronizando o projeto. Você pode verificar os *remotes* disponíveis utilizando o comando:

`git remote -v`, onde `-v`, `--verbose` é usado para mostrar detalhes

Notará que há apenas a ligação para o seu repositório no GitHub – nomeada por convenção de *origin*. O *link* para o qualquer outro *fork* ou para o repositório do CACo será necessário para a colaboração no projeto comunitário. Por convenção, adicionaremos o ponto zero do projeto, o repositório do CACo, com o nome de *upstream*. Vá até a página do projeto no GitHub do CACo, pegue o endereço do repositório através do *"Clone or download"* e utilize o comando:

```
git remote add (nome) (endereço)
```

No nosso caso:

```
git remote add upstream
https://github.com/cacounicamp/Manual-do-Bixo.git
```

Com isso, será possível sincronizar com o projeto comunitário através da ligação nomeada *upstream*, já que o *origin* refere-se apenas ao **seu** repositório no GitHub, que não é o ponto central do projeto. Oh, desculpe, não quis parecer rude, sua participação é importante. Se te faz sentir melhor, você é o centro do nosso projeto como centro acadêmico.

Agora você tem tudo pronto! Pode conferir o estado da sua *branch* – algo similar a uma "linha do tempo" – com relação aos arquivos modificados e sincronização com algum *remote* usando:

```
git status
```

1.3 Conceitos importantes de `git`

O que é uma *branch*?

Uma *branch* é algo criado a partir da ramificação do projeto que permite alterá-lo independentemente do outro ramo. Por padrão, o ramo principal é chamado de *master*, aqui você aprenderá a fazer suas contribuições para o manual através de *branches* nomeadas de maneira breve com o fim de descrever o que ela altera.

Imagine que a *branch master*, o ramo principal do nosso projeto visto como uma árvore, é o tronco. Imagine que cada centímetro de altura é uma mudança no projeto. As ramificações criadas são galhos que saem de algum ponto do tronco, ou seja, *branches* precisam de um ponto de partida, uma altura da qual o galho começa a crescer.

A partir do ponto mais recente do projeto, o topo do tronco, a alteração mais recente da *branch master*, é onde você (normalmente) cria as ramificações. A partir delas, você pode fazer mudanças que não interferem no caminho principal do projeto, o que é útil para desenvolvimento, teste ou mesmo experimentos. Isso é importante em projetos comunitários pois permite a implementação, correção ou até remoção de alguma parte do projeto facilmente.

Entendendo por que uma *branch* se assemelha a uma "linha do tempo"

Eu chamei *branch* de linha do tempo pois, ao criar uma, você passa a modificar seu trabalho independente das outras *branches*, as suas alterações são totalmente independentes do resto do projeto. Imagine um aplicativo como o *WhatsApp*. Os desenvolvedores querem fazer algo decente finalmente: uma versão desktop que não depende do celular conectado a todo momento.

Imagine que usam a *branch master* para a linha do tempo que irá para as lojas de aplicativos. Eles criam a *branch desktop-app*, por exemplo, para modificar o aplicativo de tal forma que permita você conectar com o celular apenas quando cria novas conversas (pela questão de encriptação de ponta-a-ponta). Eles desenvolvem essa *feature* enquanto outros desenvolvedores corrigem bugs, melhoram a performance, ou seja, totalmente independente da questão de transformar a versão desktop em algo bom. Isso evita, ainda, a liberação para o público antes de tudo funcionar perfeitamente.

Para imaginarmos a situação em que não usam *branches*, todos os desenvolvedores estariam alterando os mesmos arquivos juntos, misturando todo o código antigo com o código de correção de bugs e melhora performance com o código da nova *feature* e, além disso, não poderiam liberar as correções até tudo estiver funcionando, já que, neste mundo, tudo estaria na *master* e não iriam liberar uma *feature* quebrada. Além disso, reverter alterações seria muito mais complexo sem ramificações.

Entendendo a ideia de *pull request*

Espero que tenha entendido o que é uma *branch*, pois iremos utilizá-la! Já que o centro do manual fica na conta do GitHub do CACo, a ideia é que você faça uma *branch* apenas com a alteração direta do que quer fazer para fazer uma *pull request* no GitHub. Você já vai entender o que queremos dizer.

