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

## Paulo Borba

## Centro de Informática

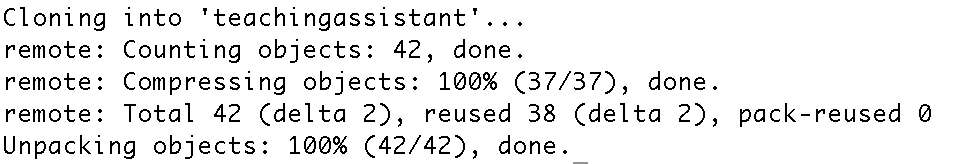
## Universidade Federal de Pernambuco

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.

## Aula 1

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))
   2. 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 que está no endereço acima)
   3. Confirme a criação e endereço do *fork*, a cópia criada do repositório. Cole abaixo o endereço.
   4. Compare o conteúdo do *fork* com o conteúdo do projeto original. Eles têm o mesmo conteúdo?
   5. 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.
2. 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 (se você não é familiar com *shells*, recomendo fortemente que estude agora; este [tutorial](https://github.com/chrisparnin/EngineeringBasics/blob/master/Shells.md#shells) traz um bom resumo), no diretório onde você deseja que o projeto seja armazenado, digite

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

* 1. Observe uma mensagem como a abaixo 
  2. 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?
  3. Analise a documentação do git ([livro](https://github.com/progit/progit2/releases/download/2.1.161/progit.pdf), [manual de referência](https://git-scm.com/docs)) e descubra o propósito do comando **git init**. Qual a diferença dele em relação ao **git clone**? Em que situações você usaria o **git init?**

1. Agora podemos efetivamente contribuir com o projeto, começando pelos requisitos.
   1. Dentro do diretório **teachingassistant**, crie o diretório **features**
   2. Nesse diretório criado, crie o arquivo **AutoAvaliacao.feature** e digite nele a feature que você especificou no roteiro de requisitos
   3. Para que o git passe a gerenciar o histórico de mudanças nos arquivos desse novo diretório, digite

**git add features**

* 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? Qual a diferença entre os comandos **add** e **commit**?

* 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. Observe no seu *fork* as outras mensagens de *commit*, que podem servir de exemplo para as suas mensagens. Cole abaixo uma mensagem nesse *fork* que você considera adequada, e outra que considera inadequada.
     2. Veja também o histórico de *commits* na sua máquina digitando

**git log --graph --branches**

Observe informações como o identificador do commit, autor e data de criação. Os autores dos *commits* são os mesmos? Por que? No manual do git, veja as opções de linha de comando e explique qual o comportamento do comando **log** caso as opções acima não tivessem sido usadas. Indique uma opção da lista que você ache que poderia ser útil para você.

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, diferença em relação ao seu *fork* (origin/master), etc. digitando o comando abaixo. Cole em seguida uma imagem com o resultado e explique-o.

**git status**

* 1. Verifique quais as mudanças feitas em seu diretório de trabalho, 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?

* 1. Analise a documentação do git e indique abaixo como o comando **diff** pode ser usado para comparar o conteúdo de dois *commits* anteriores ao atual.
  2. Registre a última mudança feita criando uma nova versão dos arquivos modificados:

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

Veja na documentação do git que o "-a" adiciona todos os arquivos modificados e previamente adicionados (via comando **add**), não precisando chamar o comando **add** repetidamente.

* 1. Modifique uma das linhas do arquivo **AutoAvaliacao.feature** e dê *commit,* com nova mensagem e agora sem o "-a". A sua modificação foi armazenada pelo *commit*? Por que?

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. Cole abaixo a imagem.
   2. Analisando o histórico, reflita sobre quais as vantagens da abordagem (de pequenos e frequentes *commits*) discutida acima? Quais as desvantagens?
2. 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 o comando a seguir. Consulte a documentação do git para indicar o significado de cada uma das partes do comando abaixo.

**git push origin master**

* 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?
  2. Clique em dois commits quaisquer do histórico do seu *fork*. Considerando que a abordagem de *commits* pequenos e frequentes não tivesse sido seguida, qual a dificuldade que você imagina que teria para revisar um *commit* grande, sem um foco claro, e com uma mensagem pouco representativa?

