Tema 5

Sistemas de Controlo de Versões

Objetivos:

- Introdução aos sistemas de controlo de versões.

5.1 Introdução

No guião de Sistemas de Colaboração em Projetos referimos que os projetos de desenvolvimento têm uma complexidade acrescida ao nível da escrita do código, verificação das funcionalidades, identificação das alterações e gestão de problemas, o que implica a utilização de plataformas de gestão de projetos.

Quando uma aplicação é desenvolvida por vários programadores, não se espera que trabalhe um de cada vez, ficando os outros à espera. Logo é fundamental a utilização de uma plataforma de gestão de projetos que permita a edição concorrencial de código de forma segura. Neste guião iremos abordar o sistema de controlo de versões Git.

5.2 Sistemas de controlo de versões

Um sistema de controlo de versões (em inglês: Version Control System (VCS)) permite gerir projetos de *software* de maneira a possibilitar a edição concorrente dos ficheiros por diferentes utilizadores ao longo do tempo, mantendo sempre um registo completo de todas as alterações feitas, de quem as fez e quando. Também permitem reverter alterações ou combinar versões. Estes sistemas também são conhecidos como sistemas de controlo de revisões (em inglês: Revision Control System (RCS) ou ainda gestores de configuração de *software* (em inglês: Software Configuration Management (SCM)), mas este termo é mais genérico. Há vários sistemas alternativos de controlo de versões usados atualmente. Neste guião usamos o sistema Git por ser um dos mais poderosos.

Antes de se analisar o funcionamento desta ferramenta, existem alguns termos que é necessário aclarar, pois irão repetir-se ao longo deste guião. Na maioria dos casos os termos usados serão na língua inglesa, uma vez que têm uma correspondência direta com os comandos e operações disponíveis.

O processo de produção de *software* envolve a manipulação de muitos ficheiros por vários programadores, o que implica que exista alguma forma de coordenação das suas atividades para produzir algo coerente e de acordo com especificações inicialmente impostas. Este processo implica um ciclo de produção e de melhoramentos sucessivos que envolve ***working trees*** (árvores de trabalho) e repositórios de versões.

As árvores de trabalho são as áreas que os programadores usam para desenvolver novas funcionalidades ou para corrigir erros e têm por base uma determinada versão do produto em que os programadores estão a trabalhar. Quando estas versões atingem um determinado ponto de maturidade, os programadores podem registá-las no repositório como (mais) uma versão do produto.

À extração de uma versão do repositório para uma árvore de trabalho dá-se o nome de ***check out***. Já ao processo inverso, o de criar uma nova versão no repositório a partir de uma árvore de trabalho, dá-se o nome de ***check in*** ou ***commit***.

Exercício 5.1

Aceda à página de desenvolvimento do kernel *Linux*, que se encontra em `http://git.kernel.org/cgit/` e verifique os múltiplos repositórios Git que contém. Verifique igualmente que cada uma possui um tema específico.

A partir de uma mesma versão no repositório podem criar-se linhas de evolução diferentes, às quais se dá o nome de *branches* (ramos). Dessa forma, a evolução das versões do *software* faz-se através de sucessivos *commits* no mesmo ramo e de ramificações a partir de algumas versões *commited*.

Exercício 5.2

Aceda à *Working Tree* do *Linux* dedicada ao controlo de temperatura, disponível em `http://git.kernel.org/cgit/linux/kernel/git/rzhang/linux.git/` e verifique que existem vários *branches*. Estes ramos contêm diferentes estados de desenvolvimento do código, sendo *master* o ramo atualmente ativo.

Feita esta explicação do paradigma base de atualização de *software* e de controlo de versões, podemos passar para uma descrição mais completa de alguns termos usados pelo Git.

***Working tree* (árvore de trabalho)** - A árvore de trabalho é um diretório no sistema de ficheiros ao qual se encontra associado um repositório (tipicamente existe nesta diretoria um sub-diretório chamado `.git`). A árvore de trabalho inclui todos os ficheiros e sub-diretórios nela existentes. No fundo é onde o programador desenvolve o seu código e tem os seus ficheiros. Cada programador pode ter uma ou mais árvores de trabalho associadas a um mesmo repositório.

