# Roteiro de exercícios sobre Gerência de configuração, disciplina Engenharia de Software e Sistemas

## Paulo Borba

## Centro de Informática

## Universidade Federal de Pernambuco

Adicione as respostas imediatamente após as perguntas, usando uma outra cor. Leia com calma as instruções, e reflita com cautela sobre as consequências de cada comando.

1. O sistema de gerenciamento de metas (SGM) está armazenado no GitHub. Segundo as recomendações dos responsáveis pelo projeto, para colaborar com o projeto precisamos primeiro criar um *fork* do mesmo, isto é, uma cópia pública do projeto, armazenada no próprio GitHub.
   1. Se você não tem, crie uma conta no GitHub ([https://github.com](https://github.com/pauloborba/teachingassistant))

https://github.com/mcbsf

* 1. Vá no site do projeto (<https://github.com/pauloborba/teachingassistant>) e solicite a criação de um *fork* (ver opção no canto direito superior da página acima)

OK

* 1. Confirme a criação e endereço do *fork*, a cópia criada do repositório

https://github.com/mcbsf/teachingassistant

* 1. Compare o conteúdo do *fork* com o conteúdo do projeto original. Eles têm o mesmo conteúdo?

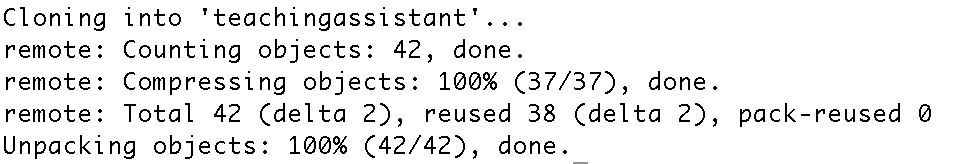
sim

* 1. Os responsáveis pelo projeto não permitem que você envie contribuições diretamente para o repositório principal; você terá primeiro que enviar contribuições para o *fork* e depois solicitar que as mesmas sejam analisadas e integradas ao repositório principal. Outros projetos podem ter formas diferentes de colaboração; com o SGM, temos que seguir esse procedimento.

1. Para começar a efetivamente contribuir com o SGM, precisamos copiar o conteúdo do *fork*, que está nos servidores do GitHub, para a máquina na qual vamos trabalhar. Para isso, precisamos criar um *clone* do *fork*.
   1. Na linha de comandos da sua máquina, no diretório onde você deseja que o projeto seja armazenado, digite

**git clone <URL\_DO\_SEU\_FORK>**

OK

* 1. Observe uma mensagem como abaixo 

OK

* 1. Liste e inspecione o conteúdo do diretório, comparando com o conteúdo do projeto original. Temos agora o mesmo conteúdo tanto no repositório principal quanto no *fork* do GitHub e no *clone* da sua máquina?

Sim

1. Agora podemos efetivamente contribuir com o projeto, começando pelos requisitos.
   1. Dentro do diretório **teachingassistant**, crie o diretório **features**

OK

* 1. Nesse diretório criado, crie o arquivo **AutoAvaliacao.feature** e digite nele a feature que você especificou no roteiro de requisitos

OK

* 1. Para que o git passe a gerenciar o histórico de mudanças nos arquivos desse novo diretório, digite

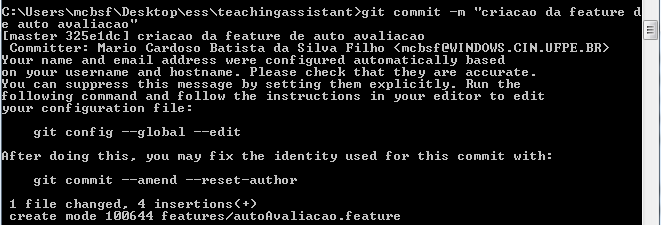
**git add features**

OK

* 1. Para registrar essa mudança, criaremos a primeira versão desse arquivo digitando

**git commit -m "criação da feature de auto avaliação"**

Isso cria, e armazena no histórico do projeto, um *commit*, ou seja um conjunto de versões dos arquivos modificados (nesse caso, só o arquivo .feature criado mais acima). Caso mais adiante seja feita uma mudança indesejada a um arquivo, teremos disponível a versão desse commit como referência. Qual a mensagem retornada pela execução desse comando?



