Tema 5

Colaboração em Projetos

Objetivos:

- Sistemas de colaboração em projetos.
- Introdução aos sistemas de controlo de versões.

5.1 Introdução

Quando diversas entidades colaboram num projeto comum é necessário que existam meios de coordenar as atividades. Tal não se faz através de plataformas genéricas como o *Facebook* ou o *Google+*. Por exemplo, é necessário saber se o projeto está atrasado ou não, quais são as próximas tarefas e quem está responsável por elas, ou quem realizou uma determinada tarefa. Também é necessário que existam meios de comunicação rápida entre todos, tal como uma *mailing-list*. Também pode existir necessidade de ter uma base de informação partilhada.

Esta é a razão pela qual utilizamos a plataforma `http://elearning.ua.pt`, pois possui ferramentas que facilitam o desenrolar das aulas e a colaboração entre docentes e alunos.

Quando se fala de projetos com caráter de desenvolvimento de aplicações (programação), o processo de colaboração necessita de meios ainda mais evoluídos. Isto porque os projetos de desenvolvimento têm uma complexidade acrescida ao nível da escrita do código, verificação das funcionalidades, identificação das alterações e gestão de problemas (vg. *bugs*[1]). De igual forma, quando uma aplicação é desenvolvida por vários programadores, não se espera que trabalhe um de cada vez, ficando os outros à espera. Na Secção 5.2 iremos abordar a gestão de processos de devenvolvimento através da plataforma **`CodeUA`**, enquanto que na Secção 5.3 iremos abordar a edição concorrencial de código.

[1] Ver `http://en.wikipedia.org/wiki/Software_bug`

5.2 A plataforma CodeUA

A plataforma **CodeUA**, que pode ser encontrada em `http://code.ua.pt`, é um sistema de gestão de projetos disponível para qualquer utilizador na Universidade de Aveiro. Foi implementado com o software livre Redmine[2] e inclui diversas ferramentas para facilitar a gestão de projetos, tais como: fóruns de discussão, *Wikis*, gestão e seguimento de tarefas, calendários, diagramas de *Gantt*[3] e integração com sistemas de controlo de versões. A plataforma pode ser utilizada para a gestão de qualquer tipo de projeto, mas é particularmente útil para projetos de desenvolvimento de aplicações, que mais podem beneficiar do sistema de controlo de versões de software coordenado com as ferramentas de gestão de tarefas, de funcionalidades e de pedidos de suporte. Embora o guião seja baseado na plataforma **CodeUA**, as funcionalidades descritas são análogas às de outras plataformas de gestão de projetos.

Os alunos podem usar a plataforma **CodeUA** para gerir projetos das suas disciplinas, os professores utilizam-na para criar e partilhar conteúdos das aulas, e todos a podem utilizar para divulgar trabalhos que realizem durante o seu contacto com a *UA*.

A Figura 5.1 mostra a página inicial da plataforma **CodeUA**. Esta página inclui uma mensagem de boas vindas (lado esquerdo) e uma lista de projetos criados recentemente (lado direito). Na barra superior pode consultar a lista de projetos públicos, iniciar uma sessão ou consultar a ajuda.

Os projetos apresentados são apenas uma parte de todos os projetos. Se ainda não iniciou uma sessão, será apresentada a lista de projetos públicos. Para qualquer um destes projetos, pode verificar qual o seu propósito, quem são os seus membros, ou até contribuir para eles.

Exercício 5.1

Aceda à aplicação **CodeUA** no endereço `http://code.ua.pt` e consulte a lista de projetos públicos. Aceda a alguns e verifique a sua descrição e os seus membros.

Seja curioso! Alguns projetos poderão ser do seu interesse.

Um projeto em particular, chamado **CodeUA**, e disponível no endereço `http://code.ua.pt/projects/codeua` serve para se trocar informação sobre a plataforma. Sempre que tiver dúvidas sobre o seu funcionamento, ou encontrar problemas, deve dirigir-se a este projeto e colocar a sua questão.

[2] Ver `http://www.redmine.org/`
[3] Ver `http://pt.wikipedia.org/wiki/Diagrama_de_Gantt`

Figura 5.1: Página de entrada da plataforma **CodeUA**.