***Repository* (repositório)** - Um repositório é uma coleção de *commits*, sendo que cada um destes é um registo do estado em que se encontrava uma dada árvore de trabalho numa data passada. De uma forma simplista pode-se considerar que um *commit* é uma alteração ao código ou versões. Os *commits* de um repositório podem ainda ser identificados como ramos, caso sejam o início de um ramo, ou possuir etiquetas (*tags*) de forma a serem facilmente identificados por um nome.

***Check out* (extração)** - Quando se faz uma extração de uma versão (ou de um *commit*) de um repositório cria-se uma árvore de trabalho com todos os ficheiros e diretorias pertencentes a essa versão. O processo de extração regista também na árvore de trabalho o identificador do ramo ou *commit* do qual a árvore de trabalho actual descende. Esse identificador é genericamente designado por `HEAD`.

***Commit* (ou *check in*)** - Um *commit* é uma cópia de uma árvore de trabalho realizada num dado momento. Para além disso, um *commit* é igualmente uma evolução de algo que existia anteriormente, que é o *commit* a partir do qual a árvore de trabalho foi criada (indicada por `HEAD`). O *commit* anterior torna-se pai do *commit* atual. É esta relação entre *commits* que dá por sua vez origem à noção de histórico de revisões (*revision history*).

***Branch* (ramo)** - Um ramo é uma sequência de *commits* sucessivos que formam uma linha de desenvolvimento independente. No Git os ramos podem ser referenciados por um nome que funciona como sinónimo do último *commit* nesse ramo. A linha principal de desenvolvimento na maioria dos repositórios é feita num ramo chamado `master`. Embora seja um nome definido por omissão, não é de qualquer forma especial.

***Index* (índice)** - O índice também é chamado de *staging area* (área de ensaio/testes), pois regista as alterações efetuadas na árvore de trabalho que serão incluídas no próximo *commit*. O fluxo mais comum de eventos envolvendo o índice é o representado na Figura 5.1. Após a criação de um repositório, cria-se uma árvore de trabalho com um *check out*, na qual são realizada todas as edições de ficheiros. Assim que o trabalho numa árvore de trabalho atinja uma meta (implementação de uma funcionalidade, correção de um erro, fim de um dia de trabalho, compilação com sucesso, etc.), as alterações na mesma são acrescentadas de forma sucessiva ao índice. Assim que o índice contiver todas as alterações que pretende salvaguardar no repositório, num *commit*, as mesmas são transmitidas a este.

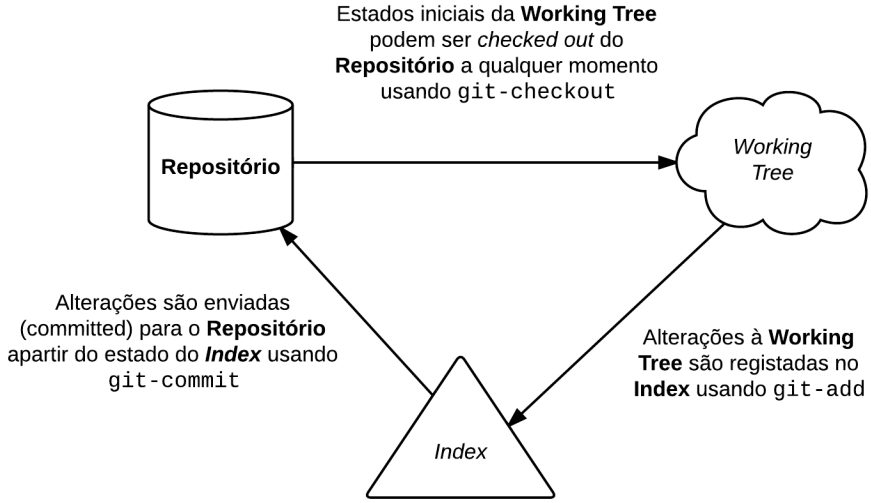

Figura 5.1: Fluxo de eventos no ciclo de vida de um projeto gerido por Git.

Com base neste diagrama, as próximas seções irão descrever a importância de cada uma destas entidades na utilização do Git.

5.2.1 Repositório: Monitorização dos conteúdos de um diretório

Como referido anteriormente, a função do repositório resume-se a manter cópias congeladas dos conteúdos de um diretório tiradas em diversos momentos ao longo do tempo. A estas cópias chamam-se *snapshots* (instantâneos).

