

CSS

Construção de Sistemas de Software - LTI

O presente documento agrupa material oriundo de várias fontes:

- documentos elaborados por mim para as edições anteriores da disciplina (para a versão LTI ou LEI pós-laboral),
- documentos escritos por mim para outras disciplinas (LABP, PCO, PPO),
- documentos elaborados em conjunto com colegas para outras disciplinas (Guião Git),
- slides e exercícios criados por colegas para edições anteriores desta disciplina (CSS-LTI) ou (CSS-LEI) com uma menção especial para o meu colega Francisco Martins que imaginou e criou esta disciplina.
- finalmente, foram usadas várias fontes na internet.

Objectivos p.3 • Controle de versões p.5 • Desenvolvimento em camadas p.63 • Padrões para a camada de negócio p.69 • Padrões para a camada de persistência p.91 • Camada de apresentação (WEB) p.155 • Project Memories p.185 • Guiões p.195 • Revisões p.197

Capítulo 1

Objectivos

Pretende-se que o aluno adquira competências no desenho e implementação de sistemas em grande escala através da aplicação de padrões, que traduzem as boas práticas de arquitetura e desenho destes sistemas, e que lhes forneçam competências no desenvolvimento e testes de sistemas concorrentes, distribuídos, construídos a partir de componentes locais ou distribuídas pela Web, e que utilizem servidores aplicacionais e de base de dados. O foco da disciplina vai integralmente para as construção de software do ponto de vista da Engenharia de software centrando-se no desenvolvimento alto-nível de aplicações onde, claro, se instanciam os conhecimentos aprendidos nas disciplinas que focam nos detalhes de funcionamento e implementação a baixo-nível.

1.1 Tópicos

- Sistemas de Controle de Versões,
- Arquitectura e Desenho de sistemas em grande escala,
- Persistência de objectos em sistemas relacionais,
- Apresentação via Web; concorrência e distribuição.

Capítulo 2

Controle de versões

Controle de versões p.5 • Os diferentes tipos de VCS p.5 • Git p.9 • Guião Git 1 p.23 • Guião Git 2 p.42 • Exercícios p.53

2.1 Controle de versões

O desenvolvimento de software é uma tarefa complexa. É realizada por equipas as vezes muito numerosas. Os Sistemas de Controle de Versões (VCS) têm como objetivo ajudar a coordenação do trabalho de desenvolvimento de software em equipas, grandes ou pequenas. O papel dos VCS no desenvolvimento de software é integrar o trabalho de várias pessoas. Um VCS pode guardar diversas versões de cada artefato (ficheiros) produzido pelos elementos da equipa, ao longo do tempo. Esta facilidade permite “regressar no tempo” e examinar o código em várias instantes do passado. Será útil por exemplo para identificar bugs e corrigilos.

O VCS trata de fundir várias versões dos ficheiros, modificados por vários membros da equipa. Este aspeto pode ser realizado automaticamente ou não conforme a presença de conflitos (vamos ver mais a frente o que é um conflito).

Um dos conceitos mais importantes no contexto dos VCS é o repositório. Um repositório é um conjunto de ficheiros que pertencem a um mesmo projeto. Para além dos ficheiros criados no âmbito do desenvolvimento do projeto, o repositório armazena toda a informação necessária à gestão das diferentes versões. Concretamente um repositório será uma pasta no seu computador onde estão armazenados os ficheiros do projeto e ficheiros geridos pelo VCS que contêm toda a informação necessária.

2.2 Os diferentes tipos de VCS

Existem vários tipos de VCS. Nesta secção vamos rever as principais características. Por exemplo, existem várias abordagem para a gestão da concorrência (quando o mesmo ficheiro está alterado por vários membros da equipa). Existe a abordagem pessimista e a a abordagem otimista.

Outra característica importante dos VCS diferencia os **VCS centralizados** e os **VCS distribuídos**.

VCS pessimistas p.6 • VCS otimistas p.6 • VCS centralizados p.7 • VCS distribuídos p.8

2.2.1 VCS pessimistas

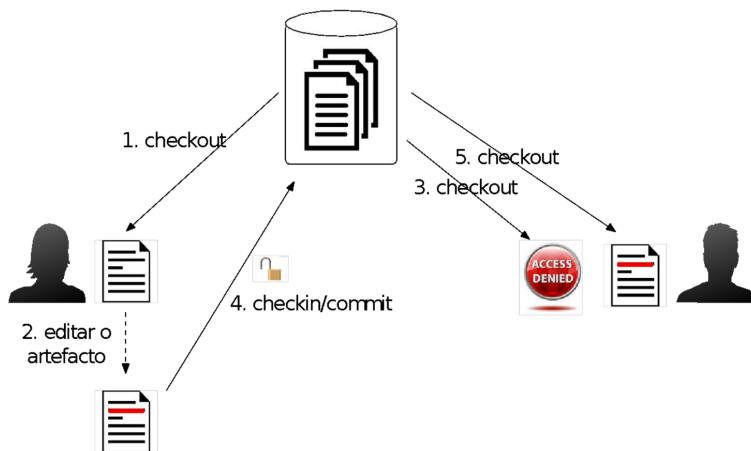
Um VCS pessimista **elimina o risco ligado a concorrência** impedindo que um ficheiro seja alterado por mais de um membro da equipa (em simultâneo). A ideia é a seguinte:

1. Um elemento da equipa pede o acesso a um ficheiro
2. Caso o ficheiro se encontre disponível, a operação de **checkout** é realizada, o ficheiro fica então indisponível para os outros membros.
3. O elemento realiza as modificações pretendidas
4. O elemento publica a nova versão do ficheiro (operação **checkin** ou **commit**).
5. O ficheiro está novamente disponível para os outros.

Note que se no passo 2 o ficheiro não está disponível, o elemento não pode modificá-lo, terá que esperar que fique disponível.

A operação “**checkout**” tem como objetivo tirar um ficheiro do repositório e o tornar disponível para modificações.

A operação “**commit**” tem como objetivo armazenar no repositório uma nova versão e avisar os colaboradores.



Uso típico de um VCS pessimista

Os VCS pessimistas foram dos primeiros a serem usados. Hoje já não são usados por causa do principal defeito : vários membros da equipa não podem trabalhar em paralelo alterando o mesmo ficheiro. Como exemplo deste tipo de VCS, existe o **RCS**.

2.2.2 VCS otimistas

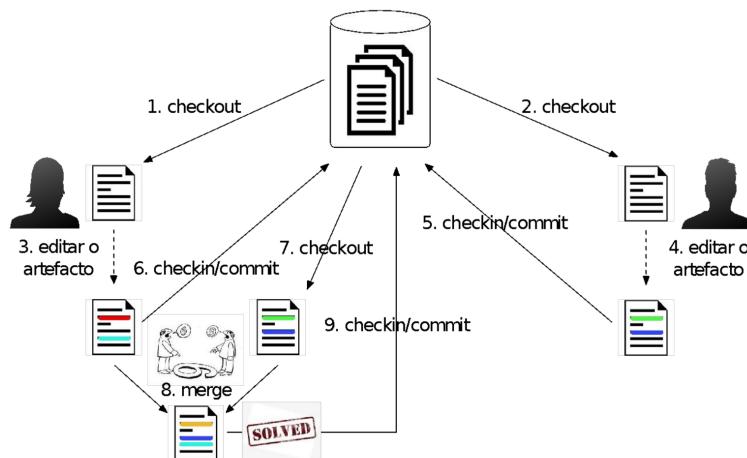
Enquanto os VCS pessimistas estão baseados no princípio que “as coisas podem correr mal” os VCS otimistas estão baseados no princípio que “vai correr bem” !

VCS otimistas permitem o acesso de um ficheiro a mais que um elemento da equipa. Cada membro da equipa pode obter do repositório uma cópia de um ficheiro (**checkout**) e modificá-lo. Deste modo **o trabalho sobre este ficheiro pode ser feito em paralelo** pelos membros da equipa.

Uma vez feitas as modificações desejadas, o elemento da equipa **publica o seu trabalho**. Isto significa que vai incluir esta nova versão no repositório (**commit**). Neste passo **é necessário “fundir” a nova versão** com a versão atual do repositório (outros elementos da equipa podem ter feito um **commit** entretanto).

Esta operação de fusão das versões (em inglês “**merge**”) é realizada por um algoritmo apropriado. Quando as diferenças entre as duas versões estão afastadas no ficheiros (por exemplo no início do ficheiro e no meio) o algoritmo consegue conciliar as duas versões para formar um nova que vai incluir as alterações efetuadas de ambos os lados.

Em **alguns casos** (por exemplo quando as diferenças ocorrem em linhas próximas ou até na mesma linha do ficheiro) o algoritmo não consegue decidir quais das duas opções escolher. Neste caso **o algoritmo deteta um conflito**. A situação deve então ser resolvida à mão. O conflito é assinalado e cabe ao membro da equipa que esta a querer publicar o seu trabalho gerar a nova versão eliminando o conflito.



Uso típico de um VCS otimista

1. Membro A faz *checkout* do ficheiro
2. Membro B faz *checkout* do ficheiro
3. A edita a sua cópia do ficheiro
4. B edita a sua cópia do ficheiro
5. B terminou as suas modificações, publica o seu trabalho no repositório (faz um *commit*)
6. A terminou as suas alterações faz um *commit*. Um **conflito** é detetado, **A deve resolver o conflito**.
7. A faz um *checkout* do ficheiro. Tem agora duas versões do mesmo ficheiro.
8. Faz o *merge* para fundir as duas versões. O conflito é resolvido.
9. Faz novamente um *commit* para publicar a nova versão.

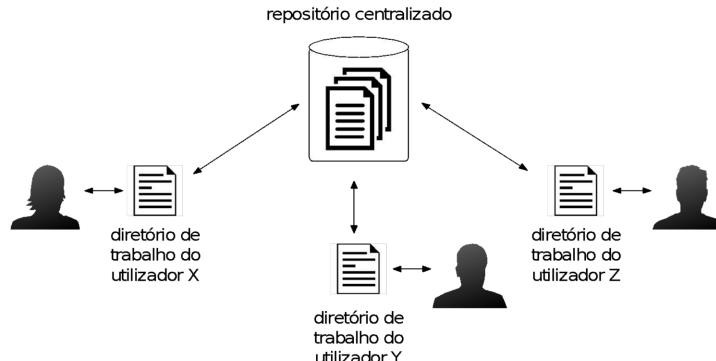
Existem vários VCS otimistas: [CSV](#), [SVN](#), [GIT](#), [Mercurial](#), [Bazaar](#).

2.2.3 VCS centralizados

VCS centralizados possuem um (e apenas um) **repositório central** onde são mantidas as **diversas versões**. Cada elemento de equipa extrai uma cópia (operação *checkout*) de um (ou mais) ficheiro(s). Faz as alterações que quer e publica-as (*commit*). Atualiza as alterações dos outros elementos na sua cópia local (*update*). Eventualmente terá de resolver conflitos.

VCS centralizados

Os VCS centralizados possuem apenas um repositório central.



Uso típico de um VCS centralizado.

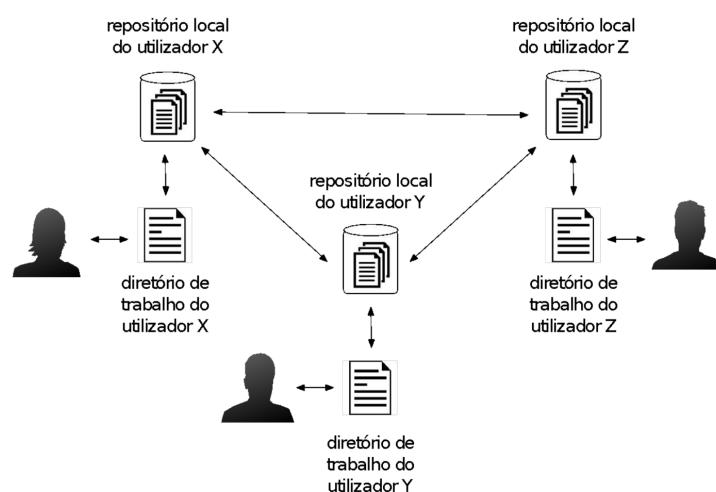
Cada membro da equipa tem um exemplar do ficheiro na sua área de trabalho.

Uma desvantagem dos VCS centralizados é que é necessário ter acesso ao repositório central. Este pode estar num servidor remoto; o acesso depende da ligação ao mesmo.

Com exemplos de VCS centralizados temos: **CSV**, **SVN**.

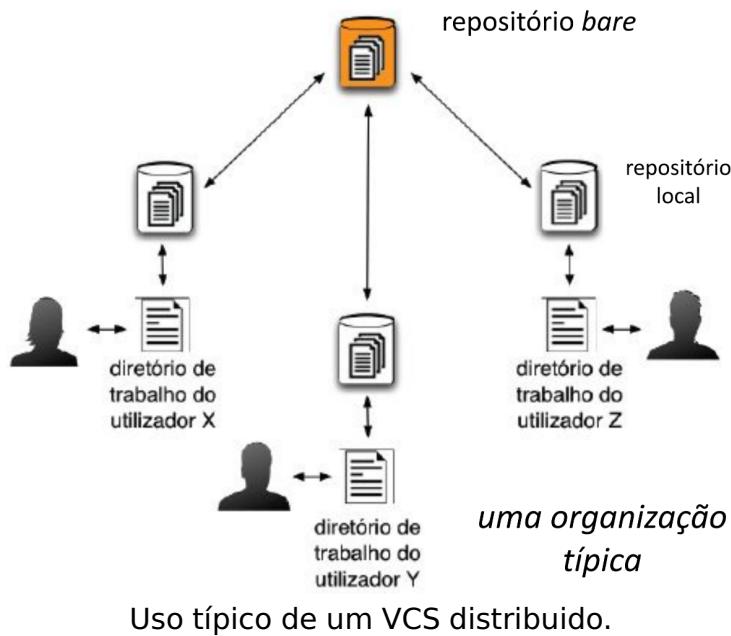
2.2.4 VCS distribuídos

Os VCS distribuídos não possuem um repositório central. Cada membro vai ter localmente no seu computador uma cópia do repositório. Significa que tem acesso localmente a toda história de desenvolvimento do projeto. Como é óbvio é necessário sincronizar os vários repositórios. A sincronização entre os vários repositórios seria rapidamente um pesadelo a medida que o número de membros da equipa aumenta.



No caso de um VCS distribuído cada membro da equipa tem uma cópia do repositório.

Por essa razão define-se geralmente um repositório especial chamado **repositório principal** ou **repositório bare** (a não confundir com o repositório central descrito na secção anterior). Cabe a cada membro sincronizar a sua cópia do repositório com o **repositório principal**.



2.3 Git

O Git é o VCS mais usado atualmente. Vamos usa-lo ao longo do semestre.

Conceitos p.9 • Fluxo de trabalho p.10 • Repositórios p.13 • Os comandos p.14

2.3.1 Conceitos

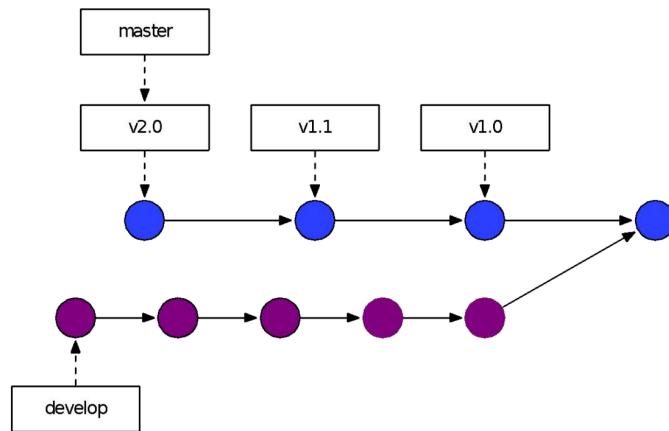
O **Git** é um VCS otimista e distribuído. Foi criado pelo Linus Torvalds para o desenvolvimento do kernel do Linux, é open-source. É muito flexível e permite usar vários fluxos de trabalho e modos de colaboração numa equipa.

A estrutura de dados usada para armazenar as versões do repositório é o **Direct Acyclic Graph (DAG)**.

Os principais conceitos associados ao Git são:

- **Repositório:** uma estrutura de dados que guarda metadados sobre ficheiros e pastas, incluindo um histórico de alterações, um conjunto de *committed objects* (ficheiros publicados), entre outras informações
- **Working Directory (pasta de trabalho):** é a pasta onde estão os ficheiros editados pelo programador.
- **Staging Area:** zona onde estão guardados os ficheiros que vão fazer parte do próximo *commit*. Também é chamado “índice”.

DAG: as diferentes versões do repositório estão armazenadas num grafo (DAG). A figura seguinte visualiza o histórico do repositório:



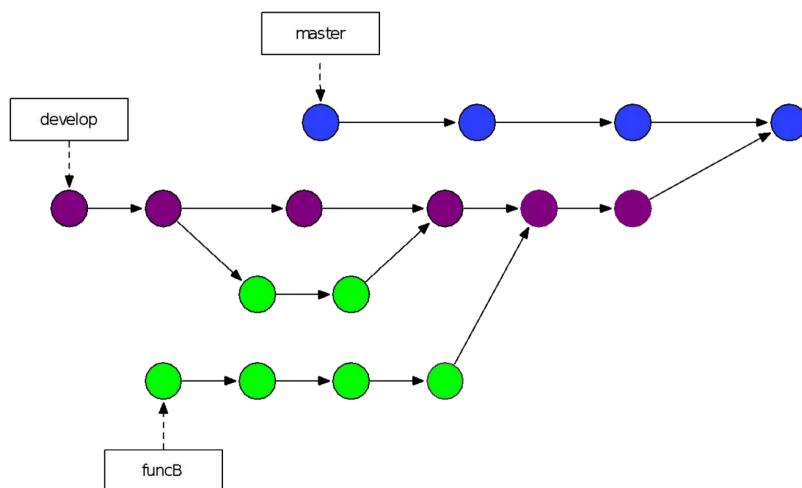
Grafo dos *commits* sucessivos.

Cada círculo corresponde a um *commit*. O *commit* mais antigo é o mais a direita. As **setas apontam para o *commit* anterior**. Os *commits* mais a esquerda são os mais recentes. As cores correspondem a **ramos** (a noção de ramo var estar esclarecida mais a frente). A um *commit* pode estar associada uma etiqueta.

