

## Anexo 7 - GIT - Conceitos

### 7.1 - Introdução



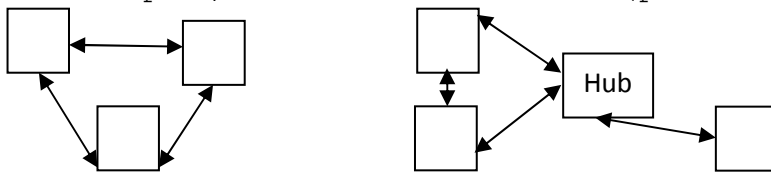
O Git é um sw de controle de versões distribuído. Ele foi desenvolvido tendo-se em mente vários grupos colaborando de forma independente / autônoma, porém sem perder o controle do que está sendo feito. O Git veio a atender os seguintes problemas:

- Indenpendência de um servidor centralizado;
- Foco na aplicação e não nos arquivos de forma individual;
- Permitir que vários grupos colaborem e compartilhem o código, incluindo o desenvolvimento de várias versões da aplicação em paralelo.

Bom, a melhor forma de entender o Git é comaprá-lo com o CVS que todos conhecem.

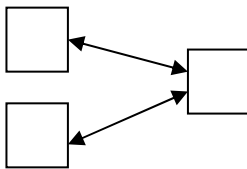
#### Quanto a topologia:

Git: distribuído puro, ou distribuído com hubs (parciais e / ou final).



Como exemplo de hubs final temos o GitHub e o Google que hospedam o código mais recente de um determinado projeto. O objetivo é todo mundo atualizar o hub final e obter o código mais recente do mesmo. Os hubs parciais podem ser configurados, por exemplo, para receber código de desenvolvedores externos, depois, um desenvolvedor interno atualizará o hub final a apartir do hub parcial.

CVS: estrela (centralizado).



#### Unidade de Commit:

Git: Tree (árvore podendo conter vários arquivos).

CVS: arquivo.

#### Unidade de Versão:

Git: aplicação.

CVS: arquivo.

#### Vantagens / Desvantagens:

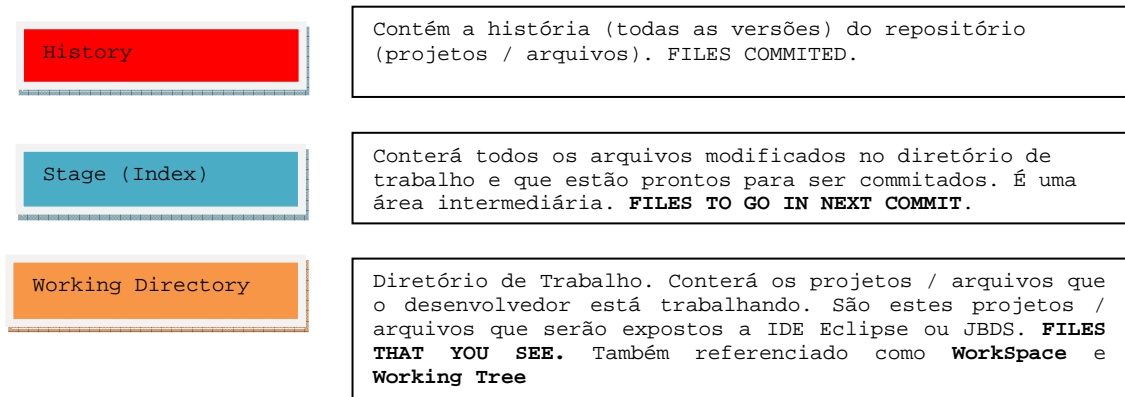
<b>Grande Vantagem do CVS</b> Quando você pega um arquivo para trabalhar, você tem garantias que ele não será alterado por outra pessoa.	<b>Grande Vantagem do Git</b> Como a unidade de versionamento é a aplicação, fica muito fácil voltar o estado da mesma a qualquer tempo / versão.
<b>Grande problema do CVS`</b> Como a unidade de versionamento é o arquivo, fica difícil, ou mesmo impossível, retornar uma aplicação a	<b>Grande problema do Git</b> Supor que você pegue um arquivo para alterar em t0. Em t1 um outro usuário altera (ou até mesmo apaga) o arquivo



O HEAD pode estar apontando para o branch master ou desvio1, dependendo da opção do desenvolvedor (switch branch).

#### Esquema Git em 3 camadas:



É muito comum a apresentação de esquema Git, principalmente para mostrar a aplicação de comandos, em forma de 3 camadas.