Todos os membros da *UA* podem possuir uma conta na plataforma **CodeUA**, bastando para isso identificarem-se perante o sistema. A partir desse momento passa a ser possível criar projetos e gerir o seu funcionamento.

Um aspeto importante em qualquer projeto é a gestão dos seus membros. No caso de projetos privados, apenas os seus membros terão acesso aos conteúdos do projeto. Para projetos públicos, o acesso que os membros possuem será diferenciado dos restantes (vg. convidados). Uma diferenciação normal é que os convidados apenas podem aceder a conteúdos existentes, enquanto os membros podem criar conteúdos.

De entre os membros também é possível diferenciar qual o seu papel no projeto. A plataforma **CodeUA** distingue entre vários papéis:

Manager: Gere o projeto e define como deve prosseguir.

Developer: Desenvolve conteúdos para o projeto. Num projeto de desenvolvimento de uma aplicação, estes serão os programadores.

Reporter: Relata questões ou problemas para o projeto, mas não tem como função o desenvolvimento de conteúdo. Num projeto de desenvolvimento de uma aplicação, estes serão pessoas que testam a aplicação ou que fornecem algum *feedback* sobre ela.

Exercício 5.2

Aceda à plataforma. Na lista de projetos, crie um projeto com o identificador `labi2020-tXgY` em que ***X*** representa o número da turma e ***Y*** representa o seu grupo. Como este projeto vai ser um projeto para gestão do grupo, defina-o como privado.

De entre os módulos, escolha apenas o módulo de *notícias*.

Pode adicionar uma descrição e definir um nome, que pode ser diferente do identificador.

Defina os membros do grupo como gestores (*Manager*) e o seu professor como *Reporter*.

Nas subseções 5.2.1, 5.2.2 e 5.2.3 serão abordados alguns dos módulos mais relevantes (e geralmente utilizados) para a gestão de projetos. Existem no entanto outros que não irão ser abordados, mas que são interessantes para alguns casos. Experimente-os.

5.2.1 Comunicação no Projeto

A comunicação é vital nos projetos. Para isso a plataforma `CodeUA` possui o módulo de *notícias* que permite a divulgação de anúncios. Estes anúncios são enviados para todos os participantes do projeto e ficam disponíveis para consulta pelos membros.

Além disto, se o projeto for público a notícia fica disponível para todos os indivíduos que visitem o projeto. É extremamente útil para anunciar novas funcionalidades ou marcos no desenvolvimento do projeto.

As notícias não são uma simples mensagem e podem ser bastante mais ricas (ver a Figura 5.2). São compostas por 3 partes: título, sumário e descrição. O título identifica claramente a notícia. O sumário é um texto curto, apresentado com a lista de notícias, que deve ajudar o leitor a decidir se a notícia lhe interessa ou não. O campo descrição permite escrever o corpo da notícia. Aqui é possível recorrer a alguma formatação na composição da notícia, bem como adicionar ficheiros e imagens.

Repare na Figura 5.2 como se adiciona uma imagem. Em primeiro lugar é necessário adicionar o ficheiro e depois ele pode ser referido no texto entre carateres `!!`.

