Laboratórios de Informática I
2023/2024

Licenciatura em Engenharia Informática

Ficha 7
Sistemas de Controlo de Versões

O desenvolvimento de *software* é cada vez mais complexo, e obriga a que uma equipa de programadores possa desenvolver uma mesma aplicação ao mesmo tempo, sem se preocuparem com os detalhes do que outros membros dessa mesma equipa estejam a fazer. Alterações concorrentes (realizadas por diferentes pessoas ao mesmo tempo) podem provocar conflitos quando várias pessoas editam o mesmo bocado de código.

Além disso, não nos devemos esquecer que algumas alterações a um programa, no sentido de corrigir ou introduzir alguma funcionalidade, podem elas mesmas conter erros, e pode por isso ser necessário repor uma versão prévia da aplicação, anterior a essa alteração.

Para colmatar estes problemas são usados sistemas de controlo de versões.

1 Panorama nos Sistemas de Controlo de Versões

Existe um grande conjunto de sistemas que permitem o desenvolvimento cooperativo de *software*. Todos eles apresentam diferentes funcionalidades mas os seus principais objetivos são exatamente os mesmos.

Habitualmente divide-se este conjunto em dois, um conjunto de sistemas denominados de *centralizados*, e um outro de sistemas *distribuídos*:

- Sistemas de controlo de versões centralizados:
  - Concurrent Versions System (CVS): `http://www.nongnu.org/cvs/`;
  - Subversion (SVN): `https://subversion.apache.org/`;
- Sistemas de controlo de versões distribuídos:
  - Git: `http://git-scm.com/`;
  - Mercurial (hg): `http://mercurial.selenic.com/`;
  - Bazaar (bzr): `http://bazaar.canonical.com/en/`;

Estes são apenas alguns exemplos dos mais usados. A grande diferença entre os centralizados e os distribuídos é que, nos centralizados existe um repositório, denominado de servidor, que armazena, a todo o momento, a versão mais recente do código fonte. Por sua vez, nos distribuídos, cada utilizador tem a sua própria cópia do repositório, que podem divergir, havendo posteriormente métodos para juntar repositórios distintos.

2 Instalação do Git

Na disciplina de Laboratórios de Informática I será utilizado o sistema `Git`. Para o instalar siga as instruções em `https://git-scm.com/downloads`.

3 Configurar o git

Para configurar o `Git` ao nível do sistema deve indicar o nome e o email que ficarão associados às actividades realizadas no repositório.

```
$ git config --global user.name "a999999"
```

Configura o nome que irá ficar associado aos git commits, `a999999` neste exemplo.

```
$ git config --global user.email "a999999@alunos.uminho.pt"
```

Configura o email que irá ficar associado aos git commits, `a999999@alunos.uminho.pt` neste exemplo.

Poderá consultar a configuração actual com:

```
git config --list
```

4 Uso do Git

Os ficheiros de uma directoria de trabalho controlada pelo `Git` podem estar no estado *tracked* (fazem parte do repositório ou estão na área de preparação (*staged area*) para que possam ser incluidos no repositório) ou *untracked* (detectadas pelo `Git`, mas desconhecidos do repositório). Os ficheiros no estado *tracked* podem estar estar *unmodified* (actualizados no repositório), *modified* (modificados desde a última actualização do repositório), or *staged* (marcados como preparados para inclusão no repositório). Para actualizar o repositório, os ficheiros no estado *tracked - staged* terão de ser convertidos em *tracked - unmodified*. Vamos de seguida analisar como o `Git` gere este ciclo de transformações.

4.1 Init

Vamos criar um repositório local chamado `AulasLI1` para experimentar alguns comandos do `Git`.

```
$ git init AulasLI1
```

Verifique que foi criada na directoria actual a subdirectoria `AulasLI1/`.