### 7.3 - Breve comentário sobre os principais comando Git

#### 7.3.1 - Trabalhando com repositório local

Comando	Comentário
<b>add files</b>  <code>git add -p</code> , instead of (or in addition to) specifying particular files to interactively choose which hunks copy	copies <i>files</i> (at their current state) to the stage.
<b>commit</b>  <code>git commit -a</code> is equivalent to running <code>git add</code> on all filenames that existed in the latest commit, and then running <code>git commit</code> .  <code>git commit files</code> creates a new commit containing the contents of the latest commit, plus a snapshot of <i>files</i> taken from the working directory. Additionally, <i>files</i> are copied to the stage.	saves a snapshot of the stage as a commit.
<b>checkout -- files</b>  <code>git checkout HEAD -- files</code> copies files from the latest commit to both the stage and the working directory.  <code>git checkout -p</code> , instead of (or in addition to) specifying particular files to interactively choose which hunks copy	copies <i>files</i> from the stage to the working directory. Use this to throw away local changes.
<b>reset -- files</b>  <code>reset -p</code> , instead of (or in addition to) specifying particular files to interactively choose which hunks copy	unstages files; that is, it copies <i>files</i> from the latest commit to the stage. Use this command to "undo" a <code>git add files</code> . You can also <code>git reset</code> to unstage everything.
<b>merge</b>  Obs: em múltiplos branches um merge cria um simples commit com dois pais, deixando a	creates a new commit that incorporates changes from other commits. Before merging, the stage must match the

<p><b>história não linear.</b> Para deixar a história linear deve-se usar o rebase no lugar.</p> <p>Obs: o merge está sujeito a conflitos</p> 	<p>current commit. The trivial case is if the other commit is an ancestor of the current commit, in which case nothing is done. The next most simple is if the current commit is an ancestor of the other commit. This results in a <i>fast-forward</i> merge. The reference is simply moved, and then the new commit is checked out.</p>
<p><b>cherry-pick</b></p>	<p>"copies" a commit, creating a new commit on the current branch with the same message and patch as another commit.</p>
<p><b>Rebase</b></p> <p>Obs: em múltiplos branches um merge cria um simples commit com dois pais, deixando a história não linear. Para deixar a história linear deve-se usar o rebase no lugar.</p> <p>Obs: o rebase está sujeito a conflitos</p> 	<p>is an alternative to a merge for combining multiple branches. Whereas a merge creates a single commit with two parents, leaving a non-linear history, a rebase replays the commits from the current branch onto another, leaving a linear history. In essence, this is an automated way of performing several cherry-picks in a row.</p>



### Que tipo de conflito?

Supor o seguinte exemplo:

#### Repositório remoto (master)

20/02/12

05/06/2012

09/08/2012

Outro usuário alterou o fonte xpto.jsp e atualizou (push) o repositório remoto

#### Repositório Local (master)

20/02/12

05/06/2012

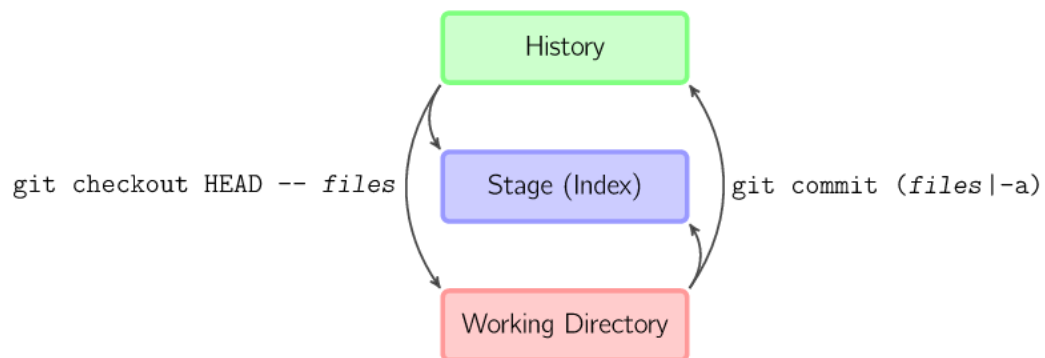
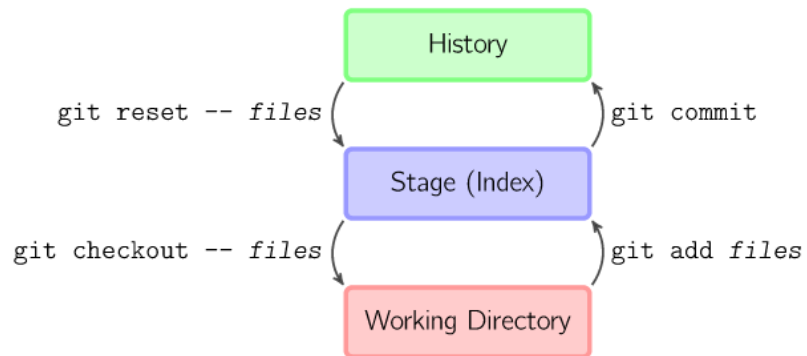
01/08/2012