Pode não ser o caso, mas imagine que pegou a versão impressa do Manual d* Bix* do ano anterior e marcou com marca texto as partes que gostaria de alterar, ou seja, você tem uma lista de coisas para alterar na versão deste ano, coisas que podem estar na mesma categoria ou não. Para não precisar fazer 54 *pull requests* sobre cada coisinha que você quer alterar, mas, ao mesmo tempo, não juntar tudo em apenas uma, tornando todas as alterações um pouco vagas e impedindo que o projeto seja retrocedido caso precisasse, você precisa juntar em mesma categoria as mudanças parecidas.

Por exemplo, tem 5 referências a restaurantes que não existem mais, a ideia da *pull request* é fazer a alteração direta e bem descrita em uma linha. No nosso exemplo, a *pull request* se chamaria "Removendo restaurantes que fecharam" e a *branch*, algo como `remover-restaurantes`. A *pull request* alteraria quantos arquivos precisarem, mas abordaria todas as remoções de restaurantes que fecharam entre este ano e o ano pasado.

O caso problemático seria se fizesse uma *pull requests* para cada restaurante removido, precisando da aprovação individual de cada uma para ser aceito na *branch master* do projeto. Ou ainda, se fizesse uma *pull request* para todas as 54 alterações, mas tivesse removido por engano um restaurante que continua aberto, fazendo outra pessoa (ou você mesmo) ter que reescrever a parte do manual, já que não seria prático reverter uma *pull request* tão grande.

Ou seja, faça as alterações em conjunto, assim podem ser revertidas e realteradas se precisar, sem a necessidade de movimentar uma quantidade enorme de mudanças ou movimentar dezenas de vezes pequenas mudanças. Trata-se de um equilíbrio.

Capítulo 2

Modificando o manual

2.1 Preparando o seu repositório para mudanças

Há alguns passos a seguir para evitar a desordem do seu repositório local. Essa desordem pode te obrigar a recomeçar o projeto em uma outra pasta ou ainda fazer uma complicada sequência de comandos. A primeira dica é não utilizar a *branch master*, deixe ela parada para ser uma cópia da *branch upstream/master* (esta é a notação usada para identificação de uma *branch* em algum *remote*), assim você sempre terá o estado atual do projeto comunitário sem precisar baixar mais nada.

Sincronização com o projeto

Antes de sair criando *branches* para todas as suas 54 alterações, já que você descobriu que é ruim fazer tudo na *master*, é necessário uma sincronização com o repositório do CACo, aquele cujo *remote* é nomeado *upstream*. Isso deve ser feito no mínimo 2 vezes: quando a *branch* é criada e quando todas as alterações tiverem sido realizadas, logo antes da *pull request* ser criada, assim as suas alterações se fundirão automaticamente, visto que a sincronização resolve colisões – você entenderá o que são colisões em breve.

Para realizar a sincronização, você tem algumas opções:

- Deixar a sua *branch master* sincronizada com a *upstream/master* e fazer qualquer nova *branch* partindo da sua *master* local.
- Ignorar a existência da sua *branch master* e fazer a nova *branch* a partir da *upstream/master*.
- Misturar os dois, mantendo sua *branch master* atualizada (caso você precise do PDF atual do manual, no nosso caso), mas partindo da *branch upstream/master* na criação da sua nova *branch*, para ter certeza que sempre partirá do estado mais recente do projeto.

Recomendamos a última, visto que o trabalho de sincronização não é difícil quando as mudanças não colidem. É recomendável que você esteja **SEMPRE** sincronizado, evitando uma bola de neve de colisões. Mas, afinal, o que são colisões?

Colisões ocorrem quando há alterações na *upstream/master* em arquivos que você já modificou. Quando o `git` não consegue solucioná-las automaticamente, você precisará fazer a fusão manualmente. Parece difícil, mas, na maioria das vezes, basta adicionar as duas versões (a sua e a da *upstream*, já que geralmente são adições de parágrafos independentes), raramente haverá um caso em que duas pessoas alteraram o mesmo parágrafo ao mesmo tempo. Se acontecer, você precisará reescrever o parágrafo fazendo uma fusão das ideias das duas modificações ou manter apenas uma ou outra versão. É fácil.

Agora que talvez tenha entendido a necessidade da sincronização, explicarei como fazê-la. Basta executar o comando a seguir:

```
git pull (remote) (branch)
```

Ou, no nosso caso:

```
git pull upstream master
```

Esse comando irá baixar (o mesmo que executar `git fetch upstream master`) e juntar os arquivos atualizados, baixados com os arquivos locais resolvendo colisões se houver (o mesmo que `git merge FETCH_HEAD`, onde *`FETCH_HEAD`* é o nome que se dá ao estado do projeto baixado, ou seja, é o nome do ponto da linha do tempo que foi baixada).