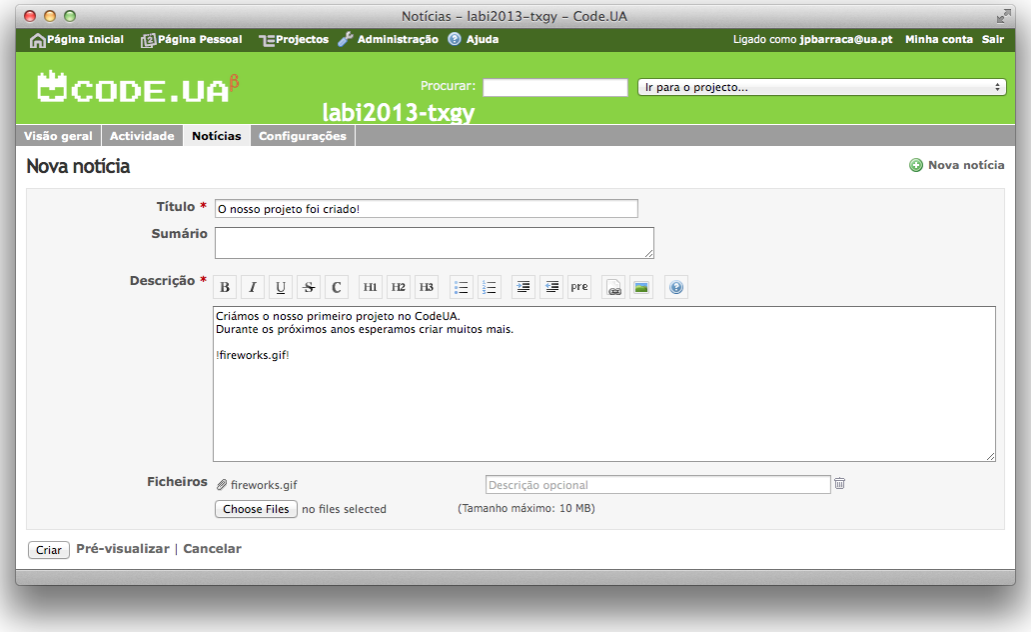


Figura 5.2: Página de introdução de uma notícia.

Exercício 5.3

Crie uma notícia no seu projeto anunciando a sua criação.

Dado o tema da notícia, adicione uma imagem para comemorar o evento. O resultado deverá ser semelhante ao apresentado na Figura 5.3.

Verifique se recebeu uma notificação na sua caixa de correio eletrónico da *UA*. Isto pode demorar alguns minutos.

Também relacionado com o módulo de notícias é o módulo de *fóruns*. Este módulo implementa um modelo em que se permite discutir assuntos de uma forma contextualizada. Isto é, é possível criar vários fóruns, um para cada sub-tema ou aspeto do projeto, onde os membros podem trocar ideias, tal como num fórum comum da Internet.

Exercício 5.4

Crie um fórum para o primeiro trabalho de aprofundamento de conhecimentos, para discussão de aspetos deste primeiro trabalho. Experimente depois enviar uma mensagem para o fórum.

Verifique a sua caixa postal na *UA*.

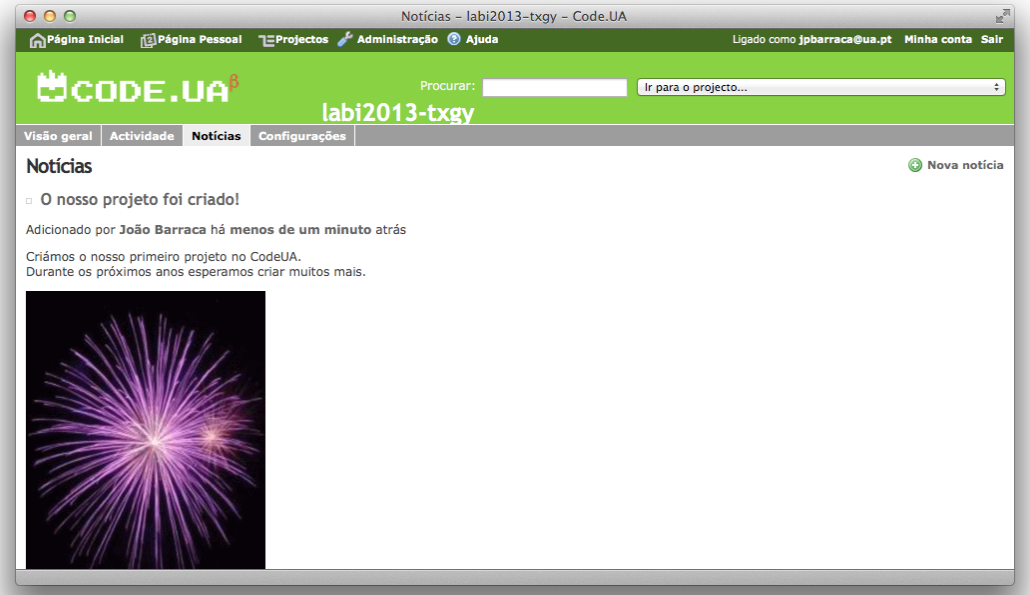


Figura 5.3: Página de apresentação das notícias.