A estrutura de um repositório Git assemelha-se à estrutura de um sistema de ficheiros *UNIX*: um sistema de ficheiros começa num diretório raiz, que por sua vez possui outros tantos diretórios, muitos destes têm por sua vez nós folha, ou ficheiros, que contêm dados.

Internamente o Git partilha uma estrutura em tudo similar embora tenha uma ou outra diferença. Em primeiro, representa os conteúdos de um ficheiro em ***blobs***, que também são muito semelhantes a um diretório.

O nome de um *blob*, também chamado o seu ***hash id***, é um número calculado pelo algoritmo Secure Hashing Algorithm (versão 1) (SHA-1) aplicado sobre o tamanho e conteúdos do próprio *blob*. O *hash id* parece um número arbitrário, mas tem duas propriedades interessantes: primeiro, certifica que o conteúdo do *blob* não foi alterado; e segundo, garante que *blobs* de conteúdo igual têm o mesmo nome, independentemente de onde apareçam.

A diferença entre um *blob* do Git e um ficheiro de um sistema de ficheiros é o facto de no *blob* não haver relação qualquer com os conteúdos do ficheiro. Toda essa informação é armazenada na árvore que armazena o *blob*. Uma árvore pode conhecer os conteúdos de um *blob* como sendo o ficheiro **`foo`** criado em Agosto de 2004, ao passo que outra pode conhecer o mesmo ficheiro como sendo o **`bar`** criado cinco anos antes.

5.2.2 Introdução ao *blob*

Agora que tem um panorama geral sobre o funcionamento do Git, resta treinar com alguns exemplos.

Exercício 5.3

Considere o repositório que criou e clonou na realização do guião anterior. Mude para o respetivo diretório e execute os seguintes comandos para associar o seu nome de utilizador e o seu endereço de *email* ao repositório.

```
$ git config --global user.name "nome do utilizador"
$ git config --global user.email "email do utilizador do tipo utilizador@ua.pt"
```

Agora dentro do repositório crie um diretório novo chamado **`teste`** e, dentro dele, crie um ficheiro chamado **`saudacao`** com o texto "Hello, world!".

```
$ mkdir teste
$ cd teste
$ echo "Hello, world!" > saudacao
```

A partir deste momento já pode usar o seguinte comando para conhecer o *hash id* que o Git irá usar para armazenar o texto introduzido.

```
$ git hash-object saudacao
af5626b4a114abcb82d63db7c8082c3c4756e51b
```

No seu computador deverá obter exatamente o mesmo *hash id*. Muito embora esteja a usar um computador diferente daquele onde este guião foi escrito, os *hash ids* serão os mesmos porque os ficheiros têm o mesmo conteúdo. Esta propriedade permite que um repositório seja utilizado por programadores em vários computadores, mantendo a geração dos *blobs* consistente.

Para criar um *blob* que exprima as alterações efetuadas neste ficheiro é necessário usar o comando **`git add`**. Depois podemos fazer um *commit* para acrescentar o *blob* à árvore de trabalho.

```
$ git add saudacao
$ git commit -m "Adicionei a minha saudação"
```

Neste momento o nosso *blob* deverá fazer parte do sistema tal como se esperava, usando o *hash id* determinado anteriormente. Por conveniência o Git requer o menor número de dígitos necessários para identificar inequivocamente o *blob* no repositório. Tipicamente seis ou sete dígitos são suficientes:

```
$ git cat-file -t af5626b
blob
$ git cat-file blob af5626b
Hello, world!
```

Ainda não conhecemos o *commit* que armazena o nosso ficheiro ou a sua *tree*, mas recorrendo exclusivamente ao *hash id* que resume o seu conteúdo, foi possível determinar que o ficheiro existe e consultar o mesmo. Ao longo de toda a vida do repositório, e independentemente do local no repositório onde o ficheiro estiver, este conteúdo manterá esta identificação.