4.2 Adição de novas directorias e ficheiros

O passo seguinte corresponde a adicionar novos ficheiros ou pastas que queiramos armazenar no repositório. Como exemplo, vamos adicionar um ficheiro `README.md` ao repositório. Crie na directoria `AulasLI1` um ficheiro chamado `README.md`, e escreva no ficheiro o seu nome completo. Este ficheiro ainda não faz parte do repositório (está no estado *untracked*).

Um comando extremamente simples, mas bastante útil, designado por `status`, permite ver o estado atual do repositório local (sem realizar qualquer ligação ao servidor)

```
$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        README.md

nothing added to commit but untracked files present (use "git add" to track)
```

Isto indica que o `Git` detecta o ficheiro `README.md`, mas não sabe nada sobre ele.

Para adicionar o ficheiro `README.md` ao repositório terá de começar por executar o comando:

```
$ git add README.md
```

O ficheiro foi adicionado à área de preparação para que possa ser incluído no repositório, mas ainda não faz parte do repositório controlado pelo `Git`. Como veremos, tal só acontecerá quando for executado o comando *commit*.

Se executar:

```
$ git status
On branch master
No commits yet
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   README.md
```

a mensagem indica que há um novo ficheiro pronto a ser inserido no repositório. Indica também forma de o remover da *staged area*.

Sempre que realizar alterações a um ficheiro e as quiser registar, terá de dar essa informação ao `Git`. O mesmo comando também é usado para adicionar novos ficheiros ao repositório. Assim, depois de terminar as alterações a um ficheiro (novo ou não), deve executar o comando `git add`.

Uma boa prática de desenvolvimento é adquirir o hábito de gerir todo o código que programar para o projeto através do sistema de versões.

Tarefas

1. Crie um novo ficheiro chamado `exemplo.txt` com um qualquer conteúdo.
2. Altere o ficheiro `README.md` adicionando o número de aluno.
3. Execute o comando `git status` e verifique que obtém:

   ```
   $ git status
   On branch master
   No commits yet
   Changes to be committed:
   ```

```
  (use "git rm --cached <file>..." to unstage)
        new file:   README.md
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.md
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        exemplo.txt
```

Isto indica que o `Git` não sabe nada sobre o ficheiro `exemplo.txt` e que portanto o irá ignorar em qualquer comando executado. É também assinalado que o ficheiro `README.md` foi modificado e que é necessário fazer *add* para actualizar a versão pronta a submeter ao repositório.

4. Faça agora:

```
$ git add README.md
$ git add exemplo.txt
```

5. Volte a executar o comando status devendo obter:

```
$ git status
On branch master
No commits yet
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   README.md
        new file:   exemplo.txt
```

indicando que ambos os ficheiros estão agora sob controlo do Git.

4.3 Commit

Para registar as alterações e os novos ficheiros ou pastas no repositório local é necessário realizar um processo designado por *commit* [1]. Isto poderá ser feito através do comando `git commit`.

```
$ git commit -m "Adicionados os ficheiros README.md e exemplo.txt"
[master (root-commit) 0ddb6d3] Adicionados os ficheiros README.md, exemplo.txt
 2 files changed, 3 insertions(+)
 create mode 100644 README.md
 create mode 100644 exemplo.txt
```

No comando *commit* executado foi adicionada uma opção (`-m`) que é usada para incluir uma mensagem explicativa das alterações que foram introduzidas ao repositório. Se escrever apenas `git commit` será enviado para um editor de texto (e.g. Vim) onde terá de escrever a mensagem a associar ao *commit*.

[1] A executar após o add

É boa prática adicionar uma mensagem clara em cada *commit*.

Deve realizar um *commit* sempre que faça alterações ao seu código que em conjunto formem uma modificação coerente. Os seguintes exemplos podem originar novos commits:

- adicionar uma nova função;
- adicionar um nova funcionalidade;
- corrigir um bug;

Note que depois do *commit* o código está no repositório, mas apenas na sua cópia local.