5.2.2 Gestão de Tarefas

Em qualquer projeto colaborativo existem sempre tarefas. Estas podem ser de vários tipos e podem estar em diferentes estados. Podem existir tarefas que relatam problemas que existam na aplicação desenvolvida. Podem existir tarefas que identificam funcionalidades interessantes que se pretendem desenvolver. Outras podem referir-se a ações que há necessidade de realizar no futuro. Mesmo em pequenos projetos, o número de tarefas pode ser bastante grande. Basta que para isso os membros do projeto formalizem o que é necessário realizar e os problemas encontrados na forma de tarefas pendentes.

As tarefas podem seguir um caminho desde que são criadas até que estão terminadas, necessitando de diversas interações dos participantes. Por exemplo, um relatório de um problema gera uma tarefa que necessita de discussão até que se compreenda o problema, se prepare uma solução e se verifique que a solução funciona. Tudo isto deverá estar devidamente documentado na plataforma.

Como pode constatar, a utilização desta metodologia, baseada em tarefas que são criadas e listadas numa plataforma comum, permite gerir com eficiência um projeto, sem que aspetos importantes sejam esquecidos.

A Figura 5.4 apresenta a interface para criação de uma nova tarefa. Note que é possível criar tarefas relativas a *Bugs* (problemas) e *Funcionalidades* (aspetos a acrescentar). Segue-se um campo de título que identifica a tarefa e uma descrição. Estes campos devem ser utilizados para descrever corretamente a tarefa.

Caso a tarefa seja o relato de um problema, deverá descrever-se o problema com detalhe de forma a auxiliar a resolução. Em alguns casos, ficheiros com capturas de ecrã ou registos de erros podem facilitar a análise.

Caso a tarefa corresponda a uma funcionalidade, devem descrever-se os detalhes da funcionalidade de forma a que não exista qualquer dúvida em relação ao que falta fazer. Podem especificar-se outros aspetos como a prioridade da tarefa, o tempo estimado para a usa resolução ou mesmo se esta tarefa está relacionada com outra. Podem-se igualmente definir observadores, que são utilizadores que poderão estar interessados em seguir o desenvolvimento da tarefa.

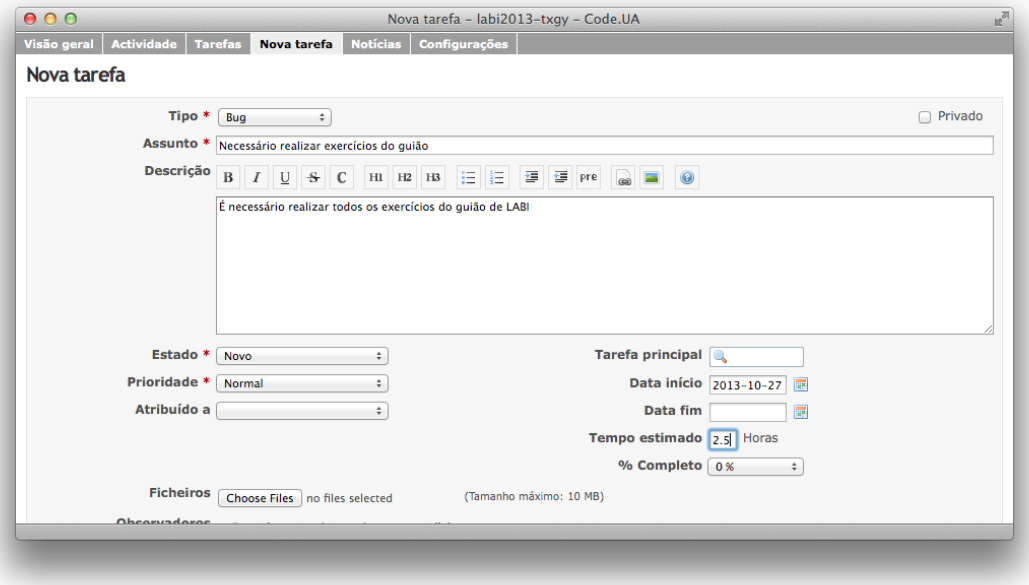


Figura 5.4: Página de criação de uma nova tarefa.

Exercício 5.5

Crie uma tarefa indicando a necessidade de realizar os exercícios desta seção do guião. Qual o tipo desta tarefa?

Adicione todo o detalhe que ache importante.