5.2.3 Os *blobs* são armazenados em *trees*

Os conteúdos de um ficheiro são armazenados em *blobs*, mas estes têm poucas funções (não têm nome, nem estrutura, servindo apenas como agregadores de conteudos). Para o Git poder representar a estrutura e o nome dos ficheiros, é necessário associar os *blobs* como nós folha de uma árvore (*tree*). É depois possível determinar que existe um *blob* na *tree* onde foi feito um *commit*. Para listar os *blobs* armazenados na *tree* **`HEAD`** execute:

```
$ git ls-tree HEAD
100644 blob af5626b4a114abcb82d63db7c8082c3c4756e51b    saudacao
```

O primeiro *commit* acrescentou o ficheiro **`saudacao`** ao repositório. Este *commit* contém uma árvore Git, que por sua vez contém apenas uma folha: o *blob* do conteúdo de **`saudacao`**.

É também possível identificar a árvore, tal como fizemos com o *blob*:

```
$ git cat-file commit HEAD
tree 24965b05570bf150f78b4f465e86fe7afa258554
parent 8bb8a6c46483f81c7cab1fc092e638596c506f51
author António Adrego <adrego@ua.pt> 1664358585 +0100
committer António Adrego <adrego@ua.pt> 1664358585 +0100

Adicionei a minha saudação
```

A *hash id* para cada *commit* é única no repositório, visto que contém o nome do autor e a data em que o *commit* foi realizado, mas a *hash id* da árvore deverá ser a mesma no exemplo deste guião e no seu sistema, contendo apenas o nome do *blob* (que é o mesmo).

Vamos verificar que se trata realmente do mesmo objecto *tree*:

```
$ git ls-tree 24965b0
100644 blob af5626b4a114abcb82d63db7c8082c3c4756e51b    saudacao
```

O processo tem início quando se acrescenta um ficheiro ao índice. Por agora, vamos considerar que o índice é usado inicialmente para criar *blobs* a partir de ficheiros. Quando se acrescenta o ficheiro **saudacao** ocorre uma alteração no repositório. Ainda não é possível ver esta alteração através de um *commit*, mas é possível ver o que aconteceu.

Execute:

```
$ git log
commit 88d9ce20ecbc2250bf1663e758ff34739849fd87 (HEAD -> main)
Author: Antonio Adrego <adrego@ua.pt>
Date:   Wed Sep 28 17:17:43 2022 +0100

    Adicionei a minha saudação

commit 8bb8a6c46483f81c7cab1fc092e638596c506f51 (origin/main, origin/HEAD)
Author: antonioadrego <103308983+antonioadrego@users.noreply.github.com>
Date:   Wed Sep 14 11:01:14 2022 +0100

    Initial commit
```

Eis a prova de que o repositório contém um só *commit*, que contém uma referência para uma árvore que armazena um *blob*, *blob* este que armazena o conteúdo do ficheiro **saudacao**.

5.2.4 *Commits*

Um ramo numa *tree* não é, pois, mais do que um nome que referencia um *commit*. É possível examinar todos os *commits* no topo de um ramo usando o comando:

```
$ git branch -v
* main 88d9ce2 [ahead 1] Adicionei a minha saudação
```

Este comando indica que a árvore **master** possui no seu topo um *commit* com *hash id* **88d9ce2**.

Neste exemplo podemos fazer o *reset* da **HEAD** da árvore de trabalho a um *commit* específico. Ou seja, colocar o nosso repositório no estado indicado pelo *commit* respetivo.

```
$ git reset --hard 88d9ce2
HEAD is now at 88d9ce2 Adicionei a minha saudação
```

A opção **-hard** serve para garantir que todas as alterações existentes na árvore de trabalho são removidas, quer tenham sido registadas para um *check in* ou não (mais será dito à frente). Uma alternativa mais segura ao comando anterior seria:

```
$ git checkout 88d9ce2
HEAD is now at 88d9ce2 Adicionei a minha saudação
```

A diferença é que as alterações aos ficheiros na *working tree* serão preservadas. Por exemplo, os ficheiros locais adicionais não são apagados.

Se usarmos a opção **-f** do comando **checkout**, o resultado será semelhante ao uso do **reset -hard**, excepto que **checkout** apenas altera a árvore de trabalho e **reset -hard** altera o **HEAD** do ramo atual para a referência da *tree* passada por argumento.

Um dos benefícios do Git, como sistema orientado a *commits*, é que é possível reescrever processos extremamente complexos usando um subconjunto de processos muito simples. Por exemplo, se um *commit* tiver múltiplos pais, trata-se de um *merge commit*: vários *commits* são unidos num único. Ou se um *commit* tiver múltiplos filhos, representa o antepassado (*ancestor*) de um ramo, etc.