4.4 Diff

O comando `git diff` mostra as diferenças linha a linha dos ficheiros alterados (antes do add).

Altere o ficheiro `README.md` acrescentando o turno de que faz parte e alterando o número de aluno (troque o "a"por "A"ou o inverso).
Execute:

```
$ 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.md
no changes added to commit (use "git add" and/or "git commit -a")
```

e pode comprovar que há alterações identificadas.
Execute agora:

```
$ git diff
diff --git a/README.md b/README.md
index 352a4e4..7159b7a 100644
--- a/README.md
+++ b/README.md
@@ -1,2 +1,3 @@
 Aluno: Olga Pacheco
-Número: a99999
+Número: A99999
+Turno: PL10
```

para identificar as diferenças entre as versões do ficheiro `README.md` (repare nas cores, no que foi substituido e acrescentado).

Se executar novamente `git add README.md` e de seguida `git diff`, verificará que não há agora diferenças identificadas. Se fizer `git status`, verificará que o ficheiro actualizado `README.md` está pronto a ser submetido ao repositório, o que acontecerá logo que faça `git commit -m "alteração README.md"`.

4.5 Log

O comando `git log` lista o histórico de versões. Depois das alterações acima mencionadas, obtemos:

```
$ git log
commit c5277f4112d6663cc8903f79fac8c540d72b705e (HEAD -> master)
Author: [omp] <omp@di.uminho.pt>
Date:   Sun Oct 24 17:37:37 2021 +0100

    alteração de README.md

commit 0ddb6d30a08045670abcdc5f83f6490f0003cec3
Author: [omp] <omp@di.uminho.pt>
Date:   Sun Oct 24 17:22:02 2021 +0100

    Adicionados os ficheiros README.md e exemplo.txt
```

Há uma grande variedade de opções para este comando. Para mais detalhes pode consultar: `https://git-scm.com/book/en/v2/Git-Basics-Viewing-the-Commit-History`.

4.6 Remove

Vamos agora ver como remover ficheiros do repositório. Isto poderá ser feito através do comando `git rm`, seguido do nome do ficheiro. Para remover o ficheiro `exemplo.txt` faz-se:

```
    $ git rm exemplo.txt
rm 'exemplo.txt'
```

Tal como na operação *add*, temos que fazer *commit* para que um ficheiro marcado para ser apagado seja efetivamente apagado no repositório local.

As mensagens do `Git` dão informações detalhadas sobre formas de remover ou reverter alterações a ficheiros, nas diferentes situações.

Tarefas

Analise a sequência de comandos seguinte:

```
$ git rm exemplo.txt
rm 'exemplo.txt'

$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
        deleted:    exemplo.txt

$ git commit -m "remoção de exemplo.txt"
[master c359369] remoção de exemplo.txt
```

```
 1 file changed, 1 deletion(-)
 delete mode 100644 exemplo.txt

$ git status
On branch master
nothing to commit, working tree clean

$ git log
commit c3593696879c7ec6afe0b72480507452081a5475 (HEAD -> master)
Author: [omp] <omp@di.uminho.pt>
Date:   Sun Oct 24 17:42:46 2021 +0100

    remoção de exemplo.txt

commit c5277f4112d6663cc8903f79fac8c540d72b705e
Author: [omp] <omp@di.uminho.pt>
Date:   Sun Oct 24 17:37:37 2021 +0100

    alteração de README.md

commit 0ddb6d30a08045670abcdc5f83f6490f0003cec3
Author: [omp] <omp@di.uminho.pt>
Date:   Sun Oct 24 17:22:02 2021 +0100

    Adicionados os ficheiros README.md e exemplo.txt
```

NOTA: Para esclarecer dúvidas sobre algum <comando> poderá escrever: `git help <comando>`.