Uma tarefa possui um percurso de vida bem definido, podendo ter um de dois estados principais: *aberto* ou *fechado*. O estado de fechado corresponde a uma tarefa que já concluiu o seu percurso e está tratada. O estado de aberto corresponde a uma tarefa que ainda se encontra ativa, podendo depois ter vários sub-estados. A Figura 5.5 mostra uma lista de tarefas abertas, indicando igualmente o sub-estado, neste caso *Novo*.

Figura 5.5: Página de listagem das tarefas

A plataforma `CodeUA` suporta vários sub-estados, a saber:

***Novo*:** Estado definido para todas as tarefas recentemente criadas e que ainda não foram processadas para um outro sub-estado.

***Em Curso*:** Tarefa que se encontra em processo de resolução. Isto é, a funcionalidade está a ser implementada ou o problema (*Bug*) está a ser resolvido.

***Feedback*:** Após ter sido tomada uma ação sobre uma tarefa, o criador da tarefa tem de validar que o assunto foi abordado de forma correta. Até lá a tarefa fica a aguardar *Feedback*.

***Recusado*:** A tarefa não se enquadra no projeto, não é válida, ou não foi especificada de forma correta, tendo sido recusada de qualquer outro processamento.

***Resolvido*:** A tarefa foi resolvida e o seu criador verificou que tudo está de acordo com o reportado.

***Fechado*:** O gestor, ou outro membro do projeto pode marcar a tarefa como inativa definindo-a como estando no sub-estado *Fechado*. Uma tarefa deve ir para o sub-estado de *Fechado* após ter sido recusada ou resolvida.

Exercício 5.6

Agora que se encontra a resolver os exercícios do guião, marque a tarefa como estando no estado *Em Curso* e defina quem é o responsável (Atribuído a).

Pode também adicionar uma estimativa de horas e indicar o quanto ela se encontra completa.

O módulo de *Tarefas* comunica com o módulo de *Diagramas de Gantt*. Este módulo é extremamente útil para planear corretamente um projeto, permitindo avaliar se este se encontra dentro das expetativas temporais. O seu funcionamento baseia-se na lista de tarefas do projeto, e caso elas possuam datas de início e de fim, na representação destas num formato que permita identificar as diferentes fases do projeto.

A Figura 5.6 apresenta o diagrama de *Gantt* relativo às tarefas de um hipotético trabalho de aprofundamento de conhecimentos. A linha vermelha representa o instante atual, sendo que cada barra horizontal representa a data de início e a duração de uma tarefa.

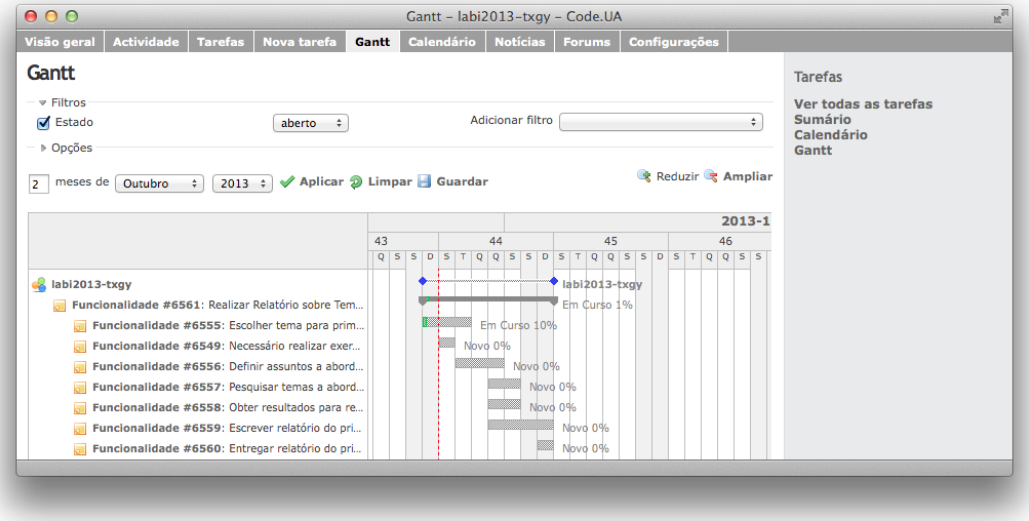


Figura 5.6: Diagrama de Gantt para o trabalho de aprofundamento de conhecimentos.