Para o Git estes conceitos não existem na realidade, são apenas "nomes" usados para descrever processos que existiam em sistemas de gestão de código anteriores. Para o Git, tudo é uma colecção de objectos de *commits*, sendo que cada um contém uma *tree* que referencia outras *trees* e *blobs* onde a informação está armazenada. Qualquer outra coisa é apenas uma questão de terminologia. A Figura 5.2 ajuda-nos a compreender melhor.

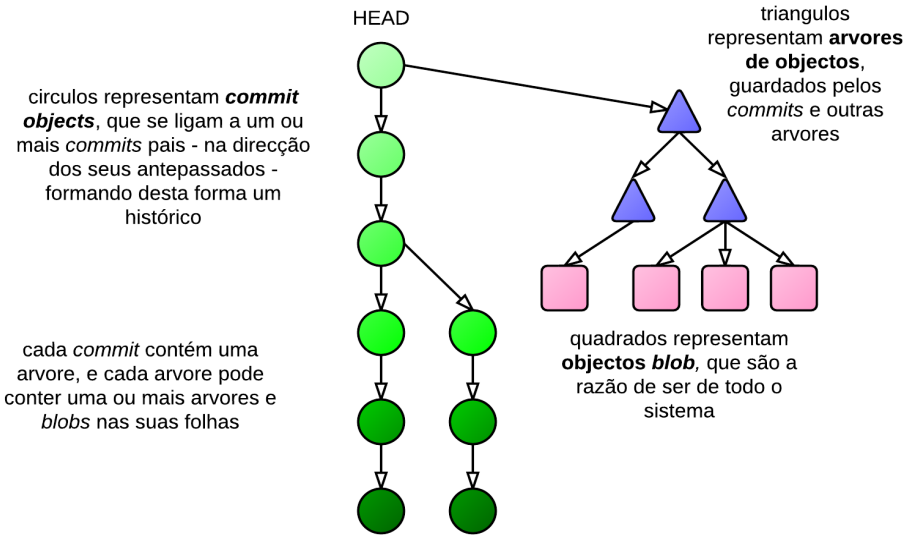

Figura 5.2: Ramificação de *commits*.

5.2.5 Nomes alternativos de *commits*

Já vimos que um mesmo *commit* podem ser referenciado por diversos nomes. Chamamos-lhes sinónimos ou nomes alternativos (em inglês *aliases*). Estes são os principais tipos de sinónimos e os seus casos de utilização:

`eb64bfd...` Um *commit* pode sempre ser referenciado usando o seu *hash id* completo de 40 dígitos hexadecimais. Normalmente isto acontece quando se recorre ao cortar&colar (*cut&paste*), já que existem normalmente outras formas mais práticas de referenciar um *commit*.

`eb64bfd` Apenas é necessário usar um número de dígitos suficientes para garantir que existe apenas um *commit* começado por esses mesmos dígitos no repositório. Na maioria dos casos, entre 6 e 8 dígitos são suficientes.

`HEAD` O *commit* actual, o mais recente, pode ser sempre referido pelo nome **`HEAD`**. Quando se faz um novo *commit*, **`HEAD`** passa a apontar para esse. Se fizer *check out* de um determinado *commit* (em vez de um nome de ramo), então **`HEAD`** refere-se a esse *commit* e não ao de qualquer outro ramo. Este é um caso especial, chamado também de *detached head*.

`branchname` O nome de um ramo, por exemplo **`master`**, funciona como sinónimo do último *commit* feito nesse ramo. É portanto um sinónimo "móvel", tal como **`HEAD`**.

`tagname` É possível associar explicitamente um nome alternativo a um qualquer *commit*, mesmo que não seja um novo ramo. Estes sinónimos chama-se *tags* (etiquetas) e ficam sempre associado ao mesmo *commit*, ao contrário dos nomes de ramos.

`name^` O pai de qualquer *commit* pode ser referenciado usando o acento circunflexo (^). Se um *commit* tiver múltiplos pais, refere-se ao primeiro.

`name^^` Vários acentos circunflexos podem ser utilizados de forma sucessiva. Neste caso, está-se a referir ao pai do nosso pai (avô).