5 GitLab, GitHub, etc.

Até ao momento utilizamos o `Git` para gerir um repositório local. Contúdo, o objectivo principal da utilização de sistemas de contolo de versões é suportar o desenvolvimento cooperativo de *software*. Nos sistemas distibuidos como o `Git`, cada utilizador tem a sua própria cópia do repositório, que podem divergir, havendo posteriormente métodos para juntar repositórios distintos.

Apesar da natureza distribuída do sistema de controlo de versões `Git`, a existência de um servidor central como ponto de referência do repositório torna a distribuição do código mais simples. Nesta secção iremos continuar este assunto, tendo em conta um tal servidor comum.

Serviços como o GitLab[2], GitHub[3], SourceHut[4], Bitbucket[5] entre outros, permitem que os utilizadores alojem um repositório *git* de forma pública (aberto a toda a Internet) ou privada (disponível apenas a certos outros utilizadores). Outra grande vantagem destes serviços é permitirem realizar tarefas de manutenção do repositório, tais como:

[2] `https://gitlab.com`
[3] `https://github.com`
[4] `https://sr.ht`
[5] `https://bitbucket.org/`

1. Disponibilizarem uma *Wiki* para fins de documentação;
2. Permitirem navegar nos vários *commits* do repositório, facilitando assim a consulta de como o código evoluiu;
3. Disponibilizarem um gestor de *bugs/issues*, onde podemos documentar problemas no código, discutí-los e, mais tarde, associar a sua resolução a um *commit*;
4. Disponibilizarem um interface de *code review* onde se torna possível estudar o código contribuído por um utilizador antes da sua integração.

5.1 Uso destes serviços

Antes de podermos usufruir de qualquer um destes serviços é necessário criar uma conta de utilizador. Note que o e-mail que utilizar neste registo deve coincidir com o e-mail institucional configurado no `Git` na secção anterior, por forma a que o registo dos commits seja devidamente acompanhado.

Criada uma conta num destes serviços, podemos agora criar um novo repositório `Git` de raíz, clonar um repositório já existente para a nossa máquina, ou criar um novo repositório a partir de outro (*fork*).

Tarefas

1. Crie ou use uma conta (com o seu e-mail institucional) no GitLab.
2. Vamos criar um repositório novo usando o painel de controlo do GitLab. Crie um projeto privado, identificado com o seu número de aluno `aXXXXXX`. Permita a inclusão no repositório do ficheiro `README.md`.

5.2 Clone

Anteriormente vimos como criar um repositório `Git` de raiz na nossa máquina. Vamos agora ver como clonar para a nossa máquina um repositório já existente .

O comando para tal é `git clone` seguido do endereço do repositório. Por exemplo, se quiser clonar o repositório individual criado na secção anterior deverá executar o seguinte comando, usando o endereço associado ao repositório no momento da criação:

```
$ git clone <endereço>
```

O resultado da execução deste comando é uma directoria, criada na directoria atual, com o nome do repositório (`aXXXXXX`). Esta directoria será a raiz do repositório e terá a mesma estrutura do repositório clonado do servidor, que neste caso tem apenas o ficheiro `README.md`.

Este comando `git clone` deverá ser efectuado **uma única vez**, sendo as alterações futuras ao repositório central geridas através de comandos como, por exemplo, `git push` e `git pull`.

5.3 Push

Para enviar as alterações e os novos ficheiros ou pastas para o servidor central é necessário realizar um processo designado por `push`. Isto poderá ser feito através do comando `git push`. Apenas após a execução deste comando o código estará acessível por todos os utilizadores do repositório.

Tarefas

1. Depois de clonar o repositório, altere o ficheiro `README.md` criado.
2. Registe a alteração no repositório, i.e. faça `add` (como vimos nas secções anteriores) seguido do `commit` desta alteração com a mensagem `"README.md alterado"`.
3. Faça *push* para o servidor, escrevendo `git push`.
4. Dirija-se ao repositório `aXXXXXX` no painel de controlo do GitLab e confirme que as alterações detectadas pelo GitLab estão de acordo com a alteração efectuada localmente.
5. Crie no repositório um ficheiro `exemplo.txt` com um texto à sua escolha. Actualize o repositório local. Actualize o repositório remoto. Verifique no GitLab as alterações efectuadas.