Exercício 5.7

Defina várias tarefas e sub-tarefas, com datas de início e fim diferentes. Defina que algumas destas tarefas já se encontram em curso, tendo uma percentagem do seu trabalho já realizado. Verifique depois qual o diagrama de *Gantt* produzido.

5.2.3 Partilha de informação

A plataforma `CodeUA` e, de uma forma geral, todas as plataformas de gestão de projetos, mesmo que não sejam orientadas a projetos de programação, possuem áreas para a partilha de ficheiros entre os seus membros ou entre os membros e o público em geral. Por exemplo, num projeto de elaboração de um relatório técnico, todos os artigos pesquisados, resultados obtidos e as diferentes partes do relatório irão estar alojadas na plataforma. Com isto tenta-se maximizar o paralelismo das ações, minimizando dependências entre membros. Isto é, nunca um projeto deve estar dependente das ações de um membro em específico ou pelo menos este tipo de eventos deve ser minimizado. Considerando que a lista de tarefas se encontra descrita, é natural presumir que diferentes membros irão focar-se em diferentes tarefas em paralelo, o que minimiza a duração do projeto.

A plataforma `CodeUA` em específico suporta 3 formas de alojar informação, cada uma adaptada a um fim diferente.

***Documentos*:** Permite alojar documentos gerais do projeto, não relacionados com uma versão de desenvolvimento ou considerados estáticos. Uma notícia para a comunicação social é um exemplo de um documento para esta área.

***Ficheiros*:** Permite alojar ficheiros do projeto em que se espera que existam versões diferentes destes ficheiros. Um exemplo será a publicação de versões diferentes de um programa que esteja a ser desenvolvido.

***Repositórios*:** Permite alojar conteúdos de trabalho do projeto geridos por um sistema de controlo de versões.

Exercício 5.8

Ative os módulos de *Documentos* e de *Ficheiros*. Partilhe um ficheiro local em cada um destes módulos.

O módulo de *Repositórios* merece uma atenção particular porque permite associar ao projeto um repositório de ficheiros gerido por um *sistema de controlo de versões*. Estes sistemas são muito úteis, especialmente em projetos de software, porque mantêm um registo histórico da evolução do projeto. O `CodeUA` suporta dois sistemas alternativos de controlo de versões: Apache Subversion (SVN) ou Git.

5.3 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 5.9

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 5.10

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 5.7. 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.

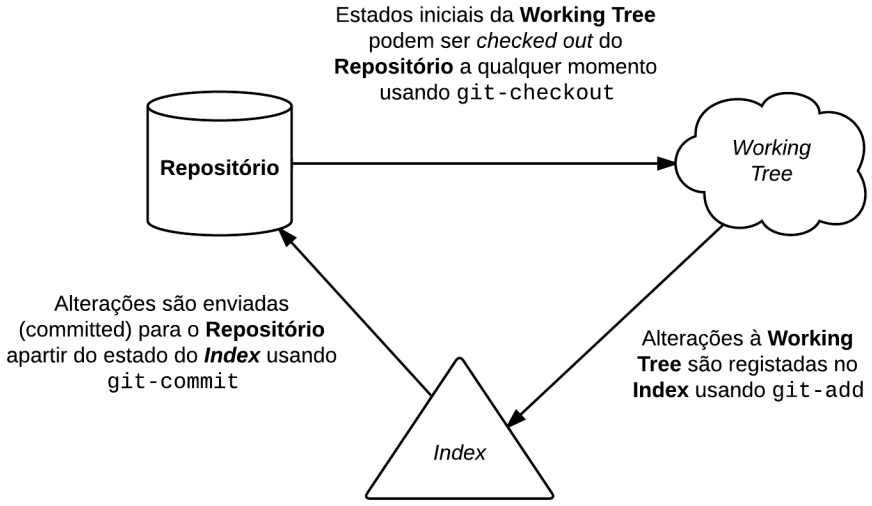


Figura 5.7: Fluxo de eventos no ciclo de vida de um projeto gerido por Git.

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

5.3.1 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 do *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.

5.3.2 Introdução ao *blob*

Agora que tem um panorama geral sobre o funcionamento do Git, resta treinar com alguns exemplos. Como primeiro passo irá criar um repositório exemplo e mostrar como o Git funciona.

Exercício 5.11

Dirija-se ao seu projeto no `CodeUA`, ative o módulo de repositórios e crie um repositório do tipo Git.