2.3.2 Fluxo de trabalho

A ideia é permitir o **desenvolvimento concorrente** por uma equipa possivelmente grande de programadores. Existem várias maneiras de organizar o trabalho com base o Git. Uma abordagem consiste em manter uma sequência de *commits* que contêm a versão mais acabada da aplicação, em cada instante. Em paralelo, várias atividades podem ser desenvolvidas: correção de bugs, adição de novas features.

Para facilitar este tipo de organização **o Git permite a criação de ramos**. Um ramo é um desvio relativo a outro ramo. Neste desvio vamos registar os *commits* que correspondem à resolução de uma tarefa (correcção de um bug, adição de uma feature). Uma vez concluída uma tarefa, o ramo usado para esta tarefa é **fundido com o ramo de origem** e (eventualmente) apagado. No exemplo seguinte, em azul temos o ramo que contém a versão mais acabada da aplicação (versão oficial). Este ramo é chamado **master** ou **main**. Note que o primeiro *commit* (*commit* inicial) pertence a este ramo.



Exemplo do efeito do fluxo de trabalho no repositório.

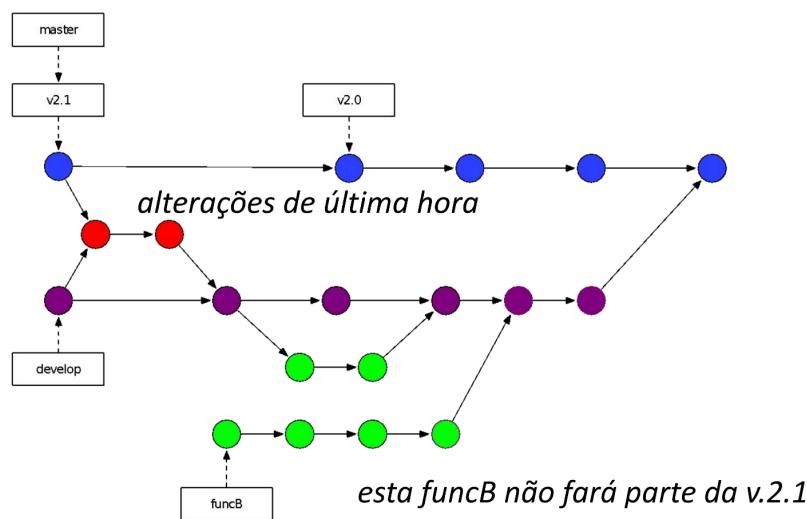
Os círculos roxos correspondem a o ramo de desenvolvimento. É neste ramo que a aplicação é desenvolvida. Isto significa que as tarefas de correção de bug vão ser realizadas em sub-ramos deste ramo. É o exemplo dos ramos verdes. O primeiro (de cima para baixo) tem dois

commits. Ao fim desses dois *commits* o objetivo foi atingido e o resultado é fundido com o ramo “develop”. No caso do segundo ramo verde (com 4 commits), contém o desenvolvimento da funcionalidade “funcB” como ainda não está finalizada, (ainda) não é fundido com o ramo “develop”.

Desenvolvimento de funcionalidades Para o desenvolvimento de uma nova funcionalidade, a equipa cria um novo ramo a partir do ramo *develop* e todas as publicações que ocorrem durante o desenvolvimento desta funcionalidade ficam associado a este novo ramo. Quando o desenvolvimento, testes e controlo de qualidade está concluído para esta funcionalidade, é feita a junção deste ramo ao *develop* que contém as outras funcionalidades já desenvolvidas e que passaram pelo mesmo controle de qualidade. Desta forma:

1. as funcionalidades que estão a ser desenvolvidas não interferem entre si nem com outras já desenvolvidas; e
 2. caso se atinja a data para a produção de uma nova versão da aplicação, a equipa pode de forma fácil e clara identificar quais as funcionalidades prontas para integrar a nova versão e quais as que ainda estão em desenvolvimento e devem ser adiadas para integração numa versão futura da aplicação.

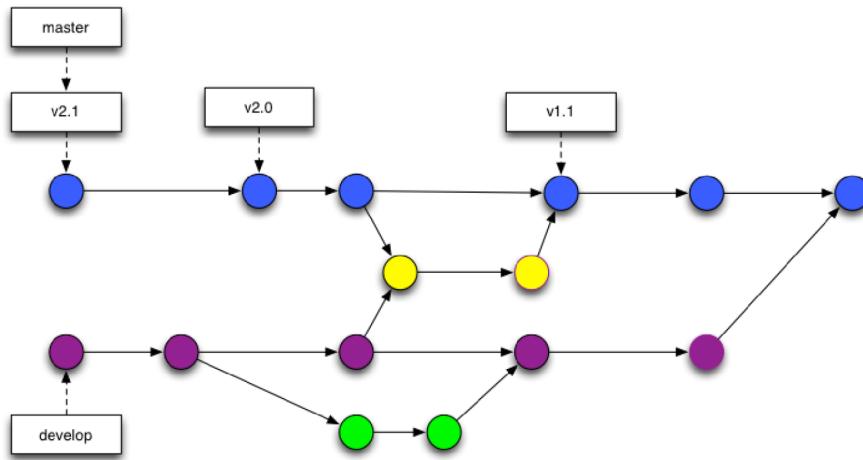
Periodicamente o ramo “*develop*” é fundido com o ramo “*master*” de modo a publicar as novas funcionalidades (ou correções de bugs):



Exemplo do efeito do fluxo de trabalho no repositório.

Neste exemplo, a junção do ramo verde ao ramo “*develop*” corresponde à conclusão do desenvolvimento de uma nova funcionalidade. Foi criado um novo ramo (encarnado) para testar a aplicação, efetuar as últimas correções antes de juntar ao ramo principal. A versão resultante foi fundida com o ramo “*master*” e o ramo “*develop*”. Notem que ambos os ramos verdes derivam do ramo *develop*. O segundo ramo verde tem quatro publicações e a funcionalidade, cuja ramo tem o nome *funcB*, ainda está em desenvolvimento.

Correção de faltas no software Para concluir a nossa discussão sobre fluxos de trabalho vamos analisar agora a situação de uma correção de uma falta detetada em software já em produção. Por exemplo, um cliente deteta uma falha numa versão da aplicação (não necessariamente na última versão) e é necessário proceder à sua correção.



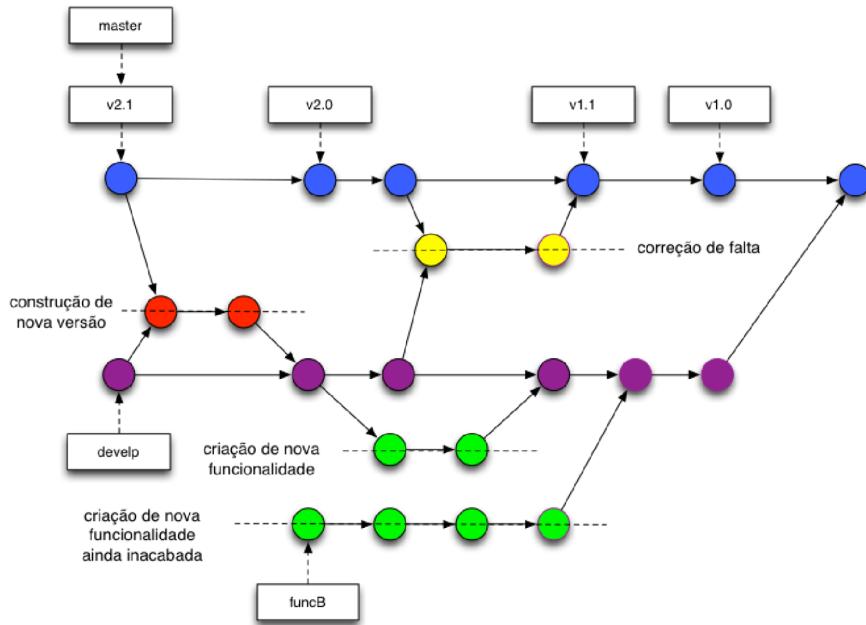
Correção de faltas detetadas no software já em produção.

A Figura ilustra o processo de correção de um problema detetado na versão 1.1. Esta é a única situação em que é permitido criar um ramo a partir do ramo master. O fluxo de trabalho é o seguinte:

1. extrair a versão onde a falha se manifestou, no presente caso na versão 1.1;
 2. criar um novo ramo, neste caso representado a amarelo;
 3. corrigir a falta;
 4. publicar as alterações para o ramo criado;
 5. juntar este ramo ao master, mantendo o ramo *master* atualizado com o código que está em produção;
 6. juntar o ramo de correção também ao ramo *develop* para que (com sorte) a falta não se volte a manifestar em futuras versões da aplicação.

A figura ilustra um cenário em que a falha ocorreu antes da versão 2.0 ter sido produzida. No caso em que a versão 2.0 e 2.1 já tivessem sido produzidas, por exemplo, a operação de junção iria acontecer a seguir às publicações destas versões e poderia originar diversas situações de conflito que, eventualmente, resultariam em horas de trabalho extra para as integrar na versão atual da aplicação. Quanto mais tarde ocorrer a junção de ramos mais probabilidade existe de aparecerem conflitos. Por esta razão é que os membros das equipas são encorajados a juntar o seu trabalho com o dos restantes membros o mais cedo possível, minimizando assim o trabalho de todos. Há ferramentas que ajudam nesta integração contínua de trabalho, mas que está para além do âmbito desta disciplina.

A figura seguinte sumaria os fluxos de trabalho descritos nesta secção, que constituem uma possível forma de tirar partido de ferramentas de VCS aplicadas ao desenvolvimento concorrente de software por uma equipa. Contudo, os fluxos de trabalho devem ser adaptados às necessidades das equipas e dos projetos e devem ser simplificados ou até mais detalhados dependendo da complexidade do projeto a ser desenvolvido e do tamanho da equipa.



Resumo de todos os fluxos de trabalho ilustrados.

2.3.3 Repositórios

Estamos na altura de voltar à noção de repositório. Como vimos, cada membro da equipa tem um exemplar do repositório e vimos que havia **um repositório principal** com o qual todos sincronizam o seu exemplar do repositório.

Na prática o repositório principal é muitas vezes colocado num servidor acessível via internet. É chamado o **repositório remoto** ou no contexto do Git, o **origin**.

Quando um repositório é criado tem automaticamente um ramo. Este ramo é chamado conforme a sua configuração **master** ou **main**. Existem duas maneiras de criar um repositório:

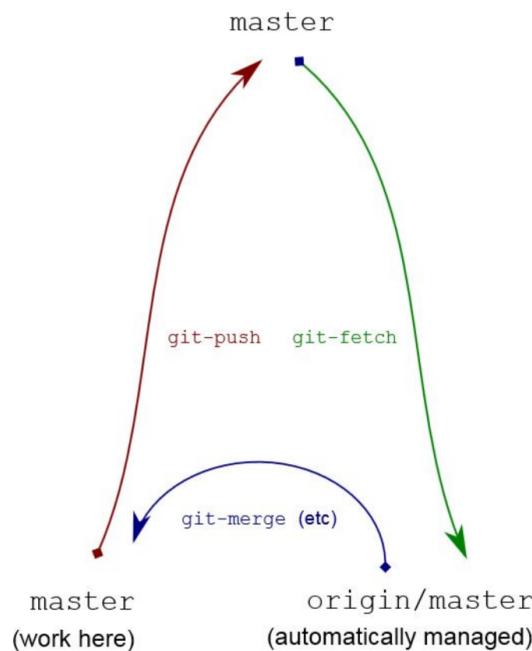
- criando localmente um repositório vazio, usando o comando `git init`
- criando uma cópia local de um repositório (remoto) existente, usando o comando `git clone`

No primeiro caso o repositório vai conter um ramo master (ou main) e no segundo vai conter todos os ramos que foram até a altura criados pelas pessoas que contribuiram ao repositório. Vai portanto existir um ramo **master** local e outro no repositório remoto. O Git, para facilitar a fusão dos repósitórios vai manter (automaticamente, não será visível a não ser via comandos Git) uma versão local do repositório remoto. No caso mais simples onde existe apenas o ramo master, vamos portanto ter:

- um **repositório local com o ramo master**
- um **repositório remoto** (é o repositório principal) chamado **origin** onde constará um **ramo master**
- uma **cópia local do repositório remoto** que será designada por **origin/master**

Vários comandos permitem sincronizar o repositório local com os outros: `fetch`, `push`, `merge`.

Remote repo origin



Local repo

Exemplo do efeito do fluxo de trabalho no repositório.

2.3.4 Os comandos

Embora existem várias interfaces gráficas (*front-ends*) para usar o git, vamos numa primeira fase, usar o Git através dos comandos teclados no terminal. **O uso inadequado dos comandos do Git pode levar a uma perca do trabalho desenvolvido** e por essa razão **não é de todo aconselhado usa-lo via uma interface gráfica antes de dominar completamente todo o processo.**

O Git possui muitos comandos, cada comando tem numerosas opções. Segue nesta secção uma lista dos principais comandos. Para experimentar os principais comandos num contexto de fluxo de trabalho, deve realizar os guiões p. 23 e p. 42.

Criação do repositório p.14 • Obter informações p.15 • Working directory, Stage area p.18 • Repositório remoto p.19 • Trabalhar com ramos p.20
 • Merge e rebase p.21 • Configuração p.23

Criação do repositório

- `git init`: cria um repositório vazio na pasta corrente.

```
> mkdir meuProjeto
> cd meuProjeto
> git init
Initialized empty Git repository in /tmp/meuProjeto/.git/
```

O Git criou uma pasta `.git` (escondida) onde vai colocar toda a informação necessária para a gestão do seu repositório. Pode espreitar o conteúdo mas não deve alterar nada !

Conforme a sua configuração o Git pode fornecer uma resposta um pouco mais longa:

```
> mkdir meuProjeto
> cd meuProjeto
> git init
hint: Using 'master' as the name for the initial branch. This default branch name
hint: is subject to change. To configure the initial branch name to use in all
hint: of your new repositories, which will suppress this warning, call:
hint:
hint:   git config -global init.defaultBranch <name>
hint:
hint: Names commonly chosen instead of 'master' are 'main', 'trunk' and
hint: 'development'. The just-created branch can be renamed via this command:
hint:
hint:   git branch -m <name>
Initialized empty Git repository in /tmp/meuProjeto/.git/
```

As linhas iniciadas com `hint:` contêm dicas. Vale a pena ler. No caso presente o Git indica que o ramo inicial do repositório é chamado “master” mas explica como fazer para alterá-lo. É possível configurar o Git de forma que use sempre o mesmo nome para o ramo inicial (ver comando `git config`).

Nome do ramo inicial

Embora seja possível escolher um nome arbitrário para o ramo inicial, no contexto da disciplina devem escolher `master` ou `main`. Caso escolha “`master`”, o repositório não poderá possuir um ramo `main` e vice versa.

- `git clone <url>`: cria uma cópia local de um repositório remoto. O `url` designa o repositório remoto. Pode ser de tipo `https` (por exemplo: `https://github.com/manel1234/MeuProjeto.git`) ou `ssh` (`git@github.com:manel1234/MeuProjeto.git`). O repositório remoto pode estar alojado num servidor de uma organização ou num site especializado (`github.com`, `gitlab.com` ou no DI: `git.alunos.di.fc.ul.pt`). Este último é apenas acessível via VPN. O acesso a um repositório remoto cujo endereço é de tipo `https` necessita geralmente a introdução de um `username` e de uma `password`, cada vez que usamos um comando que comunica com o servidor. É portanto vantajoso usar um endereço `ssh` que caso o seu computador seja configurado com uma chave pública e esta seja introduzida no servidor, evita a introdução das credências. **As instruções para configurar o seu repositório/área estão no Guia git, p. 29.**

Pasta do repositório

A pasta onde vai criar o seu repositório (usando `git init` ou `git clone`) não pode ser uma sub-pasta de outro repositório. Não deve criar um repositório dentro de outro.

Obter informações

Para obter informações acerca de um repositório existem vários comandos:

- `git status`: este comando mostra um resumo do estado do repositório. É um comando muito importante que é necessário usar antes de realizar uma operação que pode modificar o seu repositório, eventualmente de uma forma não desejada ! O comando indica o

ramo corrente, o estado relativamente ao exemplar local do ramo remoto (`origin/master`) se houver *commits* pendentes.

Por exemplo se o repositório for acabado de criar vai mostrar:

```
> git status
```

On branch master

No commits yet

nothing to commit (create/copy files and use "git add" to track)

O ramo inicial é `master`, nenhum *commit* foi feito, nenhum ficheiro foi alterado.

Após ter modificado o ficheiro `README.md` o comando mostra:

```
> git status
```

Your branch is up to date with 'origin/main'.

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)
 (use "git restore <file>..." to discard changes in working directory)
 modified: README.md

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

Deve entender as indicações dadas pelo Git:

- Your branch is up to date with 'origin/main'.: Nenhum *commit* foi efetuado (desde o último comando `git clone`, `git pull` ou `git fetch`) logo o ramo corrente está sincronizado com a cópia local do ramo `master` no servidor remoto (`origin/master`).
 - Changes not staged for commit : Nenhuma alteração foi colocada na *stage area* (um ficheiro foi modificado mas não foi adicionado (ver `git add`))
 - (use: o Git faz duas sugestões: adicionar as alterações ao *stage area* (`git add`) ou ignorar as modificações (`git restore`))
 - modified: `README.md`: o Git indica quais ficheiros foram modificados (a vermelho).
 - no changes added to commit (use... : Finalmente o Git faz um resumo da situação (neste caso é repetitivo)).
- **git log**: Este comando lista os *commits* efetuados. Existem muitos parâmetros para controlar as informações incluídas e o tipo de visualização. Por omissão o comando é bastante verboso:

```
> git log
```

```
commit 98d189b94f710c92597369611547aa0f68645b23 (HEAD -> main, origin/main, origin/HEAD)
Merge: 6fb2278 30e70dc
```

```
Author: Archibaldo Cunha <ac@fcul.pt>
```

```
Date: Sat Feb 25 11:47:07 2023 +0000
```

```
conflict resolved
```

```
commit 30e70dc8dc96846af256e962ff3497966e8f119a
```

```
Author: Archibaldo Cunha <ac@fcul.pt>
```

```
Date: Sat Feb 25 10:58:54 2023 +0000
```

```
my name in README
```

```

commit 6fb227842b74a694bfffad7185efc3f409128f89
Merge: c1bf714 29551a7
Author: Virgolino Madeira <vm@fcul.pt>
Date:   Sat Feb 25 10:54:38 2023 +0000

    Merge branch 'main' of github.com:virgolinomad/test

commit c1bf714a0ba1ba06a7bf05fbe648c1481df1d23c
Author: Virgolino Madeira <vm@fcul.pt>
Date:   Sat Feb 25 10:48:33 2023 +0000

    Added README.md.

...