5.4 Pull

Sendo que o `Git` é especialmente útil no desenvolvimento cooperativo, vamos ver o que acontece quando um outro utilizador altera o repositório. Suponhamos então que um outro utilizador apagou o ficheiro `exemplo.txt` ou alterou o ficheiro `README.md`.

Um utilizador de `Git` deve atualizar a sua cópia local do repositório sempre que possível e no mínimo antes de iniciar uma sessão de trabalho, para que quaisquer alterações que tenham sido incluídas por outros programadores sejam propagadas do repositório central para a sua cópia local.

Este processo é feito através do comando `git pull`.

Tarefas

1. Apague o ficheiro `exemplo.txt` no repositório remoto (através do painel de controlo do GitLab).
2. Verifique que e ficheiro ainda se encontra no repositório local.
3. Faça agora `git pull`. Verifique que o repositório local foi actualizado e já não contém o ficheiro `exemplo.txt`.

Nem sempre os programadores estão a trabalhar em ficheiros distintos. Supondo que dois utilizadores alteraram o mesmo ficheiro em simultâneo, mas em zonas diferentes do ficheiro (por exemplo, cada um modificou apenas o seu nome no ficheiro `README.md`), e que o primeiro já fez *push* da sua alteração. O `Git` é capaz de lidar com a alteração concorrente sem problemas, e portanto, poderá ser feito o `push` destas últimas alterações.

Em algumas situações poderá acontecer que dois utilizadores tenham editado a mesma zona do ficheiro, e portanto, que o `git` não tenha conseguido juntar as duas versões. Nesta situação, ao realizar o `pull` terá de gerir o conflito manualmente.

Ao editar um ficheiro com conflito aparecerão marcas deste género:

```
codigo haskell muito bom
<<<<<<< HEAD
mais código
=======
mais código do outro utilizador
>>>>>>> branch-a
```

Isto indica a zona com conflito. Uma versão é o texto entre as marcas `<<<<<<<` e `=======` e a outra versão é o texto entre as marcas `=======` e `>>>>>>>`. Nesta situação é nosso dever remover as marcas (as linhas com `<<<<<<<` , `=======` e `>>>>>>>`) e optar por uma das partes (ou então, criar uma nova que resolva o conflito).

Depois de resolver um conflito o utilizador deverá indicar que o conflito foi resolvido:

```
$ git add README.md
$ git commit -m "conflito resolvido"
$ git push
```

5.5 Remove

Vamos agora ver como remover ficheiros do repositório central. Como vimos anteriormente isto poderá ser feito através do comando `git rm`, seguido do nome do ficheiro. Tal como na operação `add`, temos que fazer `commit` e `push` para que um ficheiro marcado para ser apagado seja efetivamente apagado no repositório central.

Note que embora o ficheiro tenha sido removido da directoria de trabalho, ele ficará guardado no servidor. Portanto se tal for necessário é possível reaver o ficheiro.

Note que para que o `git log` apresente a informação actualizada deverá sempre fazer `git pull` antes. Se existirem *commits* locais e existirem *commits* remotos que não existem localmente, devemos fazer `git pull --rebase` para que o nosso `commit` passe a ser o mais recente e não seja criado um *merge commit* desnecessáriamente.

6 Boas práticas

6.1 Branches

Quer tenha criado um novo repositório ou clonado para si um já existente, antes de começar a fazer contribuições é boa prática criar uma nova *branch*. Será nessa nova *branch* que irá registar os seus *commits* (contribuições).