Copie o comando indicado no cimo da página e execute-o numa linha de comando. Deverá ser algo como: `git clone https://code.ua.pt/git/labi2020-txgy`.

Este comando deve criar um novo diretório `labi2020-txgy/`, que é uma réplica local do repositório do projeto. É nesse diretório que irá criar e atualizar os ficheiros do projeto. Mude o seu directório atual para lá (`cd labi2020-txgy`).

Cada membro do projeto deverá fazer a clonagem, agora ou mais tarde, para criar a sua cópia pessoal do repositório.

Depois de clonar e entrar num repositório Git deve definir o seu nome de utilizador e o seu endereço de *email*. Isto é importante porque todos os *commits* usam esta informação que é imutavelmente incorporada nos *commist* que executar. Para esse efeito deve executar os seguintes comands:

```
git config --global user.name "Nome do elemento do grupo"
git config --global user.email "email do elemento do grupo@ua.pt"
```

Exercício 5.12

Crie um directório 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.

Agora podemos fazer um *commit*.

```
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.

5.3.3 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*.

Exercício 5.13

Liste os *blobs* armazenados na *tree* **HEAD**.

```
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 5ae5597b6ae0854f77d50dc8ec828eb0928b9fe2
author João Paulo Barraca <jpbarraca@ua.pt> 1382903138 +0000
committer João Paulo Barraca <jpbarraca@ua.pt> 1382903138 +0000

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 5ae5597
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
```

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**.

5.3.4 *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
* master eb64bfd Adicionei a minha saudação
```

Este comando indica que a árvore **master** possui no seu topo um *commit* com *hash id* **eb64bfd**.

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 eb64bfd
```

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 eb64bfd
```

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 5.8 ajuda-nos a compreender melhor.

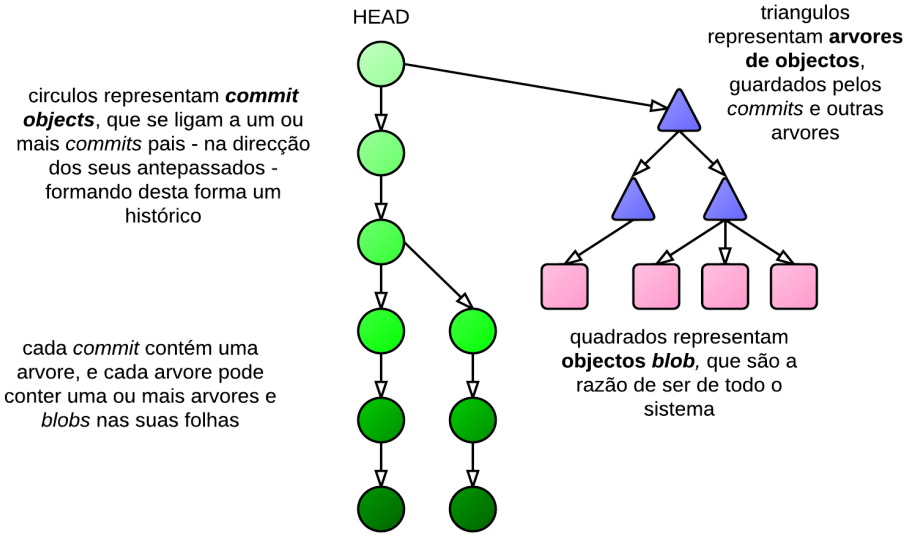


Figura 5.8: Ramificação de *commits*.

5.3.5 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).