```

Usando esta combinação de parâmetros: git log --decorate --color=always --pretty=tformat:"%C(auto)%h %ce %s %d--graph

```
> git log --decorate --color=always --pretty=tformat:"%C(auto)%h %ce %s %d" --graph
```

```

*   98d189b ac@fcul.pt conflict resolved  (HEAD -> main, origin/main, origin/HEAD)
| \
| * 30e70dc ac@fcul.pt my name in README
* | 6fb2278 vm@fcul.pt Merge branch 'main' of github.com:virgolinomad/test
| \
| * 29551a7 ac@fcul.pt Merge branch 'main' of github.com:virgolinomad/test
| |
| * \  21e9c9a ac@fcul.pt Merge branch 'main' of github.com:virgolinomad/test
| | \
| * | | 61c1a8e ac@fcul.pt library.py created
* | | | c1bf714 vm@fcul.pt Added README.md.
...

```

Ou então para focar nas datas dos commits:

```
> git log --decorate --color=always --pretty=tformat:"%C(auto) %h %ad %ce %s %d"
```

```

98d189b Sat Feb 25 11:47:07 2023 +0000 ac@fcul.pt conflict resolved  (HEAD -> main, ori
30e70dc Sat Feb 25 10:58:54 2023 +0000 ac@fcul.pt my name in README
6fb2278 Sat Feb 25 10:54:38 2023 +0000 vm@fcul.pt Merge branch 'main' of github.com:vir
c1bf714 Sat Feb 25 10:48:33 2023 +0000 vm@fcul.pt Added README.md.
...

```

- **git branch:** este comando lista todos os ramos conhecidos. Indica o ramo corrente com um “*”. Exemplo:

```
> git branch
```

```

archibaldo
main
ramo1
ramo2
* ramo3

```

- **git show**: este comando é usado para exibir informações detalhadas sobre um *commit* específico num repositório Git. Fornece uma visualização detalhada das alterações e metadata associados a um *commit* específico. Aqui exemplos de uso do comando **git show**:
 - **git show abc123** mostra os detalhes do commit `abc123`.
 - **git show --since="2 weeks ago--author="John Doe"** mostra os commits que satisfazem os critérios indicados.
 - **git show origin/ramo123** mostra os detalhes do primeiro *commit* do ramo `ramo123` no repositório remoto.
 - **git show 420d35e:README.md** mostra o conteúdo do ficheiro `README.md` como está no *commit* `420d35e`:
- **git diff** sem opções o comando mostra as diferenças entre os ficheiros que estão no staging area e os ficheiros correspondentes na área de trabalho.
 - **git diff <ficheiro>**: idem mas apenas para um ficheiro.
 - **git diff <commit>**: mostra as diferenças entre a área de trabalho e o commit indicado.
 - **git diff --staged**: mostra as diferenças entre o último commit e a staging area.
 - **git diff <commit>..<commit>**: mostra as diferenças entre os dois commits.
 - **git diff <ramoA>..<ramoB>**: mostra as diferenças entre dois ramos.
 - etc...

Working directory, Stage area

Para alterar os ficheiros presentes na área de trabalho temos os comandos:

- **git checkout**
 - **git checkout <ramoA>** torna o `ramoA` o ramo corrente.
 - **git checkout <commit>** Move o **HEAD** para o commit especificado, criando um estado “**detached HEAD**”. Isso permite visualizar o estado do repositório em um *commit* específico. Cuidado! É uma situação perigosa, não faça alterações num *commit* que pertence ao passado!
 - **git checkout <commit> -- <file>**: Restaura o ficheiro especificado para o estado no *commit* indicado. É útil quando deseja recuperar uma versão anterior de um arquivo específico. Uma alternativa é usar : **git show <commit>:<file>** que apenas mostra aquela versão do ficheiro. Se deseja ficar com a versão do commit pode usar : **git show <commit>:<file> > <file>**. Neste comando, `<file>` representa o caminho da raíz do repositório até o ficheiro.
 - **git checkout .** descarta todas as modificações feitas na área de trabalho.
- **git add** adiciona um ou mais ficheiros ao índice. **git add .** adiciona todos os ficheiros alterados.
- **git restore** Existe variante deste comando:
 - **git restore <file>** Restaura o ficheiro especificado para o estado no último commit, descartando quaisquer alterações não confirmadas na área de trabalho.
 - **git restore --staged <file>** Move o ficheiro especificado da área de staging (índice) de volta para a área de trabalho, cancelando o efeito de **git add**. As alterações não confirmadas no diretório de trabalho não são afetadas.
 - **git restore --source=:** Restaura todos os ficheiros no diretório de trabalho para o estado no último commit.

Repositório remoto

- **git fetch**: o comando vai buscar o repositório remoto e coloca-o numa área especial (fica disponível localmente). A área de trabalho não é alterada. Para trabalhar na versão que estava no remoto ainda tem de usar o comando **git merge** para fundir com a sua versão. O comando **git merge** pode originar situações de conflito.
 - **git fetch <remote>** vai buscar o repositório do remoto indicado. Caso não indique o remoto “origin” é usado.
 - **git fetch <remote> <branch>** idem mas vai buscar apenas o ramo indicado.
 - **git fetch --dry-run** apenas mostra o que seria copiado para o repositório local, sem efetuar a operação.
- **git pull** este comando é equivalente a **git fetch** seguido de **git merge**. Os ficheiros do repositório remoto ficam na área de trabalho (se não houver conflito). o comando **git pull** efetua duas operações em sequência: **git fetch** (que copia o repositório remoto para o local) e **git merge**. Na verdade, depende da sua configuração. Pode especificar o seu método preferido para juntar as versões (merge vs rebase). Quando o seu repositório não está configurado, o Git vai produzir uma mensagem semelhante a :

```
hint: You have divergent branches and need to specify how to reconcile them.
hint: You can do so by running one of the following commands sometime before
hint: your next pull:
hint:
hint:   git config pull.rebase false    # merge
hint:   git config pull.rebase true     # rebase
hint:   git config pull.ff only       # fast-forward only
hint:
hint: You can replace "git config" with "git config --global" to set a default
hint: preference for all repositories. You can also pass --rebase, --no-rebase,
hint: or --ff-only on the command line to override the configured default per
hint: invocation.
fatal: Need to specify how to reconcile divergent
```

O que é que aconteceu ? O Git não sabe como juntar as versões (merge vs rebase) portanto pergunta o que deve fazer. Quando receber esta mensagem tem de tomar uma decisão. Uma solução é configurar o seu repositório com:

```
git config pull.rebase false
```

Deste modo o método “merge” será usado. Pode voltar a fazer o **git pull** a seguir, não haverá erro. O comando **git rebase** está documentado p. 21.

- **git push <remoto> <ramo>** Uma vez feito um *commit*, este comando publica o seu repositório (o ramo indicado) para o remoto. Uma vez que o repositório está configurado e tem apenas um repositório remoto, pode usar simplesmente **git push** para empurrar o ramo corrente para o repositório remoto. Eventualmente a publicação pode ser rejeitada (se o repositório remoto ter sido alterado entretanto):

```
> git push
```

```
To github.com:thibaultlanglois/test.git
! [rejected]      ramo1 -> ramo1 (fetch first)
error: failed to push some refs to 'github.com:thibaultlanglois/test.git'
hint: Updates were rejected because the remote contains work that you do not
hint: have locally. This is usually caused by another repository pushing to
hint: the same ref. If you want to integrate the remote changes, use
hint: 'git pull' before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

- **git ls-remote** lista todos os ramos no repositório remoto. Por exemplo:

```
> git ls-remote
```

```
From git@github.com:thibaultlanglois/test.git
20144cbbb24de112f1275980e9dae3793e9b8697 HEAD
d11a9c557486c152f13ac4d9af16a6632dd75485 refs/heads/archibaldo
80627198c89b7405e0bc43c55cf64205030fb46c refs/heads/f23
20144cbbb24de112f1275980e9dae3793e9b8697 refs/heads/main
d67dbde1e73333fb03c34e062dbcbbdca6a35327 refs/heads/ramo1
01f7f73884f9579a86789c14b0fe6a7455fc7193 refs/heads/ramo2
ac13334b98681dbd331f25ef91bde5b77ad9322d refs/heads/ramo3
```

- **git remote** Este comando permite obter informações acerca do(s) repositórios remotos.

- **git remote -v** lista os repositórios remotos (com a URL).
- **git remote show <name>** idem mas para um remoto específico, mostrando os ramos presentes.
- **git remote add <name> <url>** adiciona um remoto ao repositório. Pode ser útil para corrigir o endereço de um remoto (passar do protocolo http para ssh).
- **git remote set-url origin <url>** é usado para modificar o endereço do repositório remoto. Pode ser útil se por exemplo está configurado com um endereço HTTPS e que passar a usar um endereço SSH. Exemplo: **git remote set-url origin git@gitlab.alunos.di.fc.ul.pt:**

Trabalhar com ramos

- **git branch**: este comando lista todos os ramos conhecidos. Indica o ramo corrente com um “*”. Exemplo:

```
> git branch
```

```
archibaldo
main
ramo1
ramo2
* ramo3
```

git branch <name> cria um novo ramo com o nome indicado.

git branch -r lista todos os ramos presentes no repositório remoto.

- **git checkout** ver p. 18.

- **git merge**

Merge e rebase

Os comandos `git merge` e `git rebase` são usados para fundir duas versões do repositório. Têm um comportamento diferente que vamos estudar nesta secção.

Merge

O comando `git merge` compara as duas versões do repositório e tenta produzir uma nova versão. Quando as diferenças entre as duas versões dizem respeito a ficheiros diferentes, não há dificuldade, o comando cria um novo *commit* que contém as versões mais recentes dos dois ficheiros.

Quando o mesmo ficheiro sofreu alterações em ambas as versões do repositório, pode haver duas situações :

- As partes modificadas estão em sitios distintos (e distantes) no ficheiro, neste caso uma nova versão do ficheiro com ambas as modificações é produzida.
- As partes modificadas estão incompatíveis, o algoritmo deteta um conflito e gera uma versão onde ambas as opções são presentes:

```
> git merge
```

```
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
Automatic merge failed; fix conflicts and then commit the result.
```

O problema está assinalado no ficheiro:

```
> cat README.md
```

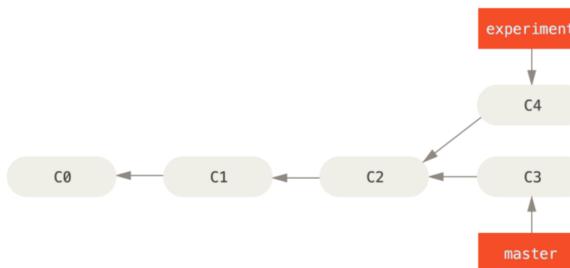
```
# testmais uma linha
modification
<<<<< HEAD
Esta linha foi escrita pelo utilizador A
=====
Esta linha foi acrescentada pelo utilizador B
>>>>> refs/remotes/origin/ramo1
```

A resolução do conflito consiste em editar o ficheiro e altera-lo para a opção desejada. Entre as linhas `<<<<< HEAD` e `=====` está a minha versão enquanto entre as linhas `=====` e `>>>>> refs/remotes/origin/ramo1` está a linha do repositório remoto.

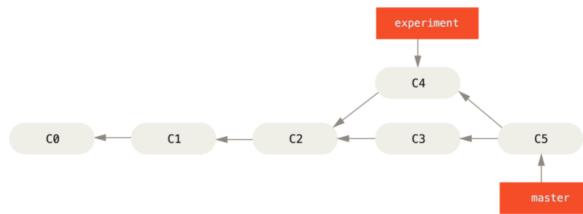
Rebase

O comando `git rebase` é usado para reorganizar ou reescrever a história dos *commits* de um ramo. É feito pegando nos *commits* de um ramo e aplicando-os no topo de outro ramo.

No `merge` é realizada a fusão de dois ramos analisando as diferenças entre os dois últimos ramos e o seu antepassado comum:



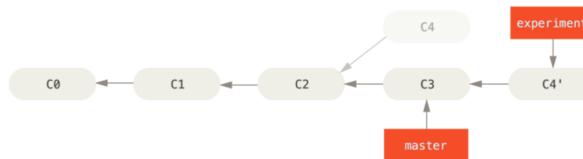
Antes de realizar o *merge*.



Após a realização do *merge*.

Se fazemos um `git push origin master` a seguir ao *merge* (`C5`) o ramo `Experiment` não vai para o remoto mas o grafo com os commits `C3` e `C4` vão. O comando `git rebase` permite evitar “sujar” o repositório remoto com muitas ramificações que têm apenas um significado local. O comando `git rebase` efetue as alterações contidas em `C4` no *commit* `C3` resultando num novo *commit* `C4'`:

```
$ git checkout experiment
$ git rebase master
```

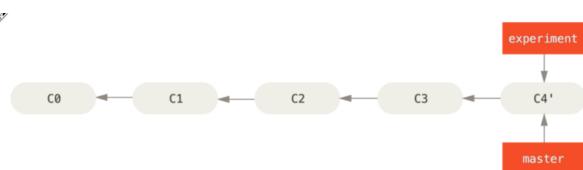


Após a realização do *rebase*.

- O `git rebase` deve ser feito apenas a ramos do nosso repositório local, não devemos fazer *rebasing* a ramos puxados do repositório remoto, pois isso poderá produzir resultados confusos e de difícil gestão.
- O `git merge` serve na perspetiva que a história dos *commits* é o registo do que realmente aconteceu, até ao mais pequeno detalhe
- O `git rebase` tem a perspetiva que a história é como o projeto foi feito, eliminando detalhes irrelevantes que apenas iriam dificultar quem tem de perceber como o projeto foi desenvolvido.

A seguir podemos ir para o ramo `master` e fazer o *merge* com o ramo `experiment`:

```
$ git checkout master
$ git merge experiment
```



O ramo `C4` é apagado da história das versões. A versão `C4'` é igual à `C5` da figura que mostra o resultado do *merge*. A única diferença é que se produz uma história mais simples.

Exemplo:

1. No início do dia fazemos um `git fetch` do repositório remoto (estamos a trabalhar em equipa) e um `git checkout` (do ramo onde estamos a trabalhar),
2. Trabalhamos um pouco e, no fim, fazemos um *commit*,
3. Publicamos (`git push`) o novo trabalho para o repositório remoto...
4. ... mas dá um erro ! (clássico):

```
[rejected]           master -> master (fetch first)
error: failed to push some refs to '.../remote/'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

Um colega publicou um *commit* antes logo o nosso foi rejeitado.

Vamos buscar a nova versão do remoto: `git fetch origin master`. Para integrar esta nova versão vamos usar git remote em vez de git merge:

```
> git rebase origin/master
```

Agora o nosso ramo contém os commits que estavam no remoto, inseridos antes dos nossos. É possível que a operação genere conflitos ! Finalemente podemos publicar o nosso trabalho:

```
> git push origin master
```

Configuração

- `git config`: ver guiões.

2.4 Guião Git 1

Descrição do git p.[23](#) • Antes de começar p.[24](#) • Introdução p.[24](#) • Os comandos p.[24](#) • A interação com um repositório remoto p.[29](#) • Integração com o Eclipse p.[34](#)

2.4.1 Descrição do git

O Git é um sistema de controle de versões que permite gerir um projecto ou conjunto de ficheiros de desenvolvimento de código à medida que estes vão sendo alterados ao longo do tempo permitindo rever e comparar diferentes versões entre si. Permite também reverter um ficheiro ou projetos inteiros para um estado anterior.

Um sistema de controle de versões como o Git é fundamental no desenvolvimento de projetos cooperativos em que diversas pessoas podem contribuir simultaneamente, através da criação de novos ficheiros ou da edição de ficheiros já existentes.

O Git organiza os ficheiros numa estrutura de dados designada por repositório. O Git funciona com base em repositórios distribuídos; ao contrário de outros sistemas de gestão de versões, não existe no Git a noção de repositório central, i.e., um único local onde se guardam todas as versões dos ficheiros. Cada repositório distribuído contém todo o historial de todos os ficheiros. A maior parte dos comandos do Git precisam apenas de recursos e arquivos locais, não sendo geralmente necessária informação que esteja guardada noutro computador.

Após efetuar as alterações desejadas num diretório de trabalho, o utilizador regista-las no repositório Git local e pode depois publicá-las num repositório Git remoto, caso este exista. Note que pela natureza distribuída da arquitetura Git, é possível vários utilizadores alterarem a sua cópia local do mesmo ficheiro criando potenciais conflitos que têm de ser resolvidos como veremos mais adiante.

Nos exemplos deste guião vamos usar a linha de comandos e no final veremos a integração com o Eclipse.

2.4.2 Antes de começar

Antes de mais, pressupõe-se que tem o Git instalado na sua máquina. Caso não tenha, sugere-se que procure online a forma de instalar a versão do sistema Git adequada ao seu sistema operativo.

Na primeira utilização do Git é necessário configurar o email e o nome do utilizador para que os comandos posteriores identifiquem quem fez as alterações ao repositório:

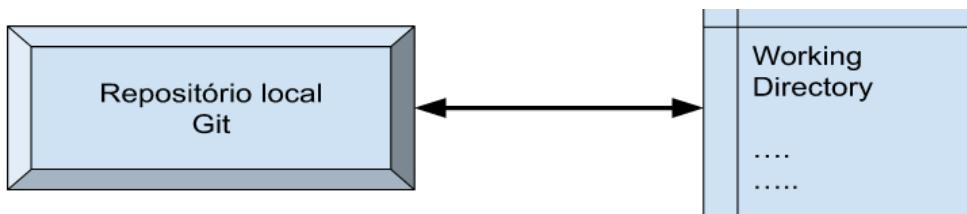
```
> git config --global user.email "endereço de e-mail"
> git config --global user.name "nome do utilizador"
```

Para verificar a sua configuração pode usar o comando:

```
> git config --list
```

2.4.3 Introdução

Para simplificar a introdução vamos supor, a título de exemplo, que queremos manter apenas um repositório local na nossa máquina e trabalhamos sobre os ficheiros do projecto no Eclipse. Ou seja, na nossa máquina temos armazenados num diretório (o **repositório local**) todas as alterações que fomos produzindo ao nosso projecto e que o nosso projecto é elaborado no Eclipse, numa área de trabalho que contém uma cópia dos ficheiros (e que, para um sistema de controle de versões, se denomina **working directory**).