Criando uma *branch*

Para criar uma *branch*, é necessário um ponto de partida. Como disse anteriormente, pode ser um ponto local ou não. Dê preferência ao *upstream/master*, já que é o estado atual do projeto comunitário. Usamos o comando:

```
git branch (nome) remote/branch
```

Temos vários exemplos:

```
git branch manual-de-edicao upstream/master
git branch manual-de-edicao origin/master
```

`git branch manual-de-edicao master`, onde *master* é a local

Note como é sutil a diferença do repositório ligado pelo *remote origin* e o repositório local. Em maioria, o repositório em nuvem estará desatualizado em relação ao seu repositório local pois é você quem deve enviar as sincronizações para ele. Ou seja, se o repositório comunitário (*upstream*) fosse atualizado e você tivesse sincronizado localmente, ainda precisaria enviar as mudanças para o repositório na nuvem – logo verá como enviar alterações para a nuvem.

Lembra-se que o comando `git status` mostrava o estado atual dos arquivos e a *branch* que estava ativa? Não se esqueça de mudar para a *branch* desejada usando:

```
git checkout (branch)
```

Se houver arquivos modificados, o comando irá falhar, pois mudar entre *branches* apagaria as mudanças que não foram guardadas através de *commits*. Você pode guardar as mudanças numa pilha e restaurá-las quando quiser, seja no mesmo *branch* ou em outro, usando os comandos:

- Para guardar: `git stash` ou `git stash --include-untracked` para incluir os arquivos que não foram adicionados ao projeto do `git` como falarei no futuro.
- Para restaurar: `git stash pop`, que colocará todos os arquivos no estado que foram guardados.

2.2 Alterando de fato o manual

Modificando uma *branch*

Agora você sabe criar e pular de *branch* em *branch*, como um verdadeiro macaco (não confundir com o movimento anti-CACo!). Basta, então, modificar os arquivos e usar o comando `scons` na raiz do projeto para compilar o PDF do manual e testar suas modificações. Verá que LaTeX é uma linguagem fácil de entender e escrever, "ver e repetir" é uma técnica que funciona incrivelmente bem!

Leia as considerações importantes no *README.md* no projeto do Manual d* Bix* no GitHub! Respeite as convenções de edição, inclusive a de que toda regra tem exceção se você achar necessário.

Salvando as modificações

Em `git`, apenas alguns arquivos são guardados como do projeto (arquivos temporários, auxiliares ou de configuração são muitas vezes deixados de fora). Esses arquivos são chamados de *staged files*, são aqueles adicionados e modificados em cada *commit*, aqueles baixados pelo GitHub, considerados "arquivos DO projeto".

Para adicionar um arquivo novo ao projeto ou adicionar as modificações a uma *commit*, você utiliza o comando:

```
git add (arquivo 1) (arquivo 2) ...  (arquivo n)
```

Poderá verificar os arquivos adicionados ao que virá a ser o *commit* usando o comando `git status`. Renomear um arquivo pode ser algo complexo, pois `git` não perceberá tão automaticamente que tal arquivo que foi removido é aproximadamente igual àquele que foi adicionado. Para evitar essas atrapalhadas, use `git mv` para renomear e `git rm` para remover arquivos.

Finalmente, para criar uma *commit*, guardando todas as alterações dos arquivos adicionados em relação ao "estado" anterior da *branch*, utilize o comando:

```
git commit
```

O `git` abrirá o seu editor preferido (você pode mudar qual é usando as configurações globais, pesquise como fazer se quiser) para escrever a mensagem que descreve as alterações. O GitHub reconhecerá a primeira linha (dependendo do seu comprimento) como o nome da *commit*, as outras linhas só aparecerão ao abrí-la no navegador e servem de descrição.

A *commit* estará associada a *branch* e guardará as diferenças dos arquivos adicionados à *commit* em relação ao estado anterior do projeto. Recomendei

o uso dos comandos de renomear e excluir do `git` pois para adicionar um novo arquivo e apagar outro manualmente como *staged file* é um pouco mais trabalhoso.