`name^2` No caso de existirem múltiplos pais, pode-se referir a um pai em concreto através do seu número de ordem. No exemplo ao 2º pai.

`name~10` Para aceder a um antepassado distante, pode-se recorrer ao til (~) para indicar quantas gerações subir na *tree*. É a mesma coisa que fazer name^^^^^^^^^^.

`name:path` Para referir um certo ficheiro dentro da *tree* de um determinado *commit*, pode-se especificar o nome do ficheiro após o caráter dois-pontos (:). Esta forma é extremamente útil em conjunto com o comando **`show`**, por exemplo para comparar versões anteriores de um ficheiro:

```
git diff HEAD^1:saudacao HEAD^2:saudacao
```

name^tree É possível referenciar a *tree* de um *commit*, na vez do próprio *commit*.

name1..name2 Esta notação permite indicar uma gama de *commits*: todos os ocorridos após **name1** (sem o incluir) e até **name2**. (Mais rigorosamente, indica todos os antepassados de **name2** que não sejam antepassados de **name1**.) Se se omitir **name1** ou **name2**, **HEAD** é usado em seu lugar. É muito útil quando usado em conjunto com o comando **log** por forma a analisar o que ocorreu num determinado período de tempo.

name1...name2 O uso de três pontos é bastante diferente do uso anterior de dois pontos. Em comandos como **log**, refere-se a todos os *commits* antepassados de **name1** ou de **name2**, mas não de ambos.

master.. É equivalente a **master..HEAD**. Ou seja, inclui todos os *commits* desde que o ramo atual se desviou de master.

..master Também é uma equivalência, especialmente útil quando usada com o comando **fetch** para determinar que alterações ocorreram desde o último **rebase** ou **merge**.

--since="2 weeks ago" Refere-se a todos os *commits* desde uma data.

--until="1 week ago" Refere-se a todos *commits* até uma data.

--grep=pattern Refere-se a todos *commits* cuja a mensagem contém o padrão "pattern".

--committer=pattern Refere-se a todos *commits* cujo autor (*committer*) contém no seu nome o padrão "pattern".

--author=pattern Normalmente é o mesmo que *committer*, mas nalguns casos pode ser diferente (envio de *patches* por email).

--no-merges Refere-se a todos os *commits* na gama que têm apenas um pai — desta forma ignora todos os *commits* de *merge*.

5.2.6 Índice: o intermediário

Entre os nossos ficheiros que contêm dados e estão armazenados na árvore de trabalho, e os *blobs* Git, que estão armazenados no repositório, fica a área de ensaio (*staging area*). A esta área também se chama índice, porque é realmente uma lista de apontadores para *trees* e *blobs* criados através do comando **add**.

Estes novos objectos são depois compactados numa nova *tree* que será *commited* no repositório. Tal quer também dizer que se aplicarmos **reset** sobre o índice, todas as alterações não *commited* serão perdidas.

O índice (ver Figura 5.3) é pois uma zona de preparação para o próximo *commit* e existe uma boa razão para existir: desta forma é possível ter mais controlo sobre as alterações a submeter. Por exemplo, podemos preparar um *commit* com vários ficheiros alterados, mas evitar incluir outro em que ainda estamos a trabalhar e que causaria problemas (como erros de compilação). Também é possível usar o Git sem usar o índice, recorrendo à opção `-a` sempre que fazemos um *commit*. Desta maneira todas as alterações feitas são submetidas no mesmo *commit*.