* 1. Note que o argumento usado acima serve para registrar a mensagem do commit, que deve retratar claramente, de forma precisa e resumida, a mudança feita.
     1. Veja um [péssimo exemplo](https://github.com/npi-ufc-qxd/gpa-pesquisa/commit/f57931b9c549e9d7df75be41a2bd6773f06414fb) de mensagem
     2. Observe no seu *fork* as outras mensagens de *commit*, que podem servir de exemplo para as suas mensagens
     3. Veja também o histórico de *commits* na sua máquina digitando

**git log --graph**

Observe informações como o identificador do commit, autor e data de criação. Os autores dos *commits* são os mesmos? Por que?

Não, pois eu fiz um commit, então tem os de paulo borba e os meus

1. Digite agora, no arquivo **AutoAvaliacao.feature**, os cenários de preenchimento bem sucedido da auto-avaliação.
   1. Verifique a situação do seu repositório local, em termos de arquivos modificados e diferença em relação ao seu *fork* (origin/master), digitando

**git status**

OK

* 1. Verifique quais as mudanças em relação ao último commit digitando

**git diff**

Como esse comando indica as adições de linhas? Modifique uma das linhas do arquivo **AutoAvaliacao.feature**. Execute novamente o comando diff. Como a mudança é indicada pelo mesmo?

Dizendo quantos commits o branch local está a frente, além de indicar com linhas verdes o que falta dar commit

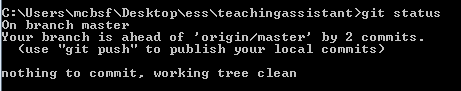
* 1. Registre essa mudança criando uma nova versão dos arquivos modificados:

**git commit -a -m "criação dos cenários de preenchimento bem sucedido da auto-avaliação"**

O "-a" adiciona os arquivos modificados, não precisando chamar o comando add explicitamente.

OK

1. Repita passos similares ao do item anterior para registrar os cenários de preenchimento mal sucedido da auto-avaliação. É melhor ter *commits* separados para contribuições que têm propósitos diferentes. Assim podemos ter *commits* frequentes, cada *commit* com pequenas contribuições, com foco bem definido, claramente refletidas pela mensagem do *commit*.
   1. Veja agora como está o histórico de *commits* na sua máquina



* 1. Analisando o histórico, reflita sobre quais as vantagens da abordagem (de pequenos e frequentes *commits*) discutida acima? Quais as desvantagens?

Torna o projeto modularizado, deixando-o mais organizado e de fácil entendimento. (FALTA COMPLETAR)\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1. Assumindo que essa parte da *feature* foi concluída, ou precisa ser revisada ou analisada por outras pessoas (*stakeholders*, outros membros da equipe, etc.), temos que enviar para o seu *fork* as modificações feitas até o momento.
   1. Como você tem autorização de escrita no seu *fork* (origin), digite

**git push origin master**

OK

* 1. No site do seu *fork*, verifique agora o histórico de *commits*. Ele contém o mesmo histórico do repositório da sua máquina? O conteúdo dos arquivos é o mesmo no fork e no repositório local?

Sim

* 1. Clique em dois commits quaisquer do histórico do seu *fork*. Reflita sobre quais as vantagens da abordagem de pequenos e frequentes *commits*? Há desvantagens significativas? Qual a dificuldade que você imagina que teria para revisar um *commit* grande, sem um foco claro, e com uma mensagem pouco representativa?