o usuário alterou o fonte xpto.jsp

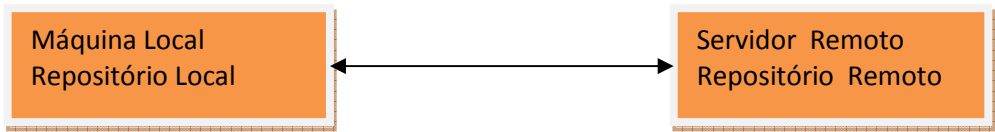
Se o usuário na máquina local realizar um PULL (fetch + merge), ou um MERGE (após o fetch no repositório remoto), o Git acusará um conflito no fonte xpto.jsp, conflito este que deverá ser administrado pelo usuário da máquina local, ou seja, receber o fonte mais novo do repositório remoto e aplicar as mudanças que ele realizou em 01/08/2012.

Um ótimo site para entender o funcionamento dos comandos locais do Git é o Visual Git Reference (<http://marklodato.github.com/visual-git-guide/index-en.html>), visto que utiliza o esquema de 3 camadas.

Abaixo, algumas ilustrações do site:



### 7.3.2 - Trabalhando com repositório remoto



Existem três tipos de branches no repositório máquina local:

- **Local branch (LB):** que é o branch tradicional criado pelo git branch;
- **Remote Tracking Branch (RTB):** são criados automaticamente quando **cloning** ou **fetching** repositórios remotos. Um RTB no repositório local sempre corresponde a um (local) branch no repositório remoto. O RTB aponta para o mesmo commit que o branch correspondente no repositório remoto, no momento do fetch / clone. RTBs podem ser usados para criação automatizada de "Upstream Configuration" dos branches locais;
- **Tracking Branch (TB):** vamos assumir por agora que o TB é idêntico ao RTB, porém o TB pode ser modificado pelo usuário, pois é read-write e o RTB não, pois é read-only. Algumas literaturas assumem o TB como Local TB ou LTB.

Bom, para diferenciar o **RTB** do **TB** é melhor apresentar uma tabela e um esquema visual, visto que esta distinção não é fácil de explicar somente no modo textual.

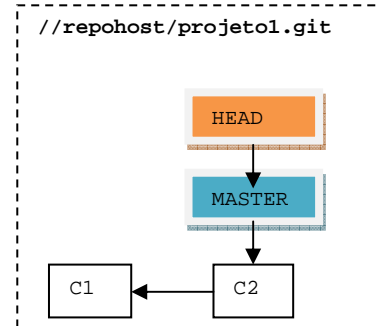
Tabela comparando **RTB** e **TB**

Tipo de Branch	RTB	TB
Diferenças		
Consegue seu conteúdo via	clone	clone
É atualizado via	fetch e pull. Ou seja, somente pelo repositório remoto	merge e pull. Ou seja, pelo repositório remoto eo pelo usuário (repositório local)
Client access	Read-only	Read-write
Pode-se publicar as mudanças via	Não pode	push

Esquemas visuais para ilustrar como o **RTB** e **TB** são criados e atualizados.

### SITUAÇÃO 0 - Criando o Repositório Remoto

Supor que tenhamos o seguinte repositório remoto.



As situações

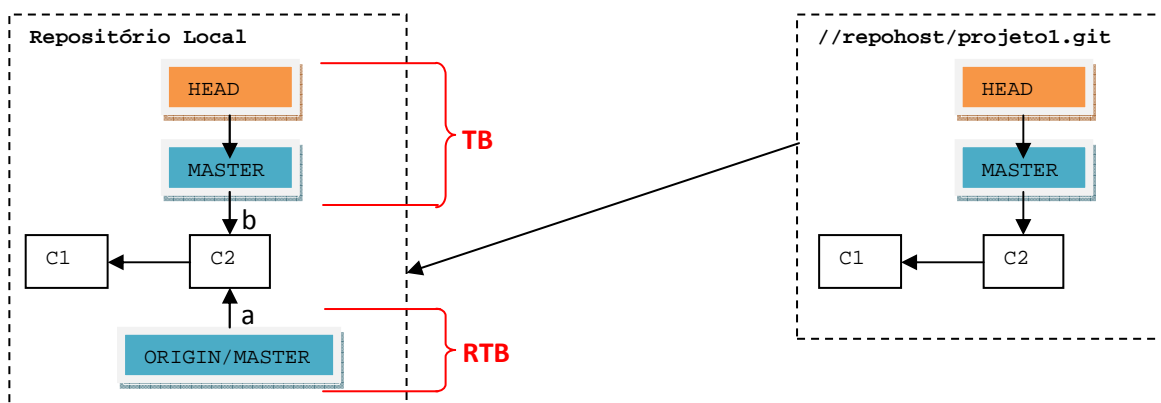
### SITUAÇÃO 1 - Clonando um Repositório Remoto com CLONE