2.3 Enviando as alterações para a nuvem

Agora que você já fez várias *commits* e talvez tenha acabado tudo o que sua *branch* representava, está na hora de enviar para a nuvem! Para isso, basta utilizar:

```
git push (remote) (branch)
```

Apesar de talvez achar que é improvável você criar várias *branches* simultaneamente e precisar enviar juntas várias alterações à nuvem, é mais comum do que parece. Um exemplo que acontecerá é o caso da *upstream/master* ter sido atualizada, assim é recomendável que você sincronize todas as suas *branches*, fazendo o que `git` chama de *merge*, o trabalho de juntar *branches* em um ponto comum resolvendo conflitos e criando uma *commit*. Então, para facilitar e enviar as *commits* de todas as *branches* (ao invés de mandar uma por uma), poderá usar o comando:

```
git push (remote) --all
```

Se abrir seu GitHub após criar a *branch* e enviar algumas *commits*, verá que agora existe a *branch* criada na lista e que há *commits* que não estão na fonte do seu projeto (“`x` *commits ahead of cacounicamp/Manual-do-Bixo*”), ou seja, na *upstream*. Haverá um botão para criar uma *pull request*, onde você pode descrever as mudanças ou até pedir participação de outras pessoas se precisar, pois, assim como você usou a *cacounicamp/Manual-Do-Bixo* como *upstream*, fonte do projeto, outra pessoa pode usar o seu repositório e participar de forma recursiva.

Após criada a *pull request*, membros do CACo poderão revisar e aprovar as modificações ao manual, verificando regras, se tudo foi compilado com excelência através de uma métodologia rígida de *try and error* que os membros do CACo estão acostumados a fazer.

O que aprendemos até aqui

Agora você aprendeu a sincronizar o repositório local com o remoto, seja ele a *upstream* ou o seu *origin*, seja enviando ou recebendo informações. Aprendeu também a criar *branches* e alterá-las adicionando *commits*. Por fim, aprendeu a enviar as mudanças ao GitHub e criar *pull requests*, que era o passo final para a participação individual no projeto.

No próximo capítulo, aprenderemos a limpar o seu repositório, local e remoto, a *origin*, após suas mudanças forem verificadas e aprovadas no projeto principal. Veremos em outro ainda como participar em grupo numa *pull request*, que servirá também aos membros da gestão a manterem o manual, ajudando os colaboradores com as regras.

Capítulo 3

Cuidando do repositório após mudanças

Antes de sair apagando *branches* que já foram aceitas no projeto principal, deixe-as abertas por um tempo. Caso algum problema estivesse passado pela revisão dos mantenedores do projeto, a sua alteração deverá ser revertida e revisada. A maneira mais rápida e fácil para arrumar o erro acontece se a *branch* estiver ainda ativa, pois bastará fazer algumas alterações para tornar a *pull request* válida novamente.

O que quero dizer é que você deve esperar alguns dias ou mesmo horas até a sua alteração ser consolidada no projeto. A razão para isso é que, no projeto do Manual d* Bix* do CACo, as *commits* são transformadas em uma, tornando mais limpo o histórico do projeto no geral, mas com o custo de desorganizar os repositórios locais após terem suas mudanças aceitas. Ou seja, após uma *pull request* ter sido aceita, a sua *branch* se tornará incompatível com o projeto e não poderá ser facilmente re-utilizada, senão com a reversão da *pull request* em casos de erro.

3.1 Excluindo a *branch* local e remota

Se já passou um tempo desde quando sua *pull request* foi revisada e aceita, podemos começar a limpar o seu repositório local. Primeiro, apagamos a *branch* que originou a *pull request* utilizando o comando:

```
git branch -D (branch)
```

A opção `-D` vem de `--delete --force`, você precisa forçar a exclusão já que tornar as várias *commits* em uma única no repositório do CACo implicará na incompatibilidade da sua *branch*, visto que `git` não reconhecerá que os arquivos estão identicos no final, mas sim que a quantidade de *commits* são diferentes.

Para enviar uma exclusão de *branch* para o repositório remoto, utilize o comando:

`git push (remote) -d (branch)`, onde `-d` significa deletar

Capítulo 4

Mantendo o repositório remoto e revisando colaborações

A necessidade disso tudo (separar em *branches*, fazer *pull requests*) já foi comentada. É uma maneira de desenvolver facilmente em grupo, seja mantendo o código como membro do CACo ou ainda como colaborador.