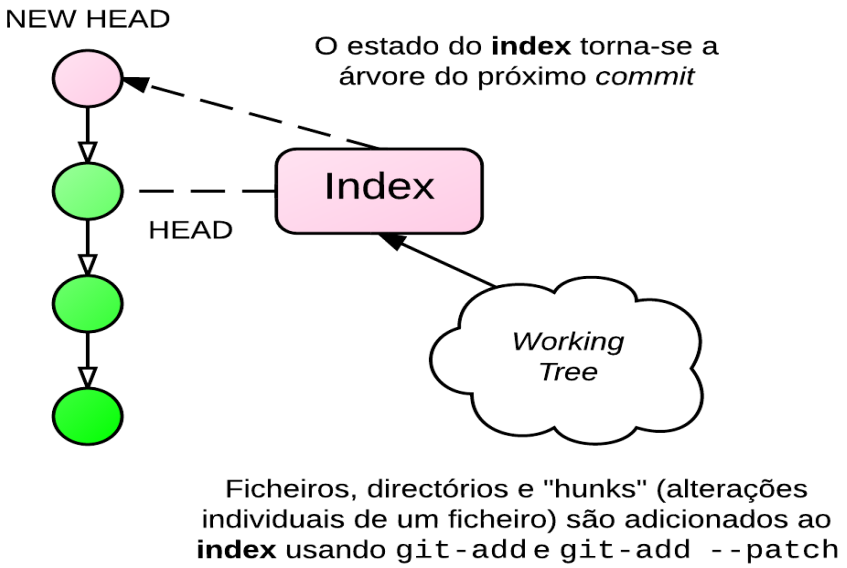
--no-merges Refere-se a todos os *commits* na gama que têm apenas um pai — desta forma ignora todos os *commits* de *merge*.

5.3.6 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 caso se aplicarmos **reset** sobre o índice, todas as alterações não *commited* serão perdidas.

O índice 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*.



Exercício 5.14

Altere o ficheiro criado, de forma a adicionar o nome do seu grupo.

De seguida utilize o comando **`git add`** para adicionar este ficheiro ao índice e depois **`git commit`** para criar um *commit*.

Use o comando **`git status`** para verificar que alterações existem na árvore de trabalho.

5.3.7 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.

Na raíz da árvore de trabalho do seu projeto, crie um ficheiro **`README.md`** com um pequeno texto. De seguida adicione o ficheiro **`README.md`** ao repositório.

```
echo "Este é o projeto labi-txgy" > README.md
git add README.md
git commit -m "Commit README inicial"
```

Agora altere o ficheiro **`README.md`** de modo a incluir uma descrição do que este repositório irá conter. Volte a registar o *commit* e a enviar para o servidor.

Sincronize com o repositório no servidor usando o comando:

```
git push --all origin
```

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

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 5.15

Inicie sessão no servidor **`xcoa.av.it.pt`** utilizando Secure Shell (SSH).

Obtenha neste servidor uma cópia do repositório criado na plataforma **`CodeUA`**. Verifique que os conteúdos locais, os mostrados na página do repositório do **`CodeUA`** e os presentes no **`xcoa.av.it.pt`** são exactamente os mesmos.

Exercício 5.16

No computador local, faça novas alterações ao ficheiro **`README.md`**, por exemplo adicionando o email de contacto e número mecanográfico dos seus membros. Submeta e envie as alterações para o repositório remoto.
No servidor **`xcoa.av.it.pt`**, dentro do directório do projeto, utilize o comando **`git pull`** para actualizar o repositório local.

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 5.17

Adicione um ficheiro **`test.txt`** com um conteúdo arbitrário e efectue um *commit*, enviando de seguida o novo *commit* para o repositório remoto.

Verifique que pode obter este ficheiro na interface web da plataforma **`CodeUA`** e no servidor **`xcoa.av.it.pt`** (após um **`git pull`**).

Remova o ficheiro e envie as alterações para o servidor remoto. Verifique que uma nova atualização (**`git pull`**) na outra máquina irá remover o ficheiro adicionado.

O comando **`git log`** no repositório do servidor **`xcoa.av.it.pt`** 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 5.18

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 5.19

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.

5.4 Para aprofundar

Exercício 5.20

Utilize os métodos descritos para gerir a elaboração do próximo relatório. Pode definir todas as etapas de realização de um relatório técnico, definir o conteúdo das seções e definir quem está responsável por que parte. Quando todas as tarefas estiverem no estado *Resolvido*, o relatório deverá estar pronto a ser entregue.

Exercício 5.21

Explore os outros módulos da plataforma `CodeUA` e identifique qual a sua utilidade para a gestão de projetos.

Exercício 5.22

De uma forma geral, o Git (ou outro sistema semelhante) é extremamente útil a qualquer programador. Experimente criar um repositório para guardar os trabalhos que faz noutras disciplinas, por exemplo em Programação I.

Glossário

RCS Revision Control System

SCM Software Configuration Management

SHA-1 Secure Hashing Algorithm (versão 1)

SSH Secure Shell

SVN Apache Subversion

VCS Version Control System