Um usuário (diferente do usuário da situação 0) em sua máquina local, limpa, só com o Git instalado emite o seguinte comando

```
> git clone git://repohost/projeto1.git
```

Neste momento o commando clone fará uma série de coisas:

- a** - uma cópia do repositório remoto para o **RTB criado e nomeado de origin/master**. Este branch é basicamente read-only para o usuário local. Ele é atualizado somente a partir do repositório remoto.
- b** - **um novo TB é criado e nomeado de master**. O conteúdo do RTB origin/master é copiado para o TB master. O usuário pode fazer modificações neste branch.

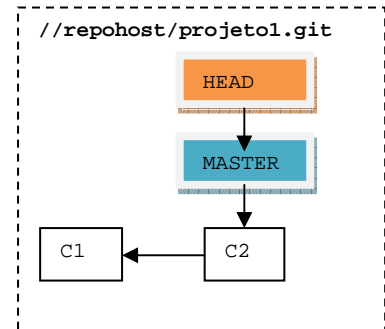
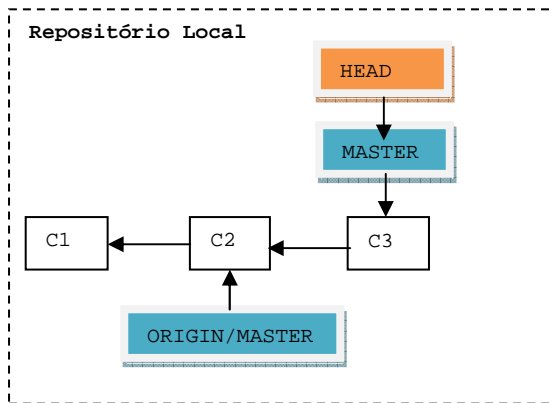


É interessante observar que a partir de agora temos dois branches (master e origin/master) que andam de forma independente.

## SITUAÇÃO 2 - Realizando mudanças no branch TB

Supor que o usuário faça o seguinte na máquina local:

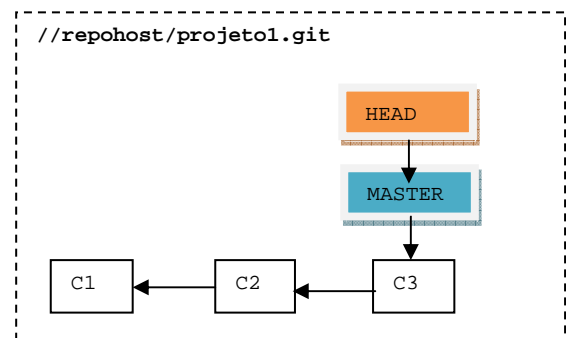
```
> altere algum arquivo  
> git add  
> git commit -m "minha primeira modificação"
```



Só o branch master se movimentou.

## SITUAÇÃO 3 - Modificações no Repositório Remoto

Retornando a situação 1, supor agora que alguém tenha alterado o repositório remoto:



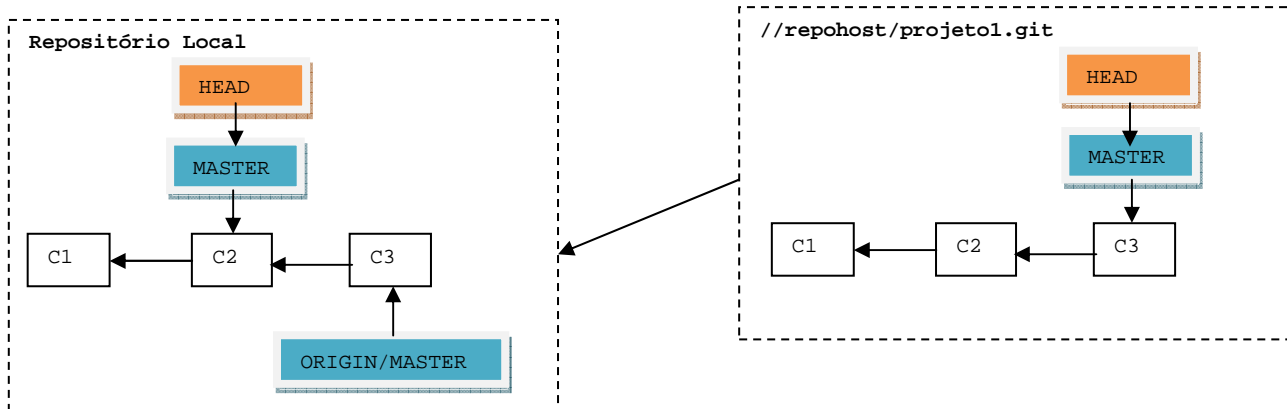


#### SITUAÇÃO 4 - Buscando novas atualizações no Repositório Remoto com FETCH

Retornando a situação 1.