Algumas das vantagens são uma explicação mais detalhada e direcionada de como o código está funcionando. Em contra partida, pode vir a ter commits sobre os mesmos assuntos(exemplo: commits sobre correção de erros bestas). Uma das dificuldades de revisar um commit grande é falta de especificidade, exige um conhecimento maior do código

1. Agora vamos trabalhar em uma parte experimental da feature de auto avaliação, a visualização de auto-avaliações discrepantes. Ainda não é certo que essa parte será integrada à próxima versão do produto que será entregue ao cliente. Assim é melhor começar a trabalhar nela em uma linha de desenvolvimento alternativa e independente, um *branch*, para evitar impacto na linha principal (*branch* **master**) de desenvolvimento.
   1. Crie um *branch* chamado **discrepantes** digitando

**git branch discrepantes**

OK

* 1. Visualize os *branches* existentes digitando

**git branch**

OK

* 1. Note o asterisco ao lado de **master**, o que indica que ele é o *branch* atual. Qualquer *commit* criado será adicionado a esse *branch*. Como queremos trabalhar no novo *branch*, temos que torná-lo o *branch* atual digitando

**git checkout discrepantes**

**OK**

* 1. Adicione ao arquivo **AutoAvaliacao.feature** um cenário de quando não há discrepância, e registre a mudança criando um commit.

**OK**

* 1. Enquanto você estava trabalhando no novo cenário, os stakeholders analisaram os cenários que você tinha publicado no seu *fork*, e notaram que o cenário de falha (assumindo que esse é o último cenário do arquivo) precisa de um passo adicional nos resultados esperados (*Then*).
     1. Mude para o *branch* principal, adicione o passo solicitado no final do cenário, registre a mudança, e a envie para o seu *fork*.

**OK**

* + 1. O que teria acontecido se você estivesse trabalhando no novo cenário diretamente no branch **master** ao invés de em **discrepantes?** O arquivo atualizado e enviado para o usuário conteria algo que não deveria conter? Por que?

Sim, conteria o cenário que eu ainda estou trabalhando, que pode não estar funcionando ainda

* + 1. Visualize o histórico de mudanças. Mude para o branch **discrepantes** evisualize o histórico de mudanças. Há diferenças entre os dois históricos de mudanças? Por que?

Não. pois a alteração foi feita em outro branch

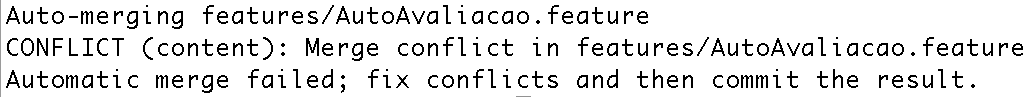
* 1. Adicione no final do arquivo **AutoAvaliacao.feature** um cenário de quando há discrepância, e registre a mudança criando um *commit*

OK

* 1. Tendo finalizado com sucesso essa nova parte da *feature* de auto-avaliação, e os *stakeholders* tendo optado pela sua incorporação à próxima versão do produto, integre as modificações do *branch* **discrepantes** ao *branch* **master**. Digite

**git checkout master**

**git merge discrepantes**

* 1. Você deve ter recebido uma mensagem de falha da integração, algo como

O que você acha que aconteceu? Por que?

Falha ao juntar 2 branches com alterações na mesma parte do código

* 1. Abra o arquivo **AutoAvaliacao.feature.** O que há de diferente nele? Qual o conteúdo final que você esperaria para esse arquivo?

Tem indicadores de onde estão os erros

* 1. Altere o arquivo para que ele contenha o conteúdo final que você esperaria, e registre a mudança criando um novo commit.

OK

* 1. Visualize agora o histórico de mudanças. Qual a diferença para o histórico anterior? Note que asteriscos representam *commits* de merge ou não. As barras representam o que?

Tem indicadores de quais branches estão sendo comittados ou sofreram merge

* 1. Envie as mudanças para o seu *fork*, e no site do *fork* analise o histórico de *commits* e *branches*.

OK

1. Tendo chegada agora uma solicitação de mudança mais significativa no cenário de visualização em que há discrepância, é interessante criar um *branch* específico para corrigir este erro. Assim, crie o *branch* **fixDiscrepancia**, ajuste o cenário, envie para revisão no seu *fork,* e assumindo que os stakeholders estão agora satisfeitos, integre a correção do erro ao *branch* principal.
   1. A integração foi feita com sucesso, sem conflito? Por que?

Sim, não tinha nada sobrescrevendo algo

* 1. Envie as mudanças para o seu *fork*

OK

* 1. Visualize as mudanças localmente e no GitHub. A história está registrada de acordo com o que você esperava?