2.4.4 Os comandos

A primeira questão que se coloca é: “Como é que criamos um repositório local, isto é, como é que criamos uma estrutura que contém as várias versões dos ficheiros que fomos alterando?” O primeiro passo é a criação de um diretório e de uma instrução git que o sinaliza como contendo a estrutura de repositório:

```
> git init
```

Esta instrução cria dentro do directório corrente, um subdirectório `.git` que contém toda a informação do estado do repositório. No início este repositório não terá ficheiros. Note que este diretório é gerido pelo Git e portanto não é aconselhável alterar o seu conteúdo. Experimente o seguinte conjunto de instruções:

```
> mkdir experienciaGit
> cd experienciaGit
> git init
Initialized empty Git repository in /home/manel/CSS/experienciaGit/.git/
> ls -al
total 0
drwxr-xr-x  3 manel  staff  102 Feb 10 08:40 .
drwxr-xr-x  5 manel  staff  170 Feb 10 08:40 ..
drwxr-xr-x 10 manel  staff  340 Feb 10 08:40 .git
```

Esta última nota leva à segunda questão: Como é que o Git sabe que ficheiros devem estar contidos

no repositório? Teremos de ser nós a dizer-lhe através do comando `add`:

```
> git add nome_do_ficheiro
```

Vejamos um exemplo concreto. Criamos previamente um ficheiro (neste caso vamos usar o comando `echo` do Linux e redirecionar o output para o ficheiro `hello.txt` com o símbolo `>`) e vamos informar o Git que este ficheiro deve estar contido no repositório:

```
> echo "hello" > hello.txt
> git add hello.txt
```

Note que o comando `add` apenas informa que o ficheiro é suposto ser tratado pelo repositório. Para efetivar a colocação de uma versão (a primeira ou uma versão atualizada) do ficheiro no repositório é necessário publicar essas alterações no repositório. Esta acção é concretizada com o comando `commit` (no qual se deve usar uma mensagem descritiva, como veremos adiante):

```
> git commit -m "mensagem elucidativa"
```

Execute a seguinte sequência de comandos e verifique o resultado. Note que o comando `touch` do Linux é aqui usado para criar o ficheiro vazio `ReadMe.txt`.

```
> touch ReadMe.txt
> ls
ReadMe.txt
> git add ReadMe.txt
> git commit -m "o Primeiro commit, adiciona o ReadMe.txt"
[master (root-commit) c233151] o Primeiro commit, adiciona o
ReadMe.txt
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 ReadMe.txt
```

Surge então a terceira questão: “O que acontece aos ficheiros entre a execução do comando `add` e a execução do comando `commit`?”

Na verdade, o Git funciona através do conceito de índice (*index*) ou palco (*stage*). Um *stage* ou *index* é um coletor de `add`'s (ou seja de registos da intenção de alteração) dos ficheiros a serem publicados num **próximo commit**. Após o `commit` o índice é limpo, e no repositório é criado um novo objecto com a informação dos ficheiros alterados e que estavam no índice, com informação sobre quem a executou, e uma mensagem descritiva desta publicação.

De forma resumida, ao usar o Git recorremos:

1. ao directório de trabalho (working directory),
2. à área de preparação (staging area) e
3. ao repositório do Git (repository).

A passagem de 1. para 2. faz-se com o comando `add` e a passagem de 2. para 3. faz-se com o comando `commit` (que requer uma mensagem descritiva das alterações feitas).

A quarta questão, “Como é que podemos saber o estado de um repositório?”, pode ser respondida usando o comando `status`.

```
> git status
```

que informa o utilizador sobre os novos ficheiros, os ficheiros que foram alterados e os que foram removidos.

Antes de publicar as alterações (i.e., de fazer o `commit`) é possível retirar do *stage* ficheiros que lá se encontram e que afinal não se pretende guardar no repositório. Tal é realizado com o comando:

```
> git reset HEAD nome_do_ficheiro
```

Neste contexto, o conceito de HEAD é a marca do Git que identifica a versão mais recente do ficheiro que está guardada no repositório. Experimente executar o conjunto de comandos a seguir:

```
> touch ReadMeNew.txt
> touch ReadMeYetAnother.txt
> git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

ReadMeNew.txt
ReadMeYetAnother.txt

nothing added to commit but untracked files present (use "git add" to track)
> git add ReadMeNew.txt ReadMeYetAnother.txt
> git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

new file:   ReadMeNew.txt
new file:   ReadMeYetAnother.txt

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

new file:   ReadMeNew.txt

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

ReadMeYetAnother.txt
```

Uma das grandes vantagens do uso de Git é a possibilidade de trabalhar em paralelo com várias versões alternativas do mesmo trabalho. Essas versões estão armazenadas numa estrutura de árvore. As alterações sucessivas feitas aos ficheiros de cada uma dessas alternativas estão num ramo (*branch*) desta árvore. Existem comandos para criar e juntar ramos. Nesta fase temos apenas um ramo, o ramo principal (**master**). Daí a mensagem: "On branch master" que aparece na resposta do sistema aos comandos anteriores.

Um branch é um *snapshot* de um projecto; pode ser necessário ter vários *snapshots* de um projecto, cada um deles deve ser um *branch* distinto. Por exemplo, pode ser útil no caso de se pretender desenvolver em paralelo uma possível alternativa ao código em desenvolvimento.

Existe ainda um outro conjunto de informações importantes que o comando `status` indica: quais são os ficheiros que estão no stage (isto é indexados para serem publicados) e os que estão no working directory e que não são alvo de indexação para publicação. Sugere ainda um conjunto de comandos alternativos para o que fazer a seguir.

Note que a versão local fica inalterada, ou seja, as alterações feitas não são perdidas. Contudo na mesma situação, pode reverter-se as alterações feitas num ficheiro para a última versão guardada no repositório através do comando `checkout`:

```
> git checkout nome_do_ficheiro
```

sendo também possível com o mesmo comando copiar uma versão anterior desde que seja explicitada

qual.

```
> echo "Esta é uma frase do ReadMe" > ReadMe.txt
> git add ReadMe.txt
> git commit -m "ReadMe alterado"
[master d67d191] ReadMe alterado
 1 file changed, 1 insertion(+)
> cat ReadMe.txt
Esta é uma frase no ReadMe
> echo "Esta é outra" >> ReadMe.txt
> cat ReadMe.txt
Esta é uma frase no ReadMe
Esta é outra
> git add ReadMe.txt
> git reset HEAD ReadMe.txt
Unstaged changes after reset:
M ReadMe.txt
> cat ReadMe.txt
Esta é uma frase no ReadMe
Esta é outra
> git checkout ReadMe.txt
> cat ReadMe.txt
Esta é uma frase no ReadMe
```

A quinta questão é “O que significa a informação ‘On branch master’?” Indica que estamos no ramo principal ou mestre (master) do nosso repositório. O Git permite a funcionalidade de criação de ramos diferenciados de desenvolvimento a partir de um ponto (isto é de um *commit*) de forma a que possam ser aplicadas alterações a ambos os ramos de forma independente e que mais tarde possam ser juntas (ou removidas). É a forma de em equipa, por exemplo, se desenvolver código em simultâneo e de forma independente, sem afetar o trabalho uns dos outros. A criação de um ramo é feita através do comando **branch** seguido do nome do ramo:

```
> git branch nome_do_ramo
```

e para mudar de ramo onde vai fazer as suas alterações usa-se o comando *checkout* seguido do nome do ramo:

```
> git checkout nome_do_ramo
```

Note que todas as alterações que fizer no novo ramo não aparecerão no ramo principal, ou seja, no ramo *master*. Para refletir as alterações do ramo criado no ramo *master* é preciso fazer uma junção através do comando **merge** no ramo *master*:

```
> git checkout master
> git merge nome_do_ramo
```

Com o primeiro comando, fica com o HEAD a apontar para o ramo *master* e, com o segundo comando, todas as alterações feitas no novo ramo serão adicionadas ao ramo *master*. Experimente agora executar os comandos seguintes na sua máquina e observe os resultados.

```
> git branch NovoRamo
> git branch -a
  NovoRamo
* master
> git checkout NovoRamo
Switched to branch 'NovoRamo'
> echo "Este é um ficheiro no ramo NovoRamo" > ReadMeNovoRamo
```

```
> cat ReadMeNovoRamo
Este é um ficheiro no ramo NovoRamo
> git add ReadMeNovoRamo
> git commit -m "ReadMeNovoRamo adicionado ao NovoRamo"
[NovoRamo 0d96867] ReadMeNovoRamo adicionado ao NovoRamo
1 file changed, 1 insertion(+)
create mode 100644 ReadMeNovoRamo
> git status
On branch NovoRamo
nothing to commit, working tree clean
> git checkout master
Switched to branch 'master'
> git merge NovoRamo
Updating d67d191..0d96867
Fast-forward
 ReadMeNovoRamo | 1 +
1 file changed, 1 insertion(+)
create mode 100644 ReadMeNovoRamo
```

Note que o ramo ainda existe. Para remover esse ramo usa-se o comando `branch` com a opção `-d`:

```
> git branch -d nome_do_ramo
```

A remoção de um ramo não remove nenhum ficheiro, apenas remove a referência ao *commit* correspondente.

Mais uma vez, experimente o seguinte conjunto de comandos na sua máquina:

```
> git branch
  NovoRamo
* master
> git branch -d NovoRamo
Deleted branch NovoRamo (was 0d96867).
> git branch
* master
```

Existe ainda um comando muito útil que permite ver o que foi sendo alterado no repositório, o comando `log`:

```
> git log
```

```
commit 0d9686785d695b869b383421ac0aaec8ea4403d4
Author: Andre Souto <ansouto@fc.ul.pt>
Date:   Fri Feb 10 10:36:21 2017 +0000

    ReadMeNovoRamo adicionado ao NovoRamo

commit d67d19108582b8745d47826d26a48f0232f11181
Author: Andre Souto <ansouto@fc.ul.pt>
Date:   Fri Feb 10 09:15:50 2017 +0000

    ReadMe alterado

commit aeaea1b0784cd33c1dbc4975703dddf2ac15a7c
Author: Andre Souto <ansouto@fc.ul.pt>
Date:   Fri Feb 10 09:02:04 2017 +0000

    new files ReadMeNew and ReadMeYetAnother
```

```
commit c233151c31a78984b094ee569a313303ed54e17c
Author: Andre Souto <ansouto@fc.ul.pt>
Date:   Fri Feb 10 08:46:21 2017 +0000
```

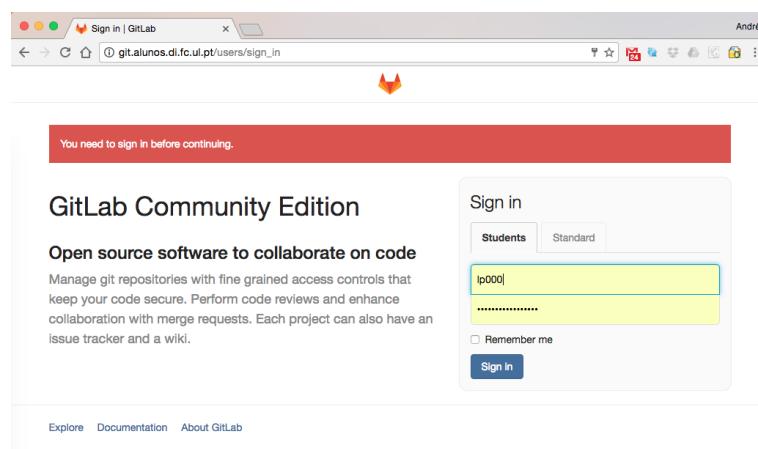
- o Primeiro commit, adiciona o ReadMe.txt

2.4.5 A interação com um repositório remoto

Até agora, temos trabalhado apenas num repositório local. O Git não tem um modelo cliente-servidor mas sim *Peer-to-peer* e cada membro da equipa tem o seu próprio repositório independente. A partilha da informação em Git é feita via protocolos de “ssh” ou “https”. De modo a facilitar o modelo de comunicação e gestão recorre-se a um repositório remoto para partilhar o trabalho entre os vários membros.

Desta forma e para que não haja necessidade de todos comunicarem com todos, cada um trabalha no seu próprio repositório local e, quando termina, “empurra” o código para o repositório remoto. Por outro lado, os outros elementos podem “puxar” o código deste repositório e obter as alterações mais recentes.

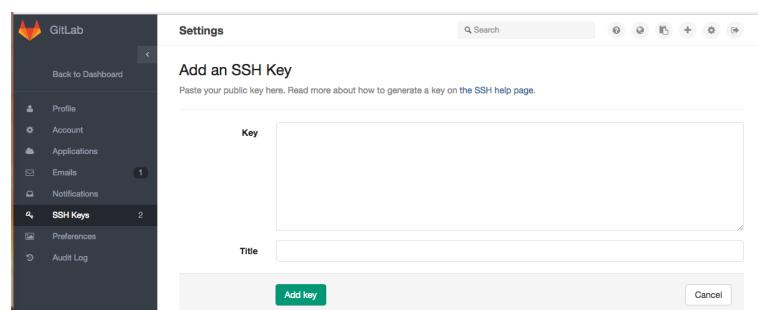
No caso dos alunos do DI, o repositório remoto que iremos usar é o servidor de GitLab instalado nos servidores do departamento (<https://git.alunos.di.fc.ul.pt/>) e que os alunos podem usar com o *login* e *password* de aluno desde que dentro da facultade fisicamente ou com uma ligação VPN.



Antes de podermos criar projectos neste repositório é necessário configurar as chaves SSH que identificam os pontos de acesso ao GitLab a partir de uma máquina (esta máquina pode ser própria e/ou dos laboratórios do DI) para que possam aceder ao conteúdo do repositório remotamente.

A alternativa à configuração da chave SSH é o use do endereço [https](https://git.alunos.di.fc.ul.pt/) para aceder ao repositório. O defeito é que terá que entrar as suas credenciais em cada interação com o repositório remoto.

A configuração é feita em **Profile Settings > SSH keys > Add key** que aparece do lado direito. A página que aparece é a seguinte:



O campo “Key” tem de ser preenchido com a sua chave RSA pública. Provavelmente ainda não criou uma chave RSA para o seu computador. Aqui estão os passos necessários e efetuar num terminal:

```
> ssh-keygen
```

```
Generating public/private rsa key pair.
Enter file in which to save the key (/home/tl/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/tl/.ssh/id_rsa
Your public key has been saved in /home/tl/.ssh/id_rsa.pub
The key fingerprint is:
SHA256:EU/FYflr5KIdL2BAK66nB1rgxheQeO3+i+OuiSfi5ss tl@voyager1
The key's randomart image is:
+---[RSA 3072]---+
| . o   . .o+o |
| . + . .+ .o  |
| . o   .... . |
| . o . o.    o |
| o o o .S.   o .|
| + = .   o o + |
| . + + . + = |
|+oo.o.+   . o .|
|*E++=..   . . |
+---[SHA256]---+
cat /.ssh/id_rsa.pub
```

```
ssh-rsa AAAAB3NzaC1yc2EAAAQABAAQCrhdnXXctW5qMcp+iRmlszblNvZdoGvQDNtQ
8bYkmoJXXAK97Tzd/hGCK+93IYXnsjLZ2LIWifwxo2EEP/EHHRQ4daICZNQHdhyBLVxwgscYc1
T944zDYFz6N/Dk0YhxTSfHwGCorMMYs6s5O3ZIhzPWAYKcnkrUxt4Uiww4N1wQXkJTzo8tmAAB
FL2zwV8ZMdM+CXf9SXrmP6tVV0RBpTU1T472NgXbH66YApKkp8dvnCzpqNurI5Gg8nTM3VHQAZ
tfZoePX9s1wM0of3gRDB3ZCjJTERZH+Urb0UdVCXYArZeCOXQf305QDbPGPhEmnc9QKMeJuuI
XiiIehXRnh tl@voyager1
```

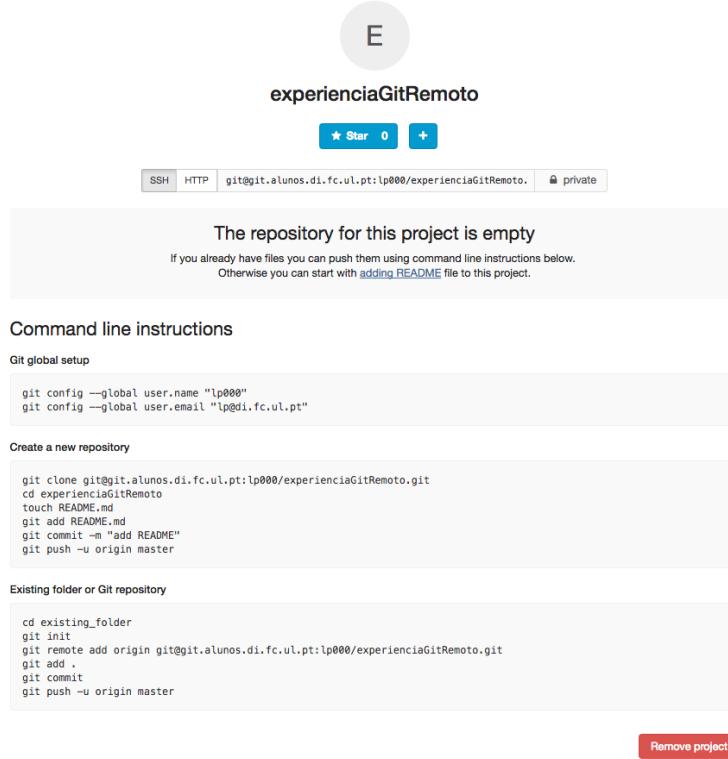
Para a geração da chave (ssh-keygen), deve carregar três vezes na tecla Enter de modo a escolher as opções por defeito. A sua chave pública encontra-se na pasta `.ssh`, na raiz da sua área. O comando cat mostra-a. Basta copiar a chave para a página de configuração do Gitlab.