Supor que o usuário faça o seguinte na máquina local:

```
> git fetch
```



Só o branch origin/master se movimentou.

E se quiséssemos atualizar o master (TB) no repositório local para apontar para o novo commit C3?

Neste caso, poderíamos emitir o seguinte comando:

```
> git merge origin/master
```

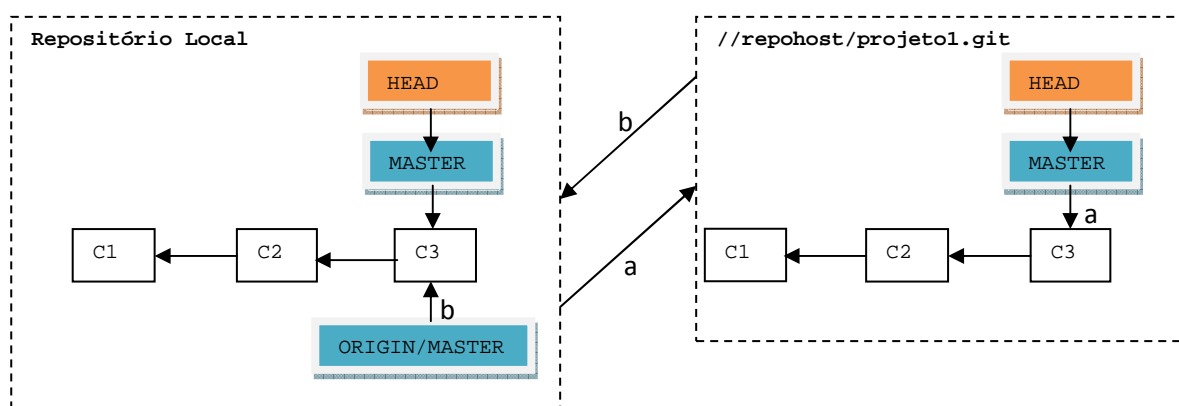
A sintaxe do merge é: `merge <branch-to-merge-from>`. Uma alternativa para o `git fetch` e `git merge` é o `git pull`.

#### SITUAÇÃO 5 - Publicando atualizações no Repositório Remoto com PUSH

Retornando a situação 2, supor que o usuário queira atualizar o repositório remoto.

Supor que o usuário faça o seguinte na máquina local:

```
> git push
```



Neste momento o commando push fará uma série de coisas:

**a** - o branch local, master, é publicado para o repositório remoto. Neste caso específico, o commit C3 é publicado no repositório remoto;

**b** - o RTB é também atualizado com novos updates a partir do repositório remoto. Neste caso específico, o branch origin/máster também aponatará para o novo commit C3.

Breve resumo dos principais comandos remotos do Git

Comando	Comentário
<b>clone</b>	O clone foi apresentado na situação 1. Como o próprio nome diz, ele clona um repositório. <b>Ele cria o TB e o RTB.</b>
<b>push</b>	A definição do manual é: Update remote refs along with associated objects , porém prefiro uma definição mais representativa tal como, Upload changes from your local repository into a remote repository. Ver situação 5.
<b>pull</b>	<b>Pull = Fetch + Merge.</b> Git pull" is exactly equivalent to "git fetch" followed by "git merge". Fetch a partir do repositório remoto e merge com o branch local corrente. <b>O PULL atualiza o TB e o RTB.</b>
<b>fetch</b>	Download novos branches e dados a partir do repositório remoto. <b>O FETCH só atualiza o RTB.</b>
<b>remote</b>	Lista, adiciona e deleta os alias de repositórios remotos