Sim

* 1. Para que o branch **discrepantes** também seja visualizado no seu *fork*, digite

**git push origin discrepantes**

Analise agora o conteúdo do seu *fork* no GitHub. Qual a diferença em relação ao que você conseguia ver antes no GitHub? Há novos *commits*? Por que? Há novos *branches*?

Agora posso visualizar o branch de discrepancia, com seus respectivos commits

* 1. Se você não tivesse criado um branch para corrigir o erro, e, após ter feito progresso e realizado alguns commits, não tivesse conseguido finalizar a correção do erro a tempo para o lançamento da nova versão do produto, o que aconteceria?

Eu não poderia subir uma versão “bugada”

1. Para solicitar que as mudanças enviadas no seu *fork* sejam incorporadas ao repositório principal, siga as seguintes recomendações:
   1. Verifique o *status* do seu repositório local

OK

* 1. Se houver mudanças no repositório principal, incorpore-as ao seu repositório local digitando

**git pull**

Que comando você usaria para baixar mudanças do repositório de um colega, ao invés do repositório principal? Se você precisar baixar atualizações do seu colega com frequência, é interessante definir o repositório dele como um *remote*; que comando do git você usaria para fazer isso?

git remote add “rep local” “nome do rep remoto”

* 1. Após integrar as mudanças do repositório principal, e verificar que está tudo OK, solicite no site do seu *fork* no GitHub a criação de um *pull request* (opção *New Pull Request*).

OK

1. Ao invés de **git add features**, você poderia ter trocado o nome *features* por um ponto, que representa o diretório corrente. Em que situações os dois comandos teriam o mesmo efeito? Em que situações eles teriam efeito diferentes?
2. É interessante que todos os arquivos do projeto sejam mantidos sob controle de versão? Por que? Para que serve o arquivo **.gitignore**? Dê um exemplo de uma linha que você acha que seria importante adicionar a esse arquivo.

Sim, para ter maior controle do fluxo e gerenciar o projeto com mais eficiência. O .gitignore serve para o programa ignorar a linha em que isso está escrito, para não dar erro de compilação.

1. Para identificar quando, e por quem, foi feita cada mudança no arquivo **AutoAvaliacao.feature**, digite

**git blame AutoAvaliacao.feature**

Analise a saída gerada e indique o que representam as várias informações que aparecem em uma linha qualquer da saída.

Indica quando e quem fez cada alteração

1. Todos esses comandos do git têm várias opções de comportamento que podem ser passadas como parâmetros para que o comando funcione de forma diferente. Verifique quais são as opções do **git blame** digitando

**git blame -h**

Qualo comportamento obtido com as opções **-L** e **--line-porcelain**?

“-l” mostra commits longos, enquanto “--line-porcelain” mostra informações mais detalhadas

1. Ao invés de ter usado **git merge discrepantes** no Item 7**,** para incorporar as modificações de **discrepantes** ao **master**, alguns projetos exigem antes uma reorganização da história do projeto para que a história do repositório principal seja mais limpa e linear. Para entender como isso funciona, siga as recomendações abaixo:
   1. Crie dois *commits* em **discrepantes** e um no **master**, e visualize o histórico do seu projeto local

OK

* 1. Antes de integrar os dois *branches*, mude a base dos novos commits adicionados a **discrepantes** digitando

**git rebase master**

no branch **discrepantes**. Isso faz com que as modificações feitas em **discrepantes** sejam agora representadas por *commits* similares (não os mesmos) em **master**.

OK

* 1. Finalize o processo indo para **master** e digitando

**git merge**

OK

* 1. Visualize o histórico do seu projeto local. O que aconteceu de diferente em relação a um simples processo direto de merge, como feito no Item 7?

O rebase muda toda a linha do branch, tornando mais ‘linear’

* 1. Integre as mudanças ao seu *fork*.