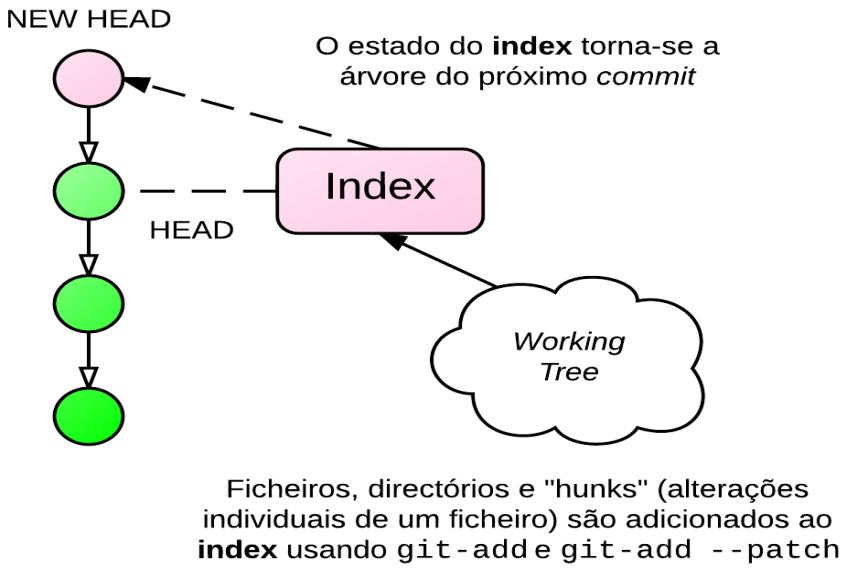

Figura 5.3: O índice e a árvore de trabalho.

5.2.7 Utilização de um repositório Git

Nas seções anteriores foram descritas algumas funcionalidades internas do Git e diversos conceitos associados aos sistemas de gestão de versões. Nesta seção serão apresentados os comandos mais práticos e úteis de utilização no dia-a-dia.

Sincronize com o repositório no servidor usando o comando **`git push`** na seguinte forma:

```
$ git push --all origin
Username for 'https://github.com': antonioadrego
Password for 'https://antonioadrego@github.com':
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 2 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (4/4), 346 bytes | 346.00 KiB/s, done.
Total 4 (delta 0), reused 0 (delta 0)
To https://github.com/antonioadrego/infor2022-txgy.git
   8bb8a6c..88d9ce2  main -> main
```

Isto evidencia um aspeto importante do Git: todas as alterações, mesmo que gerando novos *commits*, são efetuadas na cópia local do repositório. É o comando **`git push`** que força a sincronização do repositório remoto com o local.

Nota Muito Importante! A opção **`-all origin`** é necessária apenas no primeiro *push* pois ainda não existe nenhum **`HEAD`** no repositório do servidor.

Após a execução do comando aceda à plataforma **`GitHub`** e verifique que o seu repositório foi atualizado com o diretório **`teste`** e respetivo conteúdo tal como se mostra na Figura 5.4

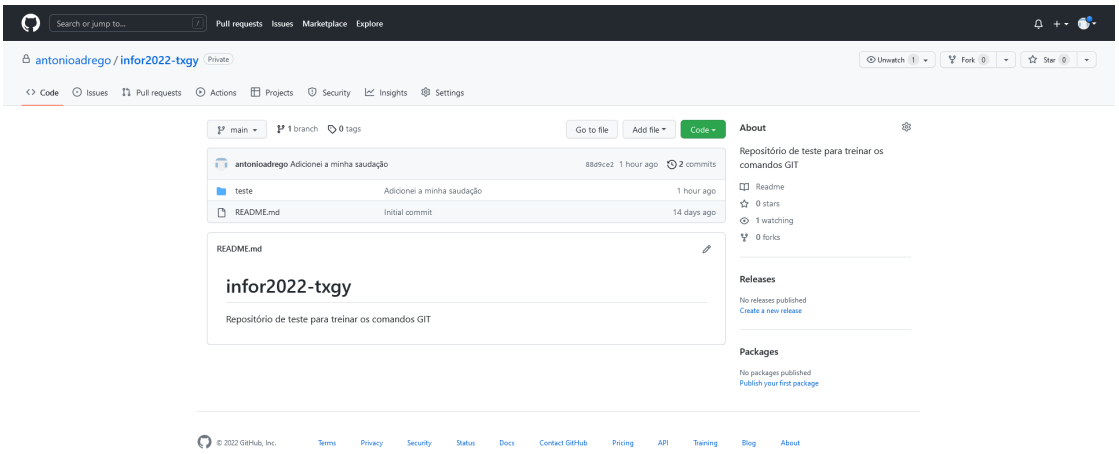

Figura 5.4: Página do repositório depois de atualizado.

Nota Muito Importante! Depois do primeiro *push* deve ser usada a opção **`origin main`**, uma vez que estamos a usar o ramo principal do repositório. No **`GitHub`** o ramo principal do repositório, por defeito designa-se por **`main`**, como se pode ver na Figura 5.4.

Uma vantagem de utilizar um repositório centralizado num servidor é que está sempre disponível para os vários programadores poderem submeter as suas alterações paralelamente. Porém o Git também pode ser usado de forma perfeitamente distribuída, ao contrário de outros VCS.