Configuração do SSH na rede do DI

Na rede do DI, a configuração do SSH requer um passo adicional. Deve criar um ficheiro com o nome `config` a colocar na pasta `.ssh` referida anteriormente. Deve copiar no ficheiro as linhas seguintes:

```
host git.alunos.di.fc.ul.pt
User fc53224
IdentityFile ~/.ssh/id_rsa.pub
KexAlgorithms +diffie-hellman-group1-sha1
PubkeyAcceptedKeyTypes ssh-rsa,ssh-dss
HostKeyAlgorithms ssh-rsa,ssh-dss
onde terá colocado o seu número de aluno onde está 53224.
```

Crie um projecto Git diretamente no servidor (o Gitlab chama um repositório “projeto”):



e seguindo a sugestão dada crie um ficheiro `README.md` diretamente no browser para poder clonar imediatamente o repositório na sua máquina.

Clonar (comando `git clone`) é a operação de Git que permite criar uma cópia do repositório Git, neste caso na sua máquina local:

```
> git clone <url>
```

Vamos experimentar clonar o repositório que criámos no início desta secção (o URL do repositório encontra-se clicando no botão “Clone”) para um repositório local. Para isso execute no seu terminal os comandos que se seguem:

```
git clone http://git.alunos.di.fc.ul.pt/alberto/experienciaGitRemoto.git
```

```
Cloning into 'experienciaGitRemoto'...
Username for 'http://git.alunos.di.fc.ul.pt': alberto
Password for 'http://alberto@git.alunos.di.fc.ul.pt': ****
remote: Counting objects: 3, done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
Checking connectivity... done.
```

```
> ls
experienciaGitRemoto
ls -al
```

```
total 8
drwxr-xr-x  4 alberto  staff   136 Feb 10 13:39 .
drwxr-xr-x  6 alberto  staff   204 Feb 10 13:38 ..
drwxr-xr-x 13 alberto  staff   442 Feb 10 13:39 .git
-rw-r--r--  1 alberto  staff    48 Feb 10 13:39 README.md
```

```
> git status
```

```
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

Com um servidor remoto e um clone local torna-se necessário distinguir os dois. O Git faz isso usando um ficheiro (designado `.git/config`) identificando a origem com o nome `origin`. Esta informação é fundamental para a sincronização entre o repositório local e o repositório guardado no servidor. Podemos então criar ficheiros novos no nosso repositório local. Observe os comandos e a informação que é devolvida:

```
> echo "Ola servidor" > testeSinconizacao.txt
> git add testeSinconizacao.txt
> git commit -m "Ficheiro criado localmente para colocar no servidor"
```

```
[master 9050317] Ficheiro criado localmente para colocar no servidor
1 file changed, 1 insertion(+)
create mode 100644 testeSinconizacao.txt
```

```
> git status
```

```
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)
nothing to commit, working directory clean
```

Note que o próprio Git informa que existe uma publicação adiantada em relação ao repositório no servidor (o `origin`) e de facto se for ver no browser os ficheiros que lá constam o `testeSinconizacao.txt` não está lá porque apenas se fez a publicação no repositório local. Para efetivar a publicação na cópia do repositório no servidor é necessário comunicar essa alteração ao servidor através do comando `git push`:

```
> git push origin master
```

```
Counting objects: 3, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 327 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To http://git.alunos.di.fc.ul.pt/alberto/experienciaGitRemoto.git
  867ac89..9050317 master -> master
```

```
> git status
```

```
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

É importante frisar que apenas os ficheiros que estão publicados localmente serão “empurrados” para o repositório no servidor. Mesmo que estejam no *stage* (isto é preparados para publicação) mas não publicados não serão empurrados. Por isso é importante que confirme todas as alterações antes de colocar as alterações no repositório remoto.

Para “puxar” as últimas atualizações do repositório guardado no servidor deve usar o comando `git pull` da origem:

```
> git pull origin
```

Already up-to-date.

Se voltar ao browser e acrescentar um ficheiro direto no repositório, a cópia que tem localmente já não estará actualizada e se repetir novamente o comando tem:

```
> git pull origin

remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From http://git.alunos.di.fc.ul.pt/alberto/experienciaGitRemoto
  9050317..2380a4b master      -> origin/master
Updating 9050317..2380a4b
Fast-forward
 NovoFicheiroCriadoNoServidor | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 NovoFicheiroCriadoNoServidor
```

Com o uso de um repositório no servidor é muito fácil acontecerem conflitos entre ficheiros. Por exemplo, alterou um ficheiro, publicou a alteração no repositório do servidor e depois num outro repositório local alterou o mesmo ficheiro sem primeiro verificar a existência de alterações (isto é, esqueceu-se de fazer um `git pull`). Ao tentar publicar as novas alterações no servidor, o Git irá interromper o `git push` porque há um conflito de versões e que terão de ser resolvidas antes de se continuar.

A título ilustrativo, suponha então que criou um ficheiro `conflito` diretamente no servidor com o texto “Para testar um conflito de versões de um ficheiro”. Faça o `git pull` para transferir o ficheiro para o repositório local e volte ao servidor e altere o ficheiro acrescentando o texto “Este foi alterado no servidor diretamente depois do pull para o repositório local” e na cópia local altere o mesmo ficheiro (sem fazer o `git pull`) acrescentando o texto “Será que o conflito aparece?” e agora tente fazer a publicação das alterações para o repositório no servidor.

```
> nano conflito
> git add conflito
> git commit -m "alterado localmente sem pull"
```

```
To http://git.alunos.di.fc.ul.pt/alberto/experienciaGitRemoto.git
 ! [rejected]      master -> master (fetch first)
error: failed to push some refs to 'http://git.alunos.di.fc.ul.pt/alberto/experienciaGitRemoto'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

Este conflito tem de ser resolvido! Uma alternativa é fazer um `git pull` que revela automaticamente o conflito entre os ficheiros:

```
> git pull origin

remote: Counting objects: 3, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From http://git.alunos.di.fc.ul.pt/alberto/experienciaGitRemoto
  1c48d55..bc02afd master      -> origin/master
Auto-merging conflito
CONFLICT (content): Merge conflict in conflito
Automatic merge failed; fix conflicts and then commit the result.
```

Para resolver o conflito manualmente observe o conteúdo do ficheiro conflito:

Para testar um conflito de versões de um ficheiro.

```
<<<<< HEAD
```

Será que o conflito aparece?

```
=====
```

Este foi alterado no servidor diretamente depois do pull para o repositório local.

```
>>>>> 012097daa424f54e89e40f899697f3152dbd1463
```

Note que o ficheiro conflito é alterado para evidenciar a diferenças entre a versão que se encontra localmente (assinalada com <<<<< HEAD) e a versão que existe no repositório do servidor (assinalada com ===== e com >>>>> 012097daa424f54e89e40f899697f3152dbd1463 referente à publicação que originou o conflito).

Altere o ficheiro para o que fizer sentido, no nosso caso para:

Para testar um conflito de versões de um ficheiro.

Este foi alterado no servidor diretamente depois do pull para o repositório local.

Será que o conflito aparece?

e volte a fazer `git add` e a publicação no repositório local com o `git commit` das alterações para depois as publicar com o `git push` novamente para o repositório no servidor. Siga os comandos seguintes:

```
> git add conflito
> git commit -m "resolucao de conflito num ficheiro"
[master 0ad5ac5] resolucao de conflito num ficheiro
> git push origin master
```