O comando para criar uma nova *branch* a partir da actual e ficar posicionado na *branch* criada é:

```
git checkout -b <nome-da-branch>
```

Por exemplo, fazendo

```
$ git checkout -b Temp
Switched to a new branch 'Temp'
```

estaremos a criar uma nova *branch* denominada `Temp`. A partir daqui, deve fazer os seus commits como habitualmente faria. Desta forma, quando mais tarde quiser partilhar com os restantes utilizadores as suas contribuições, apenas tem de lhes indicar esta sua *branch*.

Pode consultar as *branches* definidas escrevendo: `git branch` .

6.2 Pull/Merge Request

Suponha agora que já terminou tudo o que pretendia fazer em relação à *branch* `Temp`, e quer integrar as suas alterações junto do repositório central.

O primeiro passo a fazer é enviar a sua *branch* para o repositório. Lembre-se que o *git* é distribuído, por isso enquanto não fizer este passo a sua *branch* existirá apenas na sua máquina. Para tal, basta-nos invocar o comando *push*. Quando queremos fazer *push* de uma *branch* que ainda não existe remotamente ou quando se cria uma localmente e nunca se fez push/pull, é preciso indicar o *remote* e o nome da *branch*. No exemplo acima o comando é:

```
$ git push origin Temp
```

Também é possível passar uma flag `--set-upstream` para que nos *commits* seguintes seja só fazer *push*. Fazendo:

```
git push --set-upstream origin Temp
```

os *pushes* seguintes seriam só `git push`.

Se tiver sido feito clone, o origin aponta para o repositório central (no nosso caso o repositório no GitLab).

Agora que a sua *branch* já existe no repositório central, poderá criar um *merge request* (*pull request* em alguns sistemas). Um *merge request* consiste num requerimento para integrar as contribuições de uma *branch* noutra, onde os demais utilizadores têm a oportunidade para inspeccionar o código antes de o aceitarem.

Para criar um *merge request*, deve dirigir-se ao site onde alojou o repositório e procurar por esta funcionalidade no painel de controlo do seu repositório. Ser-lhe-á questionado qual a *branch* (e.g. `Temp`) de origem e qual a *branch* de destino (e.g. `main`).

Tarefas

1. Crie uma nova branch local, denominada `NovoReadme`.
2. Edite o ficheiro `README.md` por forma a conter o seguinte texto

   ```
   # Aulas LI1
   Experiência com branch.
   Por: <nome> <mail>
   ```

3. Registe, i.e. faça `commit` desta alteração com a mensagem "README.md reescrito"(depois de fazer `add`).
4. Faça *push* da branch `NovoReadme` para o servidor.

5. Dirija-se ao painel de controlo do GitLab para o repositório `aXXXXXX`, e inicie um novo *Pull/Merge request*. Confirme que as alterações detectadas pelo GitLab estão de acordo, introduza uma mensagem descritiva e crie finalmente o *pull/merge request*.

6. Agora que o *pull/merge request* foi criado, inspeccione-o, e aprove-o.

7. No seu repositório local, troque agora para a *branch* principal, fazendo

   ```
   $ git checkout <nome-da-branch-principal>
   ```

8. Invoque `git fetch` para sincronizar o repositório local com as alterações do servidor, e de seguida invoque `git status` para verificar se pode importar novas alterações.

9. Invoque `git pull` para trazer as alterações mais recentes para o seu repositório local.

10. Crie no repositório local uma directoria chamada `Exercícios`. Crie nessa directoria ficheiros Haskell com a resolução dos exercícios a seguir indicados.

11. Relembre as funções de ordem superior estudadas em Programação Funcional:
    `map, filter, takeWhile, dropWhile, span, zipWith, all, any`.
    Use estas funções para reescrever exercícios resolvidos em aulas anteriores.

12. Actualize o repositório local e o servidor com estas resoluções.

Referências

Para informação mais detalhada sugere-se a consulta de documentação do `Git`, nomeadamente:

`https://git-scm.com/doc`

`https://docs.gitlab.com/`