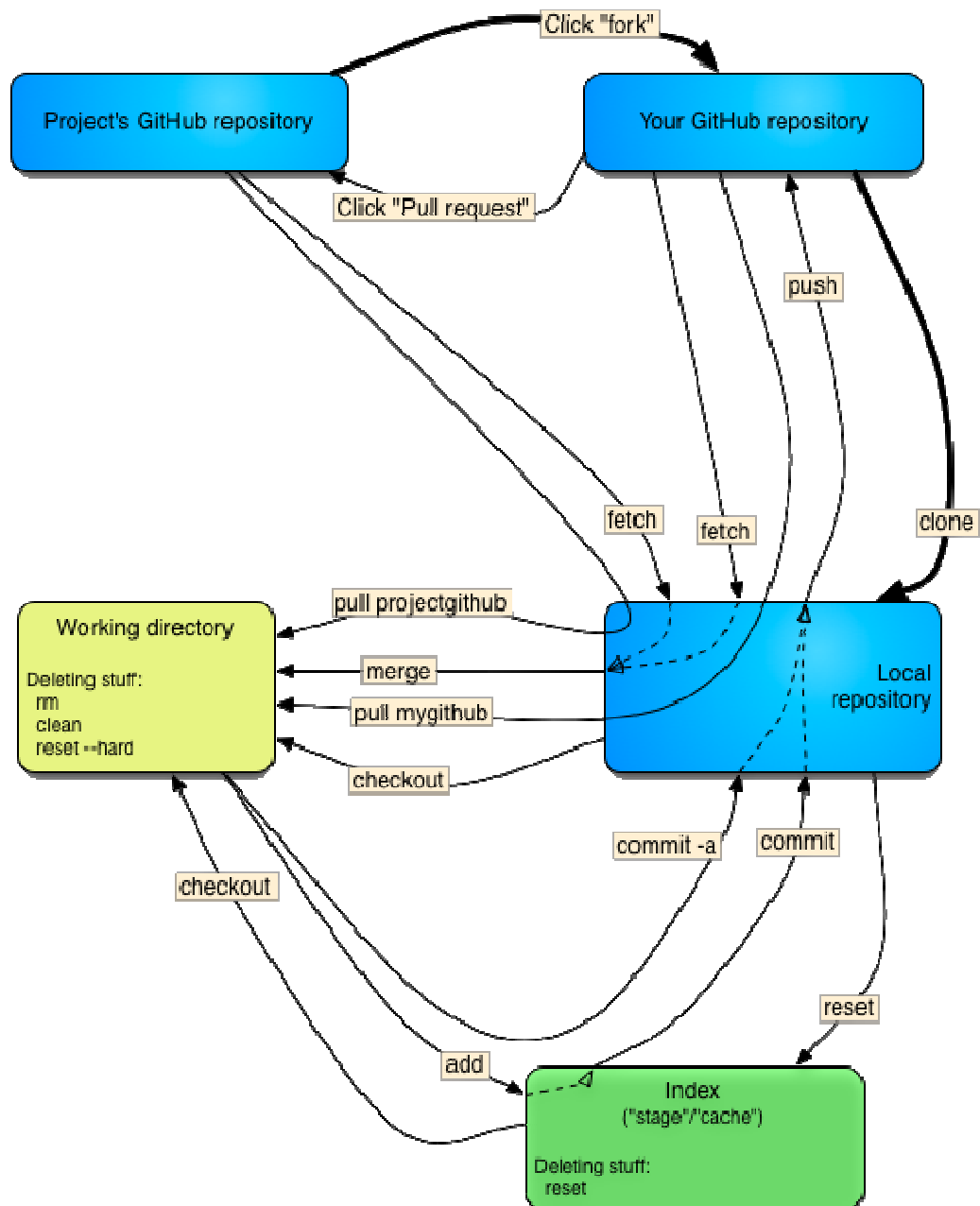
Um ótimo site para entender o funcionamento dos comandos remotos do Git de forma visual é o "Tracking Branches" And "Remote-Tracking Branches"

(<http://www.gitguys.com/topics/tracking-branches-and-remote-tracking-branches/>)

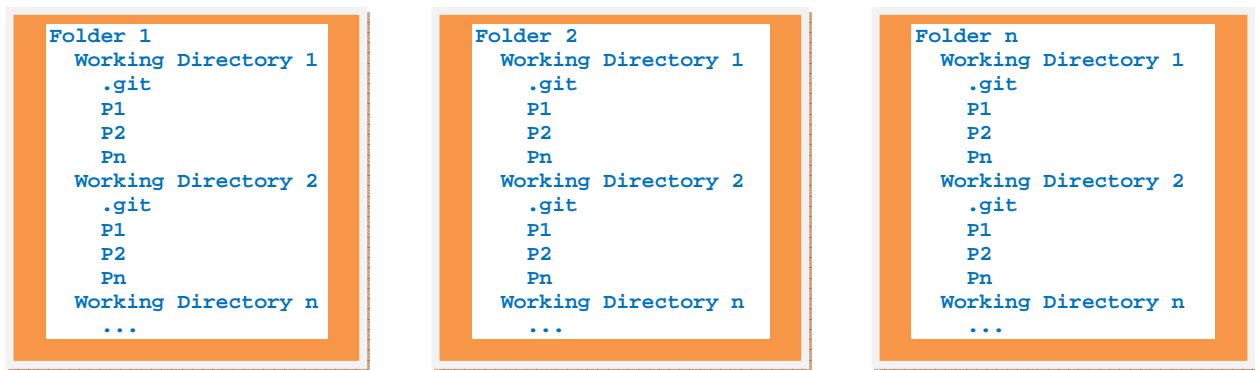
Uma referência visual completa incluindo comandos locais e remotos:

- Repositório local;
- Um repositório remoto, hub do usuário. É a partir deste hub que ele atualiza o hub do projeto;
- Um repositório remoto, hub do projeto.