## Aula 2

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**

* 1. Visualize os *branches* existentes digitando

**git branch**

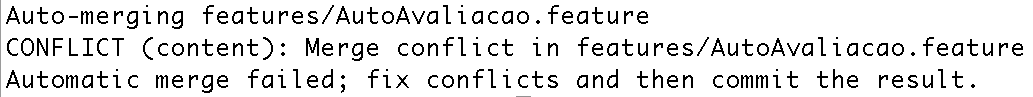
* 1. Note o asterisco ao lado de **master**, o que indica que ele é o *branch* atual. Qualquer *commit* criado será adicionado a esse *branch*. Como queremos trabalhar no novo *branch*, temos que torná-lo o *branch* atual digitando

**git checkout discrepantes**

* 1. Adicione ao arquivo **AutoAvaliacao.feature** um cenário de quando não há discrepância, e registre a mudança criando um *commit*.
  2. 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** (atenção às localizações exatas das mudanças explicitadas nesta questão, já que isso pode fazer diferença no resultado), registre a mudança, e a envie para o seu *fork*.
     2. 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? O que? Por que?
     3. Visualize o histórico de mudanças. Mude para o *branch* **discrepantes** evisualize o histórico de mudanças. Há diferenças entre os dois históricos de mudanças? Quais? Por que?
  3. Adicione **no final do arquivo** **AutoAvaliacao.feature** um cenário de quando há discrepância, e registre a mudança criando um *commit*.
  4. 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?

* 1. Abra o arquivo **AutoAvaliacao.feature.** O que há de diferente nele em relação a uma arquivo de *features* típico? Qual o conteúdo final que você esperaria para esse arquivo?
  2. Altere o arquivo para que ele contenha o conteúdo final que você esperaria, e registre a mudança criando um novo *commit*.
  3. 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?
  4. Envie as mudanças para o seu *fork*, e no site do *fork* analise o histórico de *commits* e *branches*.

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 (adicionando duas linhas quaisquer), envie para revisão no seu *fork* (veja o comando do item d abaixo)*,* 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?
   2. Envie as mudanças para o seu *fork*
   3. Visualize as mudanças localmente e no GitHub. A história está registrada de acordo com o que você esperava?
   4. 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*?

* 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?

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
   2. 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? Indique qual a diferença entre o comando **pull** e o **fetch**? Em que situação você usaria o segundo?

* 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*).

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.
3. Opcionalmente, 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.

1. Opcionalmente, 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**?

## Aula 3

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
   2. 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**. Como você pode constatar que os *commits* são similares?

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

**git merge discrepantes**

para que os dois branches apontem para o mesmo commit

* 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?
  2. Integre as mudanças ao seu *fork*.

1. Opcionalmente, 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
   2. Você acha que muitas sequências de *commits* desse tipo podem comprometer o entendimento do repositório principal e dificultar a revisão das contribuições? Por que?
   3. 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". Por default, o editor apresentado é o vi. Se for esse o caso, use "esc" (sempre usado antes de invocar os comandos do editor) e depois ":wq" para gravar as modificações e sair do sistema. Para deletar um caracter, digite "x". Para deletar uma linha inteira, use "dd". Para inserir um caracter, use "i".

* 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?
  5. Ao invés de criar um novo *commit* e usar o **squash**, como recomendado acima, você poderia ter usado o comando **git commit --amend**? Indique o que esse comando pode fazer.

1. Estando no *branch* **discrepantes**, e tendo feito uma mudança no arquivo **AutoAvaliacao.feature**, gravado o arquivo,sem ter dado ainda *commit*, mude para o *branch* **master**. Após essa operação, qual o conteúdo do arquivo **AutoAvaliacao.feature** no seu diretório de trabalho local? Esse conteúdo corresponde ao que você esperava? Por que? O que teria acontecido se, no *branch* **master**, o arquivo **AutoAvaliacao.feature** tivesse sido também modificado na mesma área em que ele foi modificado quando você estava no *branch* **discrepantes**? Os comandos **git stash** poderiam ajudar nessa situação? Que comandos exatamente? Como?
2. Opcionalmente, tendo terminado o trabalho no *branch* **discrepantes**, você poderia apagá-lo?Que comando você usaria para isso? Qual o efeito desse comando (ele apaga exatamente o que? ponteiros? commits?)?
3. Opcionalmente, use o comando **git tag** para marcar o conteúdo de **master** como sendo a primeira versão da sua especificação de requisitos. Qual a vantagem disso?
4. Supondo que você tenha feito modificações em vários arquivos no seu diretório de trabalho local, e tenha chegado a conclusão de que quer descartar essas modificações. Como você poderia usar o comando **git reset** para fazer isso? Com que parâmetros você chamaria esse comando? Qual a diferença entre esse comando e o **git checkout**?
5. Opcionalmente, indique, listando os comandos específicos que você digitaria, como o comando **git config** pode ser usado para configurar seu usuário git (informações sobre autor e e-mail). Que outros tipos de informações podem ser configuradas com o comando **git config**?

1. Opcionalmente, explique a utilidade dos comandos mv, rm, grep e bisect.