```
Counting objects: 6, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 622 bytes | 0 bytes/s, done.
Total 6 (delta 3), reused 0 (delta 0)
To http://git.alunos.di.fc.ul.pt/alberto/experienciaGitRemoto.git
  012097d..0ad5ac5  master -> master
```

Existem dois outros comandos que permitem um controle mais fino que o `git pull`. O comando `git fetch` que permite obter uma atualização do repositório sem que sejam alteradas as cópias dos ficheiros locais e o comando `git merge` que atualiza os ficheiros do repositório.

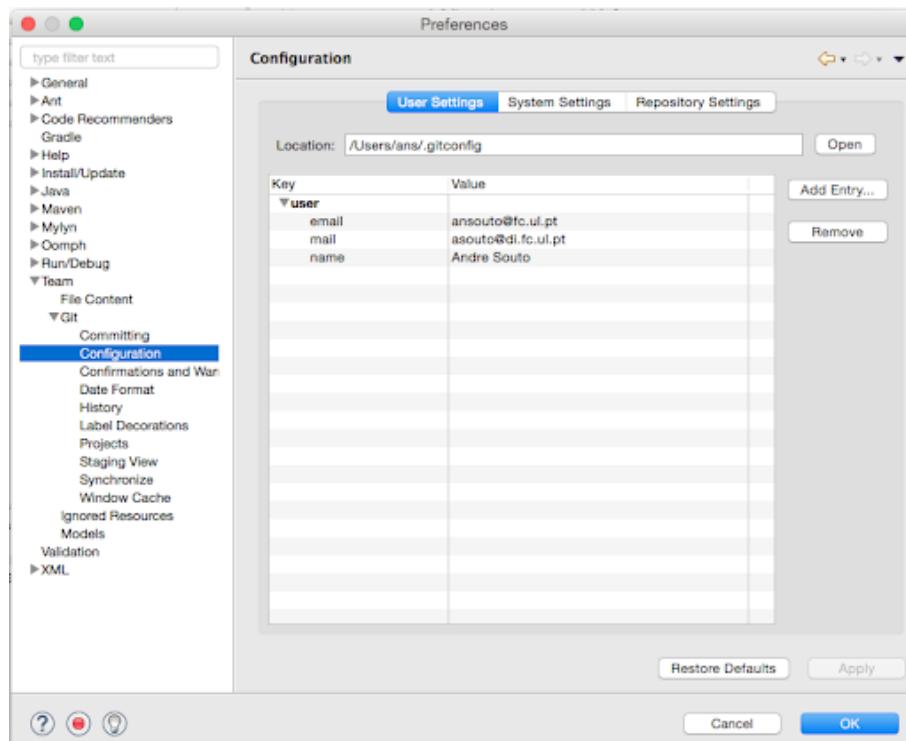
2.4.6 Integração com o Eclipse

Esta secção foca-se no desenvolvimento de código em ambiente Eclipse, descrevendo a forma como podemos integrar o uso de git neste IDE.

Por norma, a instalação padrão de Eclipse traz incorporado o EGit, um software de integração de Git com o Eclipse. Por essa razão irá permitir a interação direta entre o workspace do Eclipse e o repositório remoto onde irá guardar os seus projectos para trabalhar em locais diferentes.

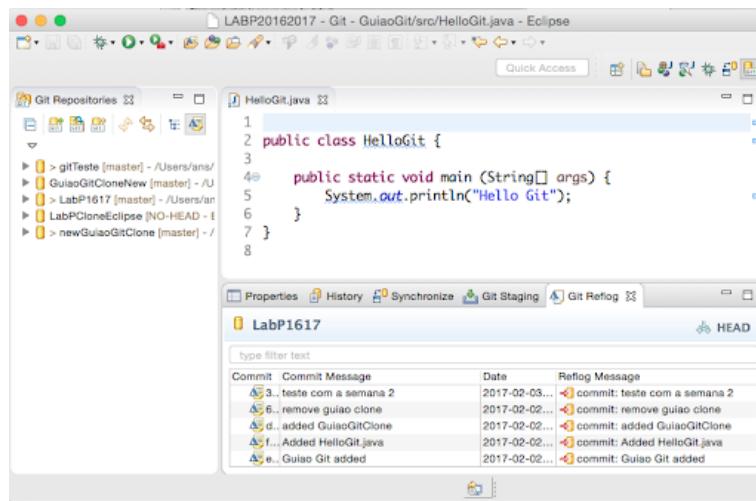
Antes de começar a trabalhar conjuntamente com o git e o Eclipse é necessário fazer as configurações adequadas para que tudo funcione corretamente.

Primeiro, é necessário configurar o nome e o email a usar no Git, informação que será utilizada para informar sobre os commits feitos. Se não fez esta configuração via linha de comandos ou se precisa confirmar alterações, pode no Eclipse ir ao menu `Window > Preferences > Team > Git > Configurations` > para ver a informação supra mencionada e corrigi-la caso seja preciso. Note que o Git pode ser facilmente configurado com o comando `git config` (ver p. 24).



Perspectiva Git

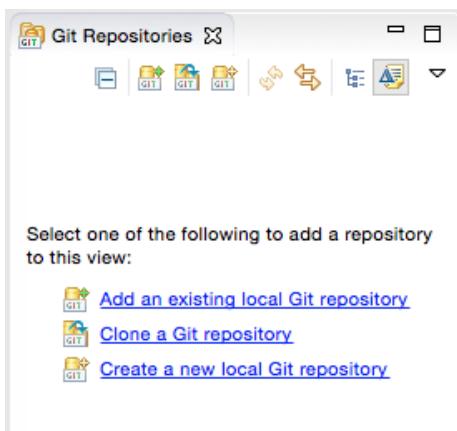
Para melhor visualizar as alterações referentes aos repositórios pode abrir a perspectiva Git através de **Window > Open Perspective > Other > Git >**.



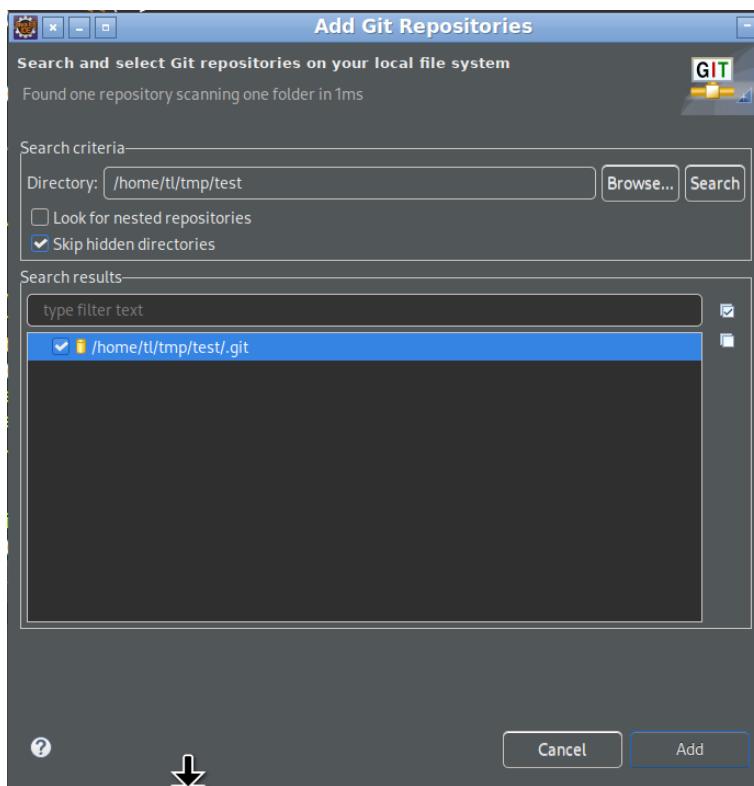
A esquerda na aba Git repositories opde ver a lista de repositórios conhecidos do Eclipse. Se não ouver nenhum vai aparecer instruções para criar ou clonar um repositório. Na aba principal aparece geralmente o conteúdo de um ficheiro. Ainda a direita mas na parte de baixo vai há várias abas dedicadas ao Git: History, Synchronize, Git Staging e Git Reflog.

Criar um projecto Java para colocar no Repositório Remoto

Dado que já temos o repositório local ligado ao repositório remoto, vamos começar por colocá-lo acessível no Eclipse através de **Add an existing local repository** acessível na perspectiva Git como mostra a figura.

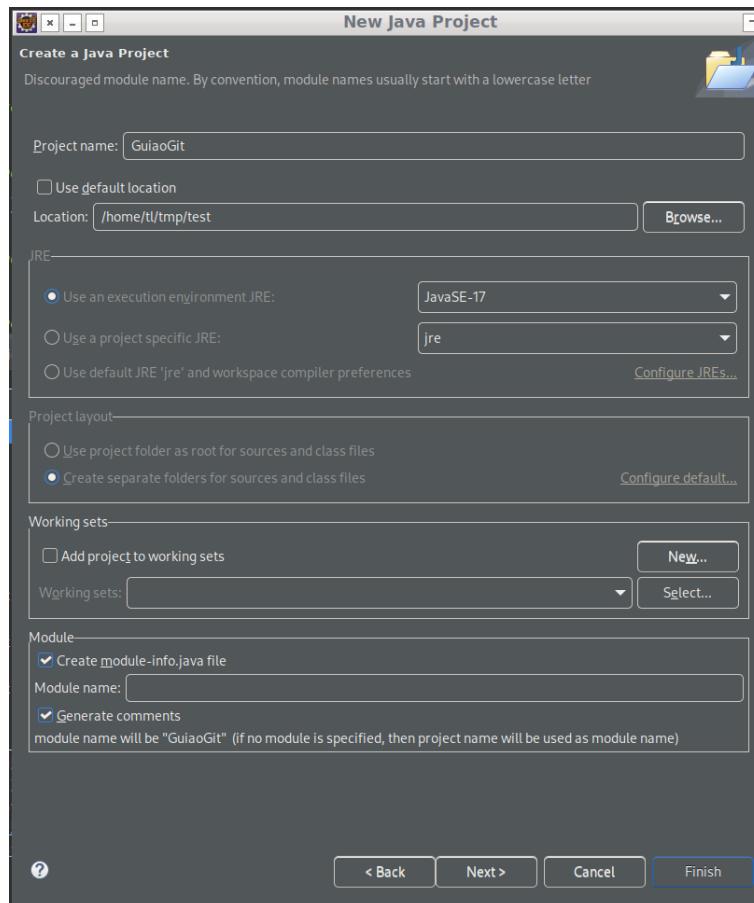


Caso não tenha a vista o link, a alternativa é clicar com o botão direto na aba “Git Repositories” e escolher a opção “Add Git Repository”. Agora é só localizar o nosso repositório:

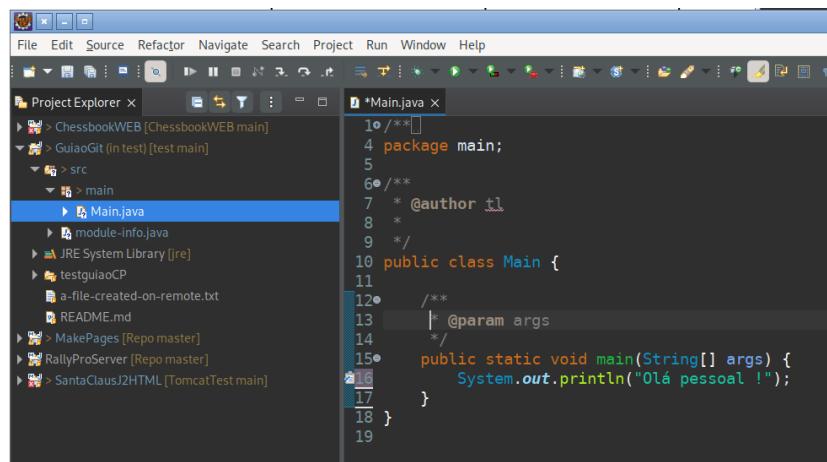


Na aba de **Git Repositories**, vemos então o repositório associado. Falta colocar o projecto Java nesta pasta para que faça parte do repositório.

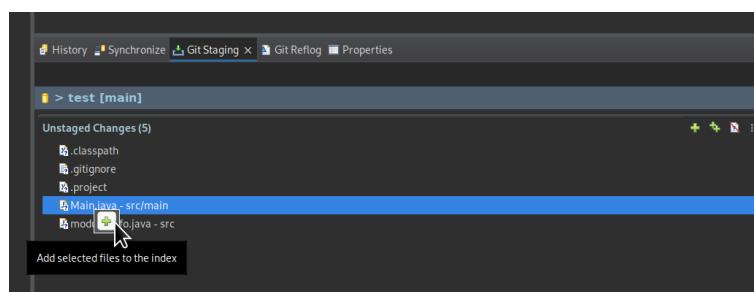
Crie um projecto Java com o nome **GuiaoGit**. É importante escolher como pasta para o projeto a pasta que corresponde ao repositório caso contrário o seu projeto não fará parte do repositório !. Tem de NÃO selecionar “Use default location” e indicar a pasta correta:



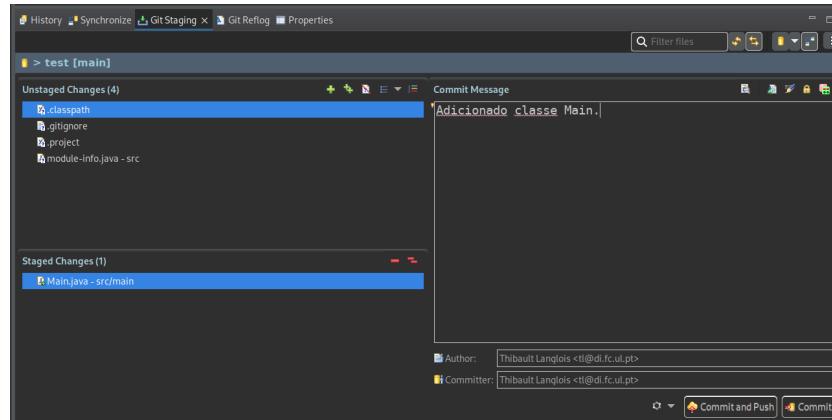
A seguir volte à perspetiva Java e acrecente ao projeto um pacote `main` e uma classe `Main`:



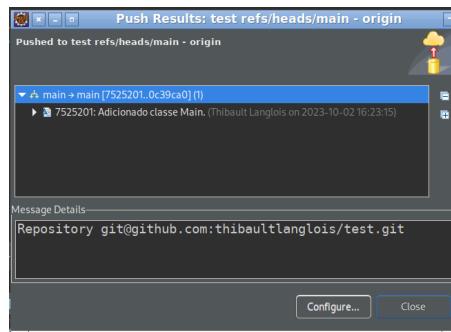
Pode agora voltar a perspetiva Git. Na aba Git Staging pode ver os ficheiros criados ou modificados (alguns foram criados pelo Eclipse). Quando clica num dos ficheiros aparece um botão "+" que permite adicionar o ficheiro ao índice.



Os ficheiros presentes no índice aparecem claramente por baixo, em “Staged Changes”. Uma vez que selecionou todos os ficheiros que quer incluir no próximo commit, pode preencher a mensagem que será associada ao commit e clicar em commit+ push :

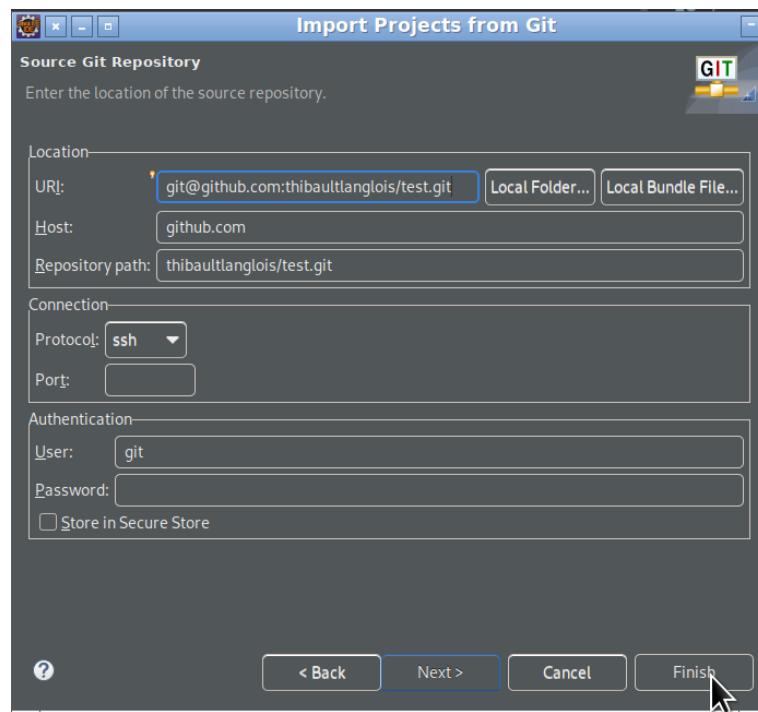


Passado algum tempo... o eclipse mostra uma janela onde confirme que a operação foi concluída com sucesso:

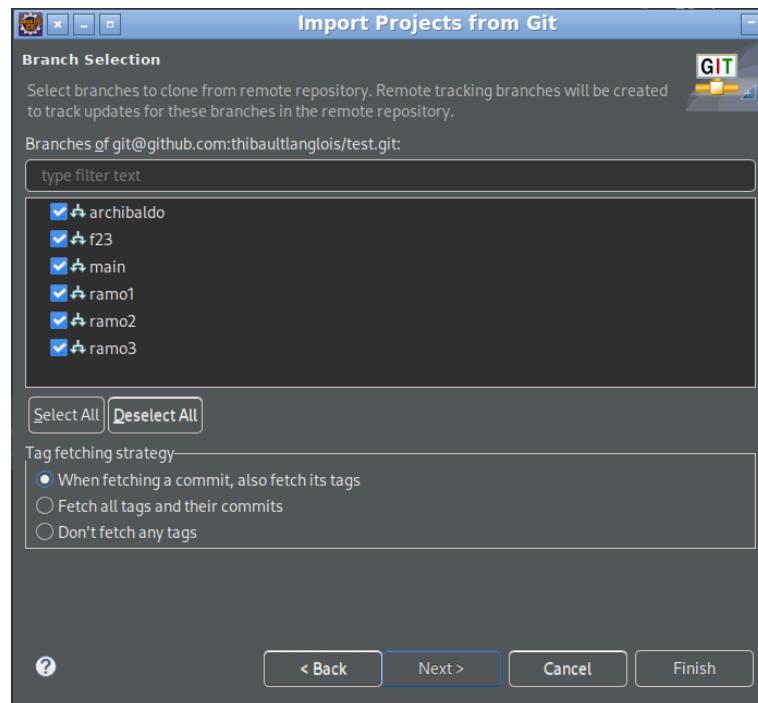


Importar ou clonar um projeto já existente

Na secção anterior importamos um repositório local existente. A situação mais comum é existir um repositório **remoto** e querermos clona-lo para trabalhar nele. Vamos ver como nesta secção. Para clonar um repositório já existente (remoto) podemos recorrer ao menu **File > import > Git > Projects from Git >** e escolher a opção **clone URI**. Basta então preencher o campo **URI**:

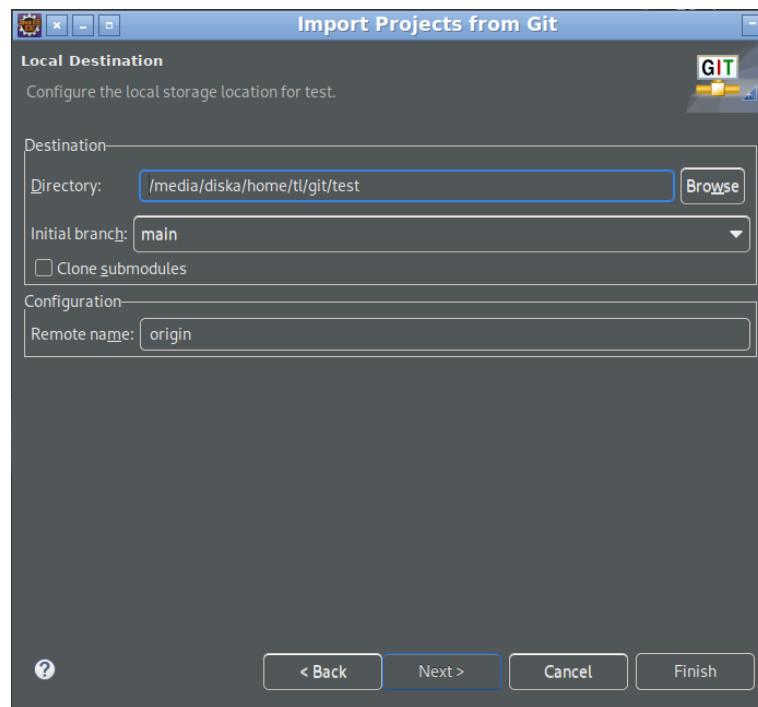


O URI encontra-se na página do projeto, basta clicar no botão **Clone**. Clicar em **Next**. A seguir deve escolher os ramos que quer importar (Cuidado, este passo vai funcionar apenas se a chave SSH estiver bem configurada !):

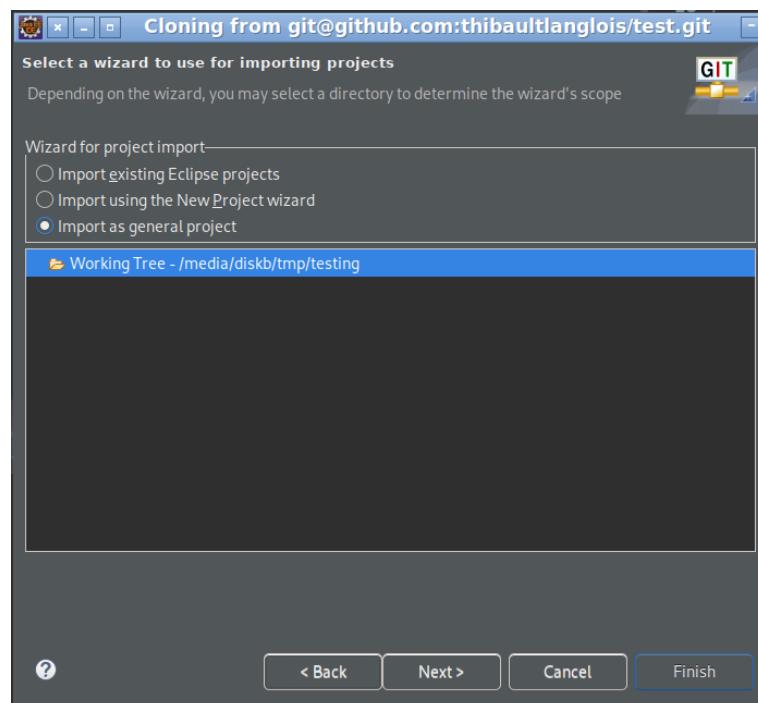


Clicar em **Next**.

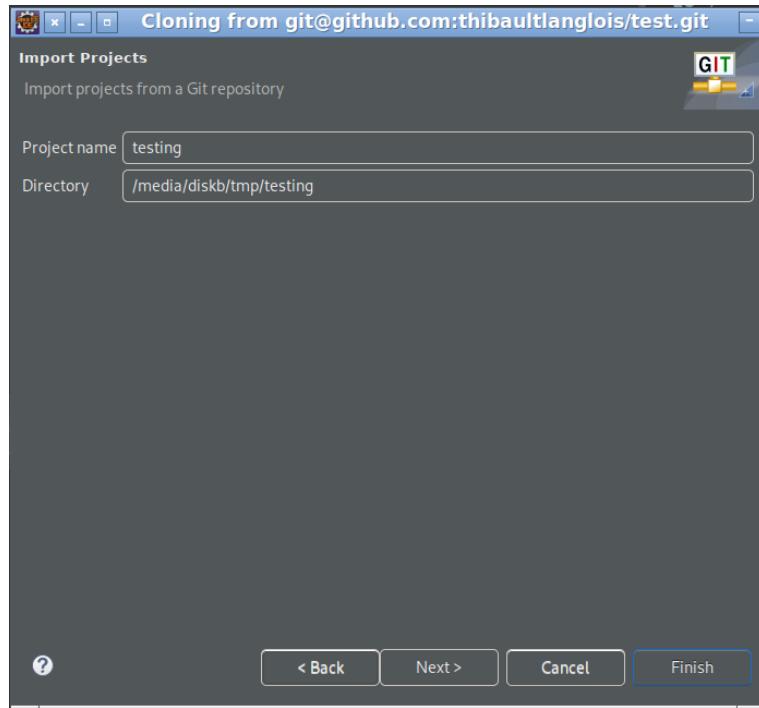
A seguir deve escolher a pasta para onde o repositório será copiado. Deve escolher um pasta nova e vazia.



O Eclipse insiste em trabalhar com “projetos”, vai criar um ficheiro .project no repositório:



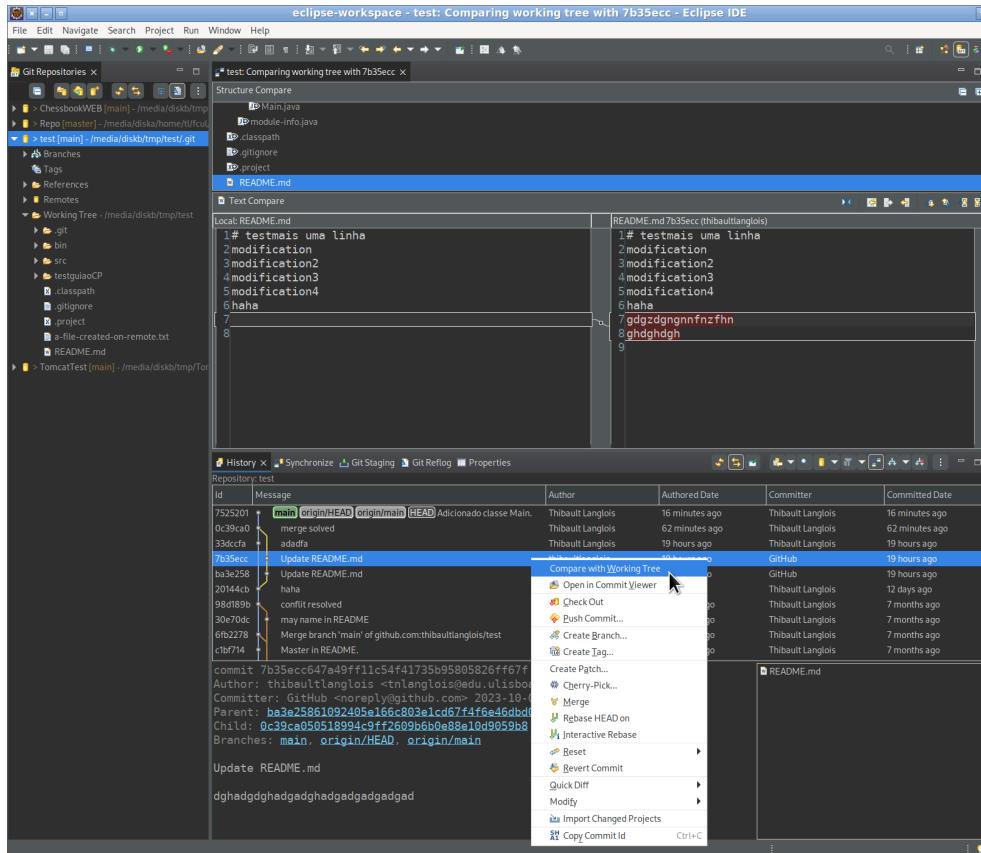
Clicar em **Next**.



Finalmente pode carregar em **Finish**.

Uso do Histórico

Na perspetiva Git existe uma aba “History” onde pode ver o histórico do repositório. Uma imagem vale mil palavras:

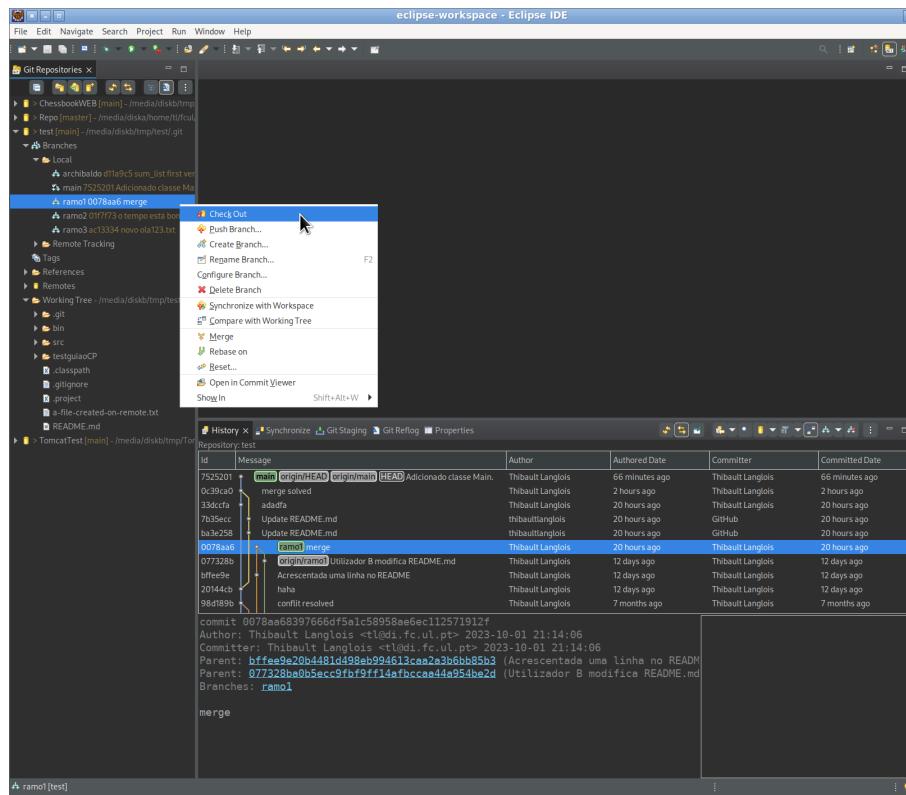


Em baixo pode ver a sequência de *commits*, os ramos e as informações relativas a cada *commit*. No seu caso não haverá tantos *commits* e portanto menos história. Como mostrado na imagem, pode

clicar num *commit* (com o botão direito) para fazer aparecer um menu. É preciso insistir sobre o facto que, caso execute um comando “ao acaso” para “experimentar” poderá colocar o seu repositório numa situação complicada e eventualmente perder trabalho. Dito isso, não vamos explicitar cada opção mas apenas a primeira: “*Compare with Working tree*”. Esta opção permite visualizar diferenças entre o *commit* selecionado e o último (*HEAD*). Aqui está selecionado o ficheiro *README.md* (parte de cima) e aparece as diferenças (no meio) entre o conteúdo atual (a esquerda) e a versão do *commit* selecionado (7b35ecc, a direita).

Trocá de ramo

Pode-se trocar de ramo via a interface gráfica:



Ao clicar no ramo, o último *commit* deste ramo está *highlighted* no histórico. Pode ver que o repositório local está adiantado de um commit em relação ao remoto.

2.5 Guião Git 2

Salvo indicado em contrário, neste guião todas as interações com os repositórios git são feitas com instruções na linha de comando.

Para obter uma representação compacta da saída do comando `git log` acrescente a seguinte linha ao seu ficheiro `.bashrc` que está na raíz da sua área:

```
alias gitlog='git log --decorate --color=always --pretty=tformat:"%C(auto)%h %ce %s %d" --graph'
```

(tudo na mesma linha). Uma vez feita esta alteração pode usar o comando `gitlog` em vez de `git log`. Quando uma opção não está completamente especificada na linha de comando, o git corre um editor para permitir entrar a informação em falta. Pode configurar o git para usar o seu editor preferido. Por exemplo para passar a usar o nano pode configurar o git com o comando seguinte:

```
git config --global core.editor nano
```

2.5.1 Criar o repositório (Aluno A)

- Criar um repositório (projeto) “guiaogit2” na sua conta em `git.alunos.di.fc.ul.pt`. Atenção ! não deve escolher a opção que inicializa o repositório com um ficheiro `README.md` porque precisamos de um repositório remoto vazio para poder empurrar o repositório local.
- No seu computador (ou no lab):
 1. Criar uma pasta nova que não esteja dentro de um repositório git.
 2. Nesta pasta correr o comando : `git init`
 3. `git remote add origin o-endereço-no-gitlab/guiaogit2.git` este comando tem como efeito de associar o endereço do repositório remoto no seu repositório local. Uma vez que a inicializou a sua área no gitlab com a sua chave ssh, pode (deve) usar o endereço ssh do repositório.
 4. `echo "# guiaogit2">>> README.md`
 5. fazer commit das alterações.
 6. empurar o commit para o repositório remoto. Atenção ! tem de especificar para onde vai “empurar” e qual ramo local vai para o remoto (nesta ordem).
 7. verificar no browser que resultou.
 8. Adicionar o seu colega aos “membros” do projeto. Em “Role e permissions” escolher “Developer”.

2.5.2 Adicionar projeto java (Aluno B)

- No seu computador, criar uma pasta nova para o repositório.
- clonar o repositório criado pelo seu colega (não se esquece de usar o endereço ssh do repositório).
- Abrir o eclipse e criar um projeto java “guiaogit2” **de tipo Maven**. Para criar um projeto deste tipo deve escolher `New > Other > Maven > Maven Project >`. Clicar em “Next”, selecionar a opção “Create simple project...”. Não selecionar “Use default Workspace location” e indique a sítio do projeto. **O projeto deve estar na pasta do repositório e não na área do workspace do eclipse**. Clicar em “Next”. O groupId é `fcul.css` e o ArtefactId é “guiaogit2”. Clicar em “Finish”.
- Em `src/main/java` criar um pacote `domain`.
- Criar uma classe `ChessPlayer` no pacote `domain` com atributos `nome (String)` e `id (int)`. Bem como um construtor.
- Fazer commit das alterações **mas não empurar para o remoto**.

2.5.3 Completar o ficheiro README.md (Aluno A)

- Acrescentar ao ficheiro `README.md` o texto seguinte:

Este guião tem como objectivo a prática dos comandos do git num contexto de uma equipa, usando um repositório remoto.

- Fazer commit das alterações.
- Empurar as alterações para o repositório remoto. Cuidado ! tem de indicar para qual repositório (`remote == origin`) e qual o ramo que se vai enviar (`master`).

2.5.4 Adicionar getters and setters (Aluno B)

- Empurrar o repositório local para o remoto.
- Vai obter um erro. Qual é o tipo de erro ? Porquê aconteceu ?
- Este erro poderia ter sido evitado ? como ?
- Como remediar a situação ?

- Uma vez resolvido o problema, observar o novo estado do repositório (gitlog). Não empurrar para o repositório.
- Adicionar getters e setters na classe `ChessPlayer`.
- Fazer commit. Empurrar para o repositório. Caso obtenha o erro ao fazer o push, consulte a secção p. 52.

2.5.5 Atualizar o README no repositório remoto (Aluno A)

- Alterar o conteúdo do ficheiro `README.md`
- Fazer commit das alterações.
- Empurrar para o remoto
- O que é que aconteceu ?
- Resolva a situação e faça push das alterações efetuadas para a resolução da situação.

2.5.6 Transformar o projeto em projeto Maven (Aluno B)

O aluno B continua o desenvolvimento da classe `ChessPlayer`:

- Adicionar um método `toString` à classe `ChessPlayer`.
- Está na altura de fazer um commit e empurrar para o repositório remoto. O que é que deve fazer para evitar o problema que aconteceu nos passos anteriores ?
- Uma vez realizados esses passos, faça commit das alterações e empurre o seu repositório para o remoto.

2.5.7 Criar um ramo (Aluno A)

Ambos os alunos decidem trabalhar cada um no seu ramo fazendo o merge no master cada vez que uma etapa de desenvolvimento está completada. Para clarificar os ramos vão corresponder aos nomes dos alunos A e B. Os ramos “pessoais” não vão existir no repositório remoto. São apenas áreas de “rascunho” onde é feito o trabalho de cada um.

- Fazer um pull para ter uma cópia atualizada do repositório.
- Criar um ramo com o seu nome (neste guia vamos chamar este ramo “JOAO”).

2.5.8 Criar um ramo (Aluno B)

- Caso seja necessário fazer um pull.
- Criar um ramo com o seu nome (neste guia vamos chamar este ramo “MARIA”).

2.5.9 ChessPiece (Aluno A)

- Tornar o seu ramo (JOAO) o ramo corrente.
- Criar uma classe `ChessPiece` com dois atributos: `chessPieceKind` e `color` o primeiro atributo é de tipo `ChessPieceKind` que é uma classe enumerada (que contempla as constantes `KING`, `QUEEN`, `BISHOP`, `KNIGHT`, `ROOK` e `PAWN`). O segundo tipo do atributo é também uma classe enumerada (`BLACK`, `WHITE`).
- Adicionar um construtor na classe `ChessPiece`.
- Fazer commit dessas modificações para o ramo corrente.
- No “Package Explorer” observar os ficheiros java do projeto e a indicação do ramo corrente junto do nome do projeto.
- No terminal, passar para o ramo master.

- O eclipse não atualiza logo a interface. Está provavelmente a indicar o ramo antigo (JOAO). Para força a atualização da interface carregar na tecla F5.
- Agora a interface indica o ramo master, os ficheiros adicionados ao ramo JOAO desapareceram do eclipse. (Estão armazenados no repositório local).

2.5.10 Classe ChessPlayer (Aluno B)

O aluno B continua o desenvolvimento da classe `ChessPlayer`, desta vez no seu ramo. O Aluno B decide agora criar testes JUnit para verificar a classe `ChessPlayer`.

- Caso seja necessário, importar o projeto usando **File > Open Projects from File System > Show other specialized import wizards > Maven > Existing Maven Project > Browse >**; procurar a pasta do seu projeto (onde o ficheiro `pom.xml` está). Clicar em **Finish**
- Editar o ficheiro `pom.xml` e acrescentar as linhas seguintes a seguir ao bloco `<version>`:

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.13</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

copy

- Criar uma classe de teste: **New > JUnit Test Case >** se a opção não aparece pode ter que escolher **New > Other > Java > JUnit > JUnit Test Case >**. Para o nome da classe, escolher `TestChessPlayer`. Para a opção “Class under test” indicar `domain.ChessPlayer`. Clicar em “Next”. No ecrã seguinte selecionar os métodos `setName()` e `getName()`. Clicar em “Finish”.
- Completar os métodos de teste. Por exemplo o teste para o método `getName` será:

```
@Test
public void testGetName() {
    ChessPlayer pl = new ChessPlayer("Manuel");
    assertEquals(pl.getName(), "Manuel");
}
```

copy

- Correr os testes usando o eclipse. Clicar com o botão direito na classe de teste e escolher **Run as > JUnit Test >**. Os testes deve correr sem erros.
- No terminal use o comando `mvn clean` para eliminar os ficheiro criados na última compilação.
- Corre os testes com o comando:

```
mvn test
```

- Pode limpar, compilar e testar o projeto com :

```
mvn test
mvn install
```

- Uma vez que a classe passou os testes, pode adicionar o conteúdo do seu ramo de desenvolvimento ao ramo master:
 1. faça um commit no ramo corrente.
 2. torne o ramo master o ramo corrente.
 3. use o comando merge para fundir o ramo MARIA no ramo master.
 4. empurre o ramo master para o remoto.

2.5.11 Incluir testes para ChessPiece (Aluno A)

- O aluno A verifica que está no ramo master.
- Faz um pull de forma a syncronizar o ramo master do repositório local com o mesmo ramo no remoto.
- Passa para o ramo JOAO.
- Qual é o comando que deve usar para copiar o ficheiro pom.xml que está no master para o ramo JOAO ?
- Executar este comando.
- Fazer “refresh” ao projeto e forma a atualizar o eclipse. As dependencias Maven adicionadas pelo Aluno B devem aparecer.
- Na classe ChessPiece adicionar getters (mas não setters) e `toString`.
- Usando a interface do eclipse criar as pastas `src/test` e `src/test/java`. **Para isso deve escolher New > Folder >**, indicar no campo “parent folder” `guiaogit2/src/`. Indicar “test” para o “Folder name”. Clicar em “Finish”. Repetir a operação para criar a pasta `java` na pasta `test`.

Projetos Maven têm uma estrutura

A criação dessas pastas tem como objetivo manter a estrutura de um projeto Maven. Num projeto Maven o código fonte da aplicação fica na pasta `src/main/java` enquanto os testes ficam na pasta `src/test/java`. Um projeto de tipo Maven tem várias vantagens relativamente a um projeto java “classico”, criado pelo eclipse. Essas vantagens vão ser exploradas ao longo do semestre.

- Na pasta `src/test/java` criar o pacote `domain` (usando **New > Package >**).
- A semelhança do que o seu colega (Aluno B) fez, cria um “JUnit Test Case” com o nome `TestChessPiece`. Incluir testes para os métodos `getChessPieceKind` e `getColor`.
- Verifique que o projeto passa os testes (não deve ser difícil) usando os comandos :

```
mvn clean  
mvn test
```

- Está pronto para juntar essas modificações ao ramo master. Efetue isto usando um **merge** do ramo JOAO para o ramo master. O commando deve resultar sem erro dado que não alterou o seu ramo master, mas ... pode encontrar um conflito nesta altura (ver a razão e a solução ??).
- Empurrar o ramo master para o remoto.

Nesta altura o log do repositório do Aluno A, visto do ramo master deve ser algo semelhante a:

```
*   a343104 joao@di.fc.ul.pt Solved conflict in .classpath  (HEAD -> master, origin/master)  
|\  
| * 776f4f6 joao@di.fc.ul.pt Update in ChessPiece, tests for ChessPiece. all good.  (JOAO)  
| * 6872f0c joao@di.fc.ul.pt Classes ChessPiece ChessPieceKind PieceColor  
* | 351af77 maria@di.fc.ul.pt Tests for class ChessPlayer: good to go.  
|/  
* 134b57c maria@di.fc.ul.pt Added toString  
*   6600540 maria@di.fc.ul.pt Merge branch 'master' of github.com:thibaultlanglois/guiaogit4  
|\  
| * 440a85e maria@di.fc.ul.pt Added getters and setters  
| * 2e384ec maria@di.fc.ul.pt Merge branch 'master' of github.com:thibaultlanglois/guiaogit4  
| |\  
| * | 853a0cc maria@di.fc.ul.pt New maven project  
* | | 4206caa joao@di.fc.ul.pt Mais uma linha no README
```

```
| |/
|/|
* | eac627d joao@di.fc.ul.pt mais em README.
|/
* 00494f6 joao@di.fc.ul.pt First commit
```

O Workflow usado até agora foi:

- Cada membro usa um ramo para fazer o seu trabalho,
- Quando termina a implementação de uma funcionalidade faz o merge do seu ramo para o master,
- Empurra a ultima versão do master para o repositório remoto.

Os ramos pessoais não aparecem no remoto mas todos os commit feitos por cada membro fazem parte da história do repositório. O resultado pode ser um grafo um pouco confuso que pode tornar a análise do desenvolvimento do projeto complicada.

Vamos ver agora uma alternativa baseada no comando `git rebase`.

2.5.12 Class Players (Aluno B)

O Aluno B vai continuar o trabalho a volta da representação dos jogadores. É necessário haver uma classe que representa o conjunto de jogadores.

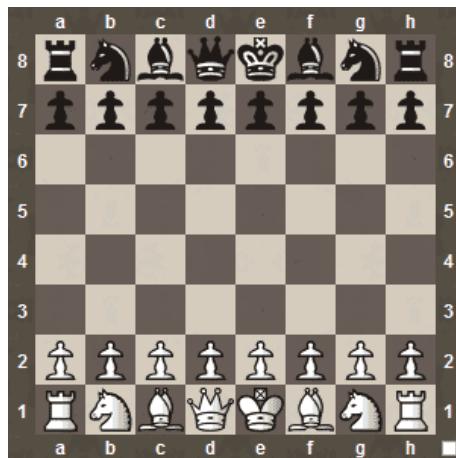
- Fazer um pull para ramo master de forma a obter a última versão do trabalho.
- Tornar o ramo MARIA o ramo corrente
- Foi decidido passar a usar o endereço de email para identificar de forma única os jogadores. Usar a opção **Refactor > Rename** para alterar o atributo “`id`” da classe `ChessPlayer` para “`email`”. Altere o tipo para `String`. Verifique o tipo de retorno do getter correspondente. Corrija o construtor para refletir o refactoring.
- Criar uma classe `PlayerCatalog` com os seguintes métodos:
 - `void addPlayer(String name, String email)` que adiciona um jogador à aplicação. Se o jogador já tiver sido introduzido previamente, o método lança a exceção `DuplicatedPlayerException`.
 - `Player getPlayer(String email)` que retorna o jogador com o email indicado ou `null` se não existir.
- Refletir sobre a estrutura de dados adequada para armazenar os jogadores.
- Pensou em definir o construtor ?
- Escrever testes (na pasta `src/test/java`, pacote `domain`) para verificar o funcionamento da classe.
- Correr os testes com os comandos :


```
mvn clean  
mvn test
```
- Fazer commit dessas alterações mas não empurrar nada para o remoto.

2.5.13 Classe ChessBoard (Aluno A)

Do seu lado o ALuno A continua o desenvolvimento para a representação do jogo. O proximo passo é a definição da classe `ChessBoard`.

- Verificar que tem todas as alterações feitas no remoto no seu ramo master (normalmente será o caso)
- Torne o ramo JOAO o seu ramo corrente.
- No pacote `domain`, defina uma classe `ChessBoard`. O tabuleiro será representado internamente por um array bidimensional de 8x8 posições. Tradicionalmente as posições num tabuleiro de xadrez são representadas por um par número-letra:



No código, vamos representar as posições por valores inteiros entre 0 e 7. A classe deve possuir os seguintes métodos:

- void set(int i, int j, ChessPiece cp) que coloca a peça indicada na casa indicada pelas coordenadas. Se a casa indicada for ocupada, o método deve lançar a exceção `IllegalMoveException`.
 - ChessPiece get(int i, int j) que retorna a peça presente na posição indicada. Caso a casa correspondente esteja vazia, retorna o valor `null`.
 - void remove(int i, int j) que remove a peça presente na posição indicada. Se a casa correspondente estiver vazia, o método não faz nada.
- Escrever testes (na pasta `src/test/java`, pacote `domain`) para verificar o funcionamento da classe.
 - Correr os testes com os comandos :
- ```
mvn clean
mvn test
```
- Fazer commit dessas alterações mas não empurrar nada para o remoto.

### 2.5.14 Javadoc ! (Aluno B)

O ramo corrente é ainda o ramo MARIA. Acrescente, nas classes desenvolvidas os comentários em formato Javadoc. Para gerar as páginas html vamos usar novamente o Maven. O primeiro passo consiste em adicionar um “plugin” ao ficheiro `pom.xml` (antes do último tag `</project>`) :

```
<plugins>
 <plugin>
 <groupId>org.apache.maven.plugins</groupId>
 <artifactId>maven-javadoc-plugin</artifactId>
 <executions>
 <execution>
 <id>attach-javadocs</id>
 <goals>
 <goal>jar</goal>
 </goals>
 </execution>
 </executions>
 </plugin>
</plugins>
```

#### copy

A seguir, para gerar as páginas html, basta usar os comandos:

```
mvn clean
mvn install
```

Examine a saída do último comando, o javadoc pode ter encontrado algumas imperfeções na sua documentação. Para visualizar a documentação produzida, pode carregar no seu browser o ficheiro target/apidocs.index.html.

Uma vez que está satisfeito com a documentação produzida, pode fazer o commit das suas modificações.

### 2.5.15 Javadoc ! (Aluno A)

Realize os passos indicados na secção anterior para criar a documentação para as classes desenvolvidas por si. Note que alteraram ambos o mesmo ficheiro pom.xml. Pode ser que no futuro tenha que resolver um conflito mas não será difícil dados que as mesmas alterações forma feitas em ambos os lados. Faça commit das suas modificações.

### 2.5.16 Rebase (Aluno B)

Na situação presente o Aluno B tem pelo menos dois commit no seu ramo que não estão no master. Antes de empurrar o trabalho feito para o remoto, o Aluno B quer juntar o trabalho feito no ramo MARIA ao master sem que seja registado o ramo onde foram feitos os últimos desenvolvimentos. O objetivo é não “sujar” o repositório remoto. Vai portanto usar a técnica do rebase:

- Corre os comandos :