Neste site (<http://steveko.wordpress.com/2012/02/24/10-things-i-hate-about-git/>), além do belíssimo resumo abaixo, o autor critica alguns aspectos do Git, principalmente a complexidade de se fazer tarefas simples.



#### 7.4 - O repositório do Git do disco local c:\



Folder (ou Pasta): localização a partir da qual a estrutura Git será criada. Podemos ter 1 ou mais Folders criados / instalados. O Folder pode ter qualquer nome.

Working Directory: é o diretório de trabalho do Git. Ele contém a pasta .git e os projetos. Podemos ter 1 ou mais Working Directory por Folder. O Working Directory pode ter qualquer nome.

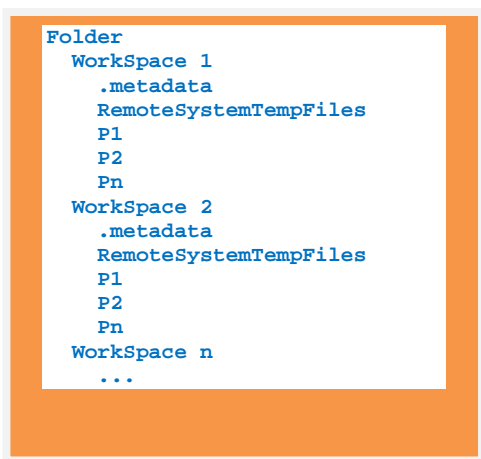
.git: este diretório contém os metadados, tabelas e índices que o Git utiliza.

P1, P2 e Pn: são os projetos os quais trabalhamos e que possuem os arquivos mais recentes. Equivale ao Workspace do Eclipse.

#### 7.5 - Integração Git e IDE (no caso, o Eclipse)

Como já vimos antes, o JBDS foi desenvolvido sobre o Eclipse. Na literatura encontraremos referência para o Eclipse Git, ou EGit. No anexo 8 falaremos sobre o EGit, porém antes é bom diferenciar os dois ambientes no que tange aos seus repositórios. O JBDS / Eclipse trabalha com o conceito de Workspace e o Git, como visto no item anterior, com o conceito de Working Directory.

Estrutura do Workspace



Folder: localização a partir da qual a estrutura Eclipse será criada. O Folder pode ter qualquer nome.

Workspace: é o diretório de trabalho do Eclipse. Ele contém a pasta .metadata e os projetos. Podemos ter 1 ou mais WorkSpaces por Folder. O Workspace pode ter qualquer nome.

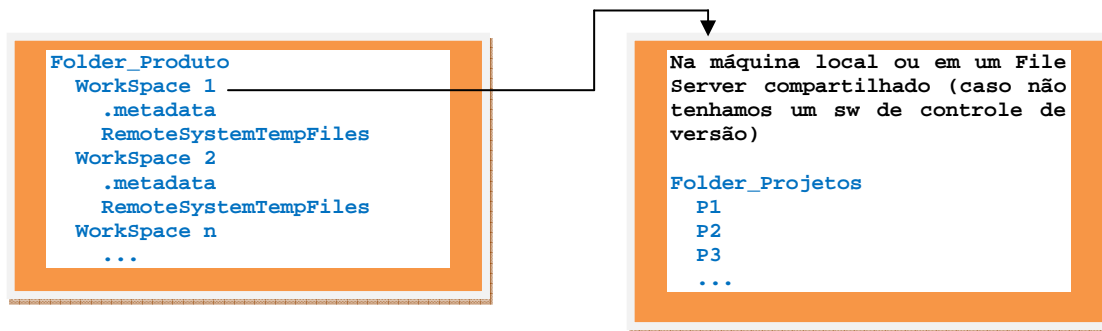
.metadados: este diretório contém os metadados, plugins, configurações que o Eclipse utiliza.

P1, P2 e Pn: são os projetos os quais trabalhamos e que possuem os arquivos mais recentes