OK

1. Faça uma modificação no arquivo **AutoAvaliacao.feature** e registre-a, por exemplo, com uma mensagem como "novo cenário de visualização de percentual de discrepância". Se logo em seguida notamos que é necessário um pequeno ajuste nessa modificação, devemos fazer o ajuste e registrá-lo com uma mensagem como "pequeno ajuste no novo cenário de visualização de percentual de discrepância".
   1. Faça esse ajuste

OK

* 1. Você acha que muitas sequências de *commits* desse tipo podem comprometer o entendimento do repositório principal e dificultar a revisão das contribuições? Por que?

Sim. excesso de informação a respeito de coisas pontuais, além de correção de bugs(desnecessário).

* 1. Para simplificar a história de desenvolvimento, digite

**git rebase -i HEAD~2**

No arquivo apresentado, digite o comando **squash** no local do **pick** do commit que você quer que seja unificado ao outro. Ajuste a mensagem do commit final para "novo cenário de visualização de percentual de discrepância".

* 1. Analise o histórico do seu repositório. O que aconteceu? Qual o conteúdo do arquivo no commit final?
  2. Integre as mudanças ao seu *fork*.
  3. Se você quisesse unificar 3, ao invés de apenas 2 *commits*, o que você mudaria nas ações feitas no Item c?
  4. Se você tivesse unificado *commits* que já tivessem sido enviados ao repositório principal, o que iria acontecer caso algum colega já tivesse baixado esses commits e integrados ao repositório dele?

1. DESCONSIDERAR DAQUI EM DIANTE! V[eja](https://git-scm.com/book/en/v2/Getting-Started-First-Time-Git-Setup#Your-Identity) como configurar seu usuário git via linha de comando, e certifique-se de que você informou corretamente ao git as informações sobre autor (use o seu login do github) e email (use o seu email do CIn)

### git checkout -- <file>, stage

### Forma de deletar o branch atual e todas as alterações dele via linha de comando

|  |
| --- |
| git reset --hard HEAD  git checkout master  git branch -D <branch you want to delete> |

### Corrigindo arquivos inválidos enviados

|  |
| --- |
| O correto, é não dar commit em arquivos gerados ou de configuração específica do ambiente, como .class, .project. idea, .classpath.... Mas caso algum arquivo desses seja erroneamente comitado e o pull request enviado, o mais fácil a se fazer é adicionar esses arquivos no gitignore e dar outro commit. Dessa forma, os arquivos inválidos não vão aparecer no pull request. Esse link: <https://help.github.com/articles/ignoring-files> explica como fazer isso. |

1. Remotes

## **Download and install p4merge**

* Download the perforce visual tool suite from here: <http://www.perforce.com/perforce/downloads/index.html>
* Copy only the p4merge.app file into your /Applications/ directory

## **Setup p4merge as a visual mergetool**

$ git config --global merge.tool p4mergetool  
$ git config --global mergetool.p4mergetool.cmd \  
"/Applications/p4merge.app/Contents/Resources/launchp4merge \$PWD/\$BASE \$PWD/\$REMOTE \$PWD/\$LOCAL \$PWD/\$MERGED"  
$ git config --global mergetool.p4mergetool.trustExitCode false  
$ git config --global mergetool.keepBackup false

## **Setup p4merge as a visual diff tool**

$ git config --global diff.tool p4mergetool  
$ git config --global difftool.p4mergetool.cmd \  
"/Applications/p4merge.app/Contents/Resources/launchp4merge \$LOCAL \$REMOTE"

## **Using p4merge to resolve conflicts**

When you run into a conflict when merging simply run:

$ git mergetool

* You will be prompted to run "p4mergetool", hit enter and the visual merge editor will launch.
* Using the merge tool you can resolve the conflict and then save the file.
* After you exit the merge tool take a look back at your terminal. You will be asked if the merge was successful, choose yes if all is well or no if you need to start over.
* This prompting is happening because the "trustExitCode" option is turned off. Personally I always like to be asked, but you can have git just trust the exit code from the merge tool.

1. git checkout 9135dae1092e248f64e062770f6829173020f939 git stash
2. Descartar mudancas, analisar versao anterior, voltar para versao anterior, etc.
3. Abortar merge, deletar branch
4. Paper de daniel, pegar tarefas de la