```
git checkout master
git log
```

e aponta o id do último commit realizado neste ramo.

- Torne ramo MARIA o ramo corrente. Com o comando gitlog vizualize os commits deste ramo. Pode ver que o último commit do master não faz parte deste grafo.

- Corre os comandos

```
git rebase master
git log
```

Pode constatar que o último commit do master foi incluído na história deste ramo e que a sequência de commits é linear (não há bifurcações).

- O ultimo passo consiste em passar esses commits para o master:

```
git checkout master
git merge MARIA
```

Pode usar novamente o comando gitlog para observar que os commit realizados no ramo MARIA foram adicionados ao master sem criar ramificações.

- Pode agora empurrar o master para o remoto.

### 2.5.17 Rebase (Aluno A)

O Aluno A também que colocar o seu trabalho no repositório remoto. Vai usar a mesma técnica de rebase. Primeiro vai fazer trazer o trabalho do seu colega para o seu master :

```
git checkout master
git pull
```

- Passar para o ramo JOAO.
- Correr o comando

```
git rebase master
```

Pode ver que o antes do último commit realizado neste ramo, foram introduzidos os commits (do ramo master) feitos pelos seu colega.

- Para levar essas alterações para o master:

```
git checkout master
git merge JOAO
```

Pode ver que agora os commits efetuados pelos dois membros da equipa inscrevem-se numa sequência de commit linear.

- Pode agora empurrar as modificações para o remoto.

### 2.5.18 Nova fase no projeto (Aluno B)

Inicia-se uma nova fase no projeto. Ambos os alunos vão participar neste novo objetivo, contribuindo cada um no seu ramo antes de juntar o trabalho efetuado de ambas as partes. Esta fase é um pouco complexa e os alunos decidem não juntar logo o código desenvolvido neste contexto ao **master**. No entanto querem manter o desenvolvimento no repositório remoto. A solução encontrada foi criar um **novo ramo FASE2**. Ao contrário dos ramos pessoais usados anteriormente, este ramo **vai existir no repositório remoto**.

O Aluno B efetua os passos seguintes :

- Criar um ramo FASE2.
- Torna o ramo MARIA o ramo corrente.
- Criar um pacote `persist`
- Criar neste pacote a interface `IDataMapper<E>` com as assinaturas seguintes:

```
Optional<E> find(int id);
void insert(E e);
E update(E e);
void delete(E e);
```

`copy`

- Fazer commit.
- O trabalho do Aluno A continua, ao longo deste tempo pode fazer vários commits no sentido de avançar na implementação deste pacote, sempre para o seu ramo pessoal.
- Após alguns commits chega a conclusão que esta parte está concluída.
- Combinou com o colega que as novas adições que fazem parte da fase 2 do projeto deveriam ser revistas pelo colega antes de passar para o ramo master.
- O Aluno A coloca as suas alterações no ramo FASE2 usando um merge.

```
git checkout FASE2
git merge MARIA
```

- Para rever o código, deve empurrar o commit (e o ramo) para o remoto:

```
git push origin FASE2
```

- Verificar no gitlab a presença do novo ramo.
- Note que o ramo master está inalterado.

### 2.5.19 Aluno A participa no desenvolvimento do ramo FASE2

O Aluno A quer agora participar no desenvolvimento do ramo FASE2 que não existe ainda no seu repositório local. Para que o ramo FASE2 seja conhecido no local, deve usar o comando :

```
git fetch
```

a seguir pode passar a usar o ramo FASE2 :

```
git checkout FASE2
```

Da mesma forma que o seu colega, o Aluno A pode continuar o desenvolvimento fazendo vários commit pelo caminho. Admitindo que o trabalho para a fase 2 está mesmo terminado, pode ser incluído no master:

```
git checkout master
git merge FASE2
git push origin master
```

Note que o último comando empurrou apenas o master. As modificações efetuadas no ramos FASE2 não foram colocadas no remoto. Se a equipa tencionava continuar a trabalhar no ramo FASE2, o Aluno A deve empurrar os commits deste ramo para o remoto. Pode ser feito com:

```
git push origin/FASE2 FASE2
```

### 2.5.20 Git tag (Aluno B)

Para obter a ultima versão do trabalho o Aluno B deve usar os comandos:

```
git checkout master
git pull
```

O seu ramo master está agora sincronizado com origin/master. Caso o Aluno B quer continuar a trabalhar no ramo FASE2 deve também o sincronizar :

```
git checkout FASE2
git merge origin/FASE2
```

Se o trabalho no ramo FASE2 estiver mesmo terminado, talvez valia a pena marcar este ponto na história do repositório. É o papel do comando `git tag`. O commando cria um tag que é um nome associado ao commit. Facilita a referência a este ponto na história do desenvolvimento do projeto. Um tag é acompanhado de uma mensagem (como os commit). O Aluno B pode portanto usar o comando:

```
git tag -a guiaogit2 -m "Acabei finalmente o guião !"
```

Para que o tag seja visível dos outros utilizadores, tem de ser empurrado para o remoto:

```
git push origin guiaogit2
```

### 2.5.21 Acabou !

Uma vez terminado este guião, deve dar acesso ao utilizador `css-1ti-000` com as permissões “Developer”.

### 2.5.22 Problemas e soluções

Apagar um ramo vazio p.[52](#) • Target option 1.5 is no longer supported p.[52](#)  
 • GitLab: You are not allowed to push code to protected branches on this project p.[52](#) • O eclipse modifica a sua área de trabalho sem avisar p.[52](#)

## Apagar um ramo vazio

Criou um ramo mas enganou-se no nome. Quer apaga-lo. Pode usar:

```
git branch -d nome-do-ramo-a-apagar
```

## Target option 1.5 is no longer supported

Se obtiver o erro do tipo:

```
[ERROR] Source option 5 is no longer supported. Use 6 or later.
[ERROR] Target option 1.5 is no longer supported. Use 1.6 or later.
```

pode ser resolvido adicionando ao ficheiro pom.xls:

```
<properties>
 <maven.compiler.source>11</maven.compiler.source>
 <maven.compiler.target>11</maven.compiler.target>
</properties>
```

copy

para passar a usar o java 11 ou :

```
<properties>
 <maven.compiler.source>1.8</maven.compiler.source>
 <maven.compiler.target>1.8</maven.compiler.target>
</properties>
```

copy

para passar a usar o java 8.

## GitLab: You are not allowed to push code to protected branches on this project

Por omissão, o git lab “protege” o ramo master. Para além de adicionar os seus colegas aos membros do seu repositório enquanto “Developpers” tem de configurar o repositório de forma a permitir os colegas alterar o master.

Na página “Settings” do seu repositório escolher **Repository > Protected Branches** > aparece a lista dos ramos protegidos. Escolhe a opção “unprotect” para o ramo master. Agora os seus colegas podem fazer push para o master.

## O eclipse modifica a sua área de trabalho sem avisar

O eclipse altera sem aviso alguns ficheiros de configuração como o ficheiro .classpath ou ficheiros na pasta .settings. Pode acontecer que se esqueça de incluir esses ficheiros num commit e posteriormente esta situação venha a provocar a rejeição de um merge (por exemplo) :

```
Auto-merging .settings/org.eclipse.jdt.core.prefs
CONFLICT (content): Merge conflict in .settings/org.eclipse.jdt.core.prefs
Auto-merging .classpath
CONFLICT (content): Merge conflict in .classpath
Automatic merge failed; fix conflicts and then commit the result.
```

Nesta altura os ficheiros foram modificados (pelo git) de forma a assinalar as áreas de conflito. **É necessário resolver a situação manualmente editando os ficheiros em causa.** Se deixar os ficheiros na sua versão alterada pelo git, os ficheiros de configuração estão corruptos e nada de bom vai acontecer pela frente. Se está a hesitar e não sabe como resolver os conflitos, fique com uma cópia dos originais numa pasta separada.

## 2.6 Exercícios

Estratégia usada pelo Git p.53 • Ooops ! p.53 • Grafos dirigidos p.53  
 • Grafo p.54 • git config p.54 • Primeiros passos p.54 • git log, git diff p.55 • git branch p.56 • git commit p.58 • git remote p.58 • git revert p.59  
 • git show p.62

### 2.6.1 Estratégia usada pelo Git

Os sistemas de controlo de versões podem adotar uma estratégia de gestão de concorrência otimista ou pessimista. Indique qual a estratégia adotada pelo Git e explique em linhas gerais em que consiste.

### 2.6.2 Ooops !

Suponha que acabou de descobrir um erro no trabalho de CSS entregue e que o docente aceita que entregue uma nova versão. Que operações Git vai ter de realizar depois de corrigir o erro? Explique o que faz cada operação.

### 2.6.3 Grafos dirigidos

Nos sistemas de controle de versões são usados grafos dirigidos acíclicos (DAG). Existem interfaces gráficas que mostram, para além do grafo outras informações:

| Graph | Description             | Date          | Author     | Commit   |
|-------|-------------------------|---------------|------------|----------|
| o     | master   origin         | 18 Apr 202... | satheesh v | 29bda252 |
| o     | v2.0   bug fix 02       | 18 Apr 202... | satheesh v | f06f9db4 |
| o     | f03 ready for QA        | 18 Apr 202... | satheesh v | 44aa1c1e |
| o     | v1.0   bug fix 01 on qa | 18 Apr 202... | satheesh v | c3879230 |
| o     | QA drop                 | 18 Apr 202... | satheesh v | 38f571a6 |
| o     | feature/f03   origin    | 18 Apr 202... | satheesh v | 3976b0cc |
| o     | feature/f02   origin    | 18 Apr 202... | satheesh v | cae40cba |
| o     | feature/f01   origin    | 18 Apr 202... | satheesh v | 6e2e9bd4 |
| o     | first commit            | 18 Apr 202... | satheesh v | c664b199 |
| o     | Init                    | 18 Apr 202... | satheesh v | 91c4bbab |

- O que é que representam as bolas no grafo ? as arestas ?
- O que é que representam as strings na coluna da direita ?
- Quem é o autor dos commits ?
- Qual é o ID do commit mais antigo ? do mais recente ?
- Qual é o significado das cores no grafo ?
- Qual é o significado da etiqueta associada aos commit 3976b0cc, cae40cba, e 6e2e9bd4 ?
- Porque está indicado “origin” nessas etiquetas ?
- Qual é o significado das etiquetas “v1.0” e “v2.0” ?
- Qual é o significado da etiqueta “master|origin” ?

## 2.6.4 Grafo

Suponha que, com um grupo de colegas, está a trabalhar numa aplicação que tem dois componentes que concretizam duas UIs diferentes da aplicação, uma web e outra desktop. Suponha ainda que

1. um dos seus colegas lhe pede para rever as alterações ao componente desktop que ele colocou no repositório do servidor do GitLab e
  2. concorrentemente você fez já um commit no seu repositório local com alterações que afetam apenas o componente web, mas ainda não empurrou este commit para o repositório no servidor.
- Que operações Git vai realizar para fazer essa revisão e disponibilizar para o grupo o seu resultado ?
  - Mostre, através de um grafo, quais os efeitos que essas operações têm no seu repositório local.
  - E quais os efeitos dessas operações na sua cópia de trabalho (working directory) ?

## 2.6.5 git config

The git config command lets you configure your Git installation (or an individual repository) from the command line. This command can define everything from user info to preferences to the behavior of a repository

```
git config --global user.name <name>
git config --global user.email <email>
git config --system core.editor nano
```

Esta última configuração especifica o editor usado pelo git. O editor será usado cada vez que é necessário fornecer algum input (por exemplo as mensagens de commit).

As vezes um comando git mostra muitas linhas. Pode optar por usar um “pager” para que a saída dos comandos sejam cortadas em páginas :

```
git config --global core.pager less
```

Se pelo contrário não quer que a saída dos comandos seja cortada pode usar:

```
git config --global core.pager cat
```

## 2.6.6 Primeiros passos

Um repositório local pode ser criado assim:

```
> mkdir css_lrep
> cd css_lrep/
> git init
```

```
Initialized empty Git repository in
/Users/Mario/css_lrep/.git/
```

Criar ficheiros na área de trabalho:

```
> cd css_lrep
> echo 'Bem vindos!' > readme.txt
> echo 'Joao Silva, Ana Reis, TOCOMPLETE' > alunos.txt
```

1. O que contém a *working directory* e a *staged area* após a execução dos dois passos anteriores?
2. E depois de executar cada uma das instruções abaixo

```
> git add readme.txt alunos.txt
> git commit -m "colocados primeiros ficheiros"
```

```
[master (root-commit) dda55c7] colocados primeiros ficheiros
2 files changed, 2 insertions(+)
create mode 100644 alunos.txt
create mode 100644 readme.txt
```

3. Utilize o comando `git status` para ver se a sua resposta está correcta.
4. Qual o identificador do *commit object* criado? E a mensagem?
5. O que aconteceria se entre o `git add` e o `git commit` tivesse modificado um dos dois ficheiros?

Após a criação do ficheiros:

```
> git status
```

```
On branch master
Initial commit
Untracked files:
 (use "git add <file>..." to include in what will be committed)
alunos.txt
readme.txt
nothing added to commit but untracked files present (use "git add" to track)
```

Depois de usar o comando `git add`:

```
> git status
```

```
On branch master
Initial commit
Changes to be committed:
 (use "git rm --cached <file>..." to unstage)
new file: alunos.txt
new file: readme.txt
```

A seguir ao `git commit`:

```
> git status
```

```
On branch master
nothing to commit, working directory clean
```

## 2.6.7 git log, git diff

Depois de mais alguns passos foi usada a operação `git log` para ver o histórico do repositório. Interprete o que se passou.

```
> git log
```

```
commit c80649daadb5363630a99c845f7c02d4752c121b
Author: Antonia Lopes <mal@di.fc.ul.pt>
Date: Thu Sep 10 23:54:20 2015 +0100
 melhorado o readme
commit dda55c7b16f1d30b1870eaaf556e0f4b1bba8f8e
Author: Antonia Lopes <mal@di.fc.ul.pt>
Date: Thu Sep 10 23:44:14 2015 +0100
 colocados primeiros ficheiros
```

```
> git status
```

```
On branch master
nothing to commit, working directory clean
```

A operação `git diff` permite saber com mais detalhe o que se passou. Interprete o resultado do comando dado.

```
> git diff c80649d dda55c7b readme.txt
```

```
diff --git a/readme.txt b/readme.txt
index a4849bf..ca16e06 100644
--- a/readme.txt
+++ b/readme.txt
@@ -1 +1 @@
-Bem vindos à disciplina de CSS!
+Bem vindos!
```

A seguir foi dada mais uma sequência de comandos. Explique o efeito de cada um desses comandos sobre a *working directory*, a *staging area* e o repositório.

```
> echo 'uau, o git arrasa!' >> readme.txt
> git add readme.txt
> git commit -m "readme mais colorido"
```

```
[master (root-commit) c4937cb] readme mais colorido
1 file changed, 1 insertion(+)

> git log
```

```
commit c4937cb585e336d799b8868f4302a1777646d208
Author: Antonia Lopes <mal@di.fc.ul.pt>
Date: Fri Sep 11 09:56:31 2015 +0100
 readme mais colorido
commit c80649daadb5363630a99c845f7c02d4752c121b
Author: Antonia Lopes <mal@di.fc.ul.pt>
Date: Thu Sep 10 23:54:20 2015 +0100
 melhorado o readme
commit dda55c7b16f1d30b1870eaaf556e0f4b1bba8f8e
Author: Antonia Lopes <mal@di.fc.ul.pt>
Date: Thu Sep 10 23:44:14 2015 +0100
 colocados primeiros ficheiros
```

## 2.6.8 git branch

Examine e explique a sequência de comandos seguinte.

```
> git branch
* master
> git branch aula1
> git branch
```

```
aula1
* master
```

```
> git checkout aula1
Switched to branch 'aula1'
> git branch

* aula1
 master

> echo 'ex1' > ex1.txt
> git add ex1.txt
> echo 'ex2' > ex2.txt
> git add ex2.txt
> git commit -m "adicionados exercícios 1 e 2"
...
> echo 'ex3' > ex3.txt
> git add ex3.txt
> git commit -m "adicionados exercícios 3"
...
> git checkout master
```

Switched to branch 'master'

```
> ls
```

```
alunos.txt readme.txt

> echo 'ler com muita atencao' > regras.txt
> git add regras.txt
> git commit -m "adicionadas regras"
```

```
[master 8d517fe] adicionadas regras
1 file changed, 1 insertion(+)
create mode 100644 regras.txt
```

```
> ls
```

```
alunos.txt readme.txt
regras.txt
```

```
> git merge aula1
```

```
Merge made by the 'recursive' strategy.
ex1.txt | 1 +
ex2.txt | 1 +
ex3.txt | 1 +
3 files changed, 3 insertions(+)
create mode 100644 ex1.txt
create mode 100644 ex2.txt
create mode 100644 ex3.txt
```

```
> git checkout aula1
Switched to branch 'aula1'
> ls
```

```
alunos.txt ex1.txt ex2.txt ex3.txt readme.txt
```

Switched to branch 'master'

```
> ls
```

```
alunos.txt ex1.txt ex2.txt ex3.txt readme.txt regras.txt
```

```
> git log
```

```
commit 88d46b138114f84329c5c89bc2c4781361153366
Merge: 8d517fe 36285ac
Author: Antonia Lopes <mal@di.fc.ul.pt>
Date: Fri Sep 11 11:16:23 2015 +0100
 Merge branch 'aula1'
commit 8d517fe267d23833cebe6adab637f27c7b3da352
Author: Antonia Lopes <mal@di.fc.ul.pt>
Date: Fri Sep 11 11:15:07 2015 +0100
 adicionadas regras
commit 36285ac3d30c22b72603d0b1b00e4892e2e19853
Author: Antonia Lopes <mal@di.fc.ul.pt>
Date: Fri Sep 11 11:13:32 2015 +0100
 adicionados exercícios