Exercício 5.4

Inicie sessão no servidor **`deti-labi.ua.pt`** utilizando Secure Shell (SSH).

Obtenha neste servidor uma cópia do repositório criado na plataforma **`GitHub`**, usando o comando **`git clone`**.

Verifique que os conteúdos locais, os mostrados na página do repositório do **`GitHub`** e os presentes no **`deti-labi.ua.pt`** são exatamente os mesmos.

Exercício 5.5

No servidor **deti-labi.ua.pt** faça algumas alterações ao ficheiro **README.md**, por exemplo adicionando o seu número mecanográfico, nome e email de contacto.

Submeta e envie as alterações para o repositório remoto utilizando as seguintes instruções. Após a execução de cada instrução interprete a mensagem que aparece no monitor, nomeadamente das instruções **git commit** e **git status**.

```
$ git add README.md
$ git status
$ git commit -m "Alterações feitas no ficheiro README.md"
$ git status
$ git push origin main
```

O comando simétrico do **git push**, ou seja, o comando que força a sincronização do repositório local com o remoto é o comando **git pull**. No computador local, para sincronizar o repositório local, utilize o comando **git pull** na seguinte forma:

```
$ git pull origin main
```

Como descrito anteriormente, o Git regista alterações, o que também inclui remoção de documentos. O comando **git rm** permite registar a remoção de ficheiros. No entanto, a remoção é apenas uma alteração ao estado do ficheiro e como qualquer alteração no Git, é sempre possível recuperar um estado anterior, ou seja, ir para o estado exacto de qualquer *commit*.

A consequência é que uma vez que um ficheiro é adicionado a um repositório, não é possível removê-lo definitivamente. Desta forma nunca existe perda de informação.

Considere a seguinte execução de comandos:

```
$ echo "teste" > fich.txt
$ git add fich.txt
$ git commit -m "teste"
$ git rm fich.txt
$ git commit -m "teste"
```

O resultado deverão ser 2 *commits*. Um registando o ficheiro **fich.txt**, e outro sinalizando a sua remoção. O comando **git log** deverá demonstrar esta sequência.

Exercício 5.6

No repositório do computador local, adicione um ficheiro **test.txt**, efectue um *commit* e sincronize o repositório remoto da seguinte maneira:

```
$ echo "teste de remoção de ficheiro" > test.txt
$ git add test.txt
$ git commit -m "Novo ficheiro test.txt"
$ git push origin main
```

Verifique que este ficheiro se encontra no repositório do **GitHub** e no servidor **deti-labi.ua.pt** (após um **git pull** para atualizar o repositório local).

No servidor **deti-labi.ua.pt** remova o ficheiro e envie as alterações para o repositório remoto da seguinte maneira:

```
$ git rm test.txt
$ git commit -m "teste delete"
$ git push origin main
```

Verifique que uma nova atualização (**git pull**) no computador local irá remover o ficheiro adicionado.

O comando **git log**, quer no repositório do servidor **deti-labi.ua.pt**, quer no repositório local, deverá mostrar os passos dados.

Um ficheiro pode ser recuperado. Basta colocar a árvore de trabalho no local correto da *tree*. Uma maneira de realizar isto é executar um **git reset** para um *commit* anterior.

Exercício 5.7

Utilize o comando **git log** e localize o *commit* em que o ficheiro **test.txt** foi adicionado. Pode também utilizar o comando **git show <hashid>** para ver o conteúdo do *commit*, isto é, quais as modificações que o *commit* efetuou.

Utilize o comando **git reset** para reaver o ficheiro perdido.

Exercício 5.8

Altere o ficheiro **test.txt** para incluir mais texto.

Utilize o comando **git diff test.txt** para verificar a diferença entre o ficheiro existente na árvore de trabalho e o registado na *tree*.

Muitos mais comandos existem que não foram mencionados neste guião. No entanto, todos obedecem às regras descritas neste guião. Processos mais avançados de gestão de versões concorrentes ficam fora do âmbito deste guião.

Glossário

RCS	Revision Control System
SCM	Software Configuration Management
SHA-1	Secure Hashing Algorithm (versão 1)
SSH	Secure Shell
VCS	Version Control System