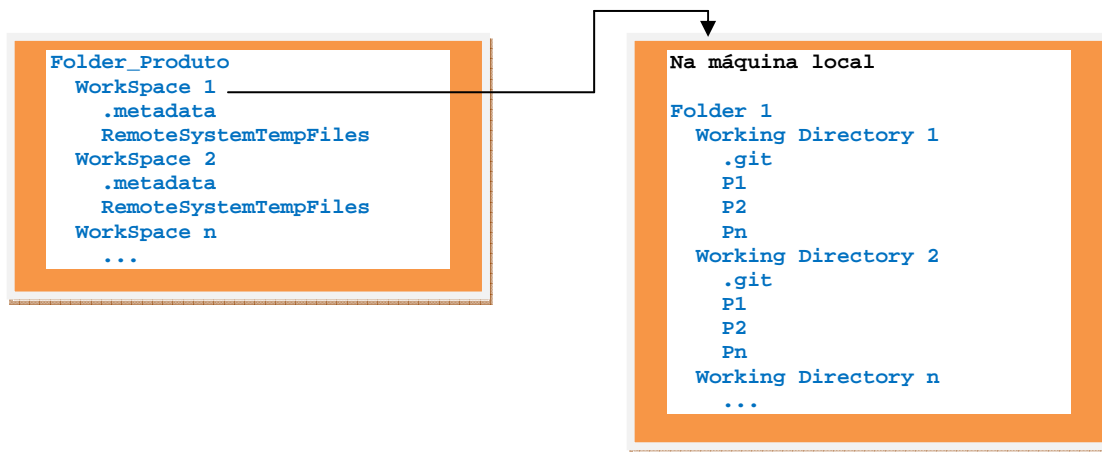
Esta é a estrutura default e não é aconselhável por diversos motivos:

- Nesta forma os projeto não estão compartilhados;
- Se quisermos ter versões diferentes do eclipse instaladas;
- Os metadados são muito acessado e realizam concorrência com os projetos.

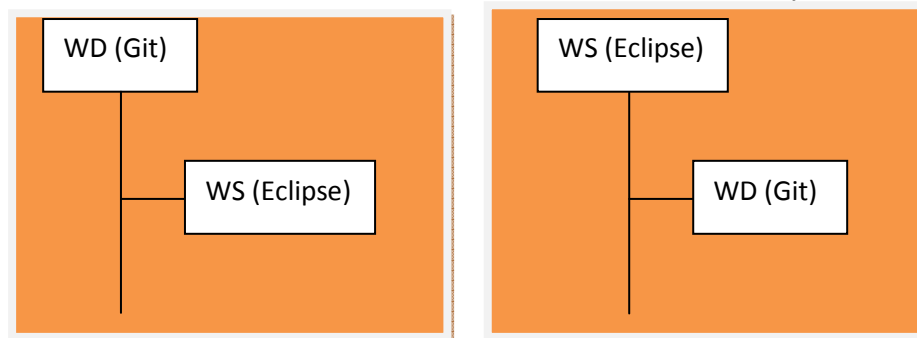
Desta forma, dá-se preferência para este tipo de estrutura:



Caso tenhamos também o Git, esta é a melhor estrutura:

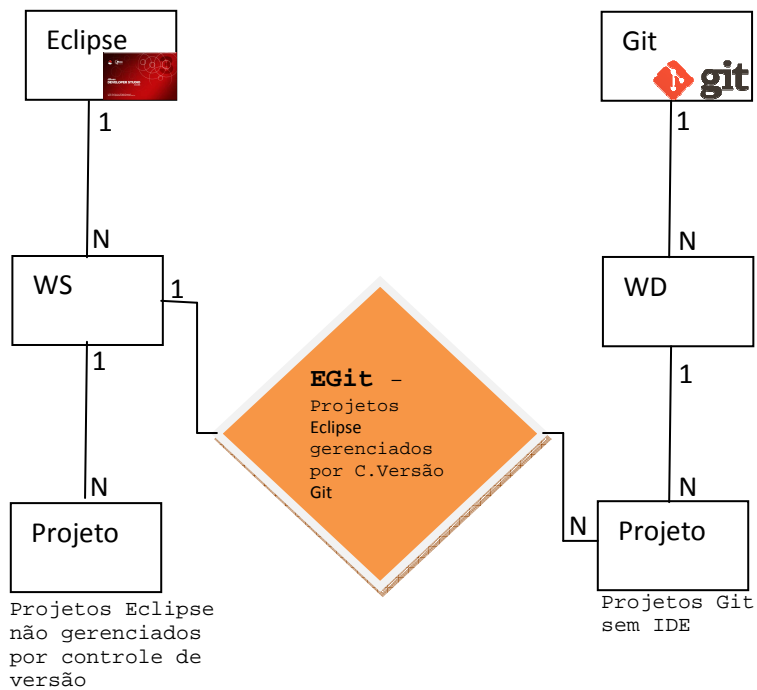


#### NUNCA DEVEMOS UTILIZAR UMA DESTAS CONFIGURAÇÕES



São inúmeros os motivos que vão desde a baixa performance a inconsistências diversas.

A partir destas informações, podemos desenhar o seguinte modelo para os dois produtos Eclipse e Git:



Para ilustrar os conceitos anteriores listei os repositórios Git e Eclipse da minha máquina, drive c:\.

## Caso Real: Relacionamento dos Repositórios Git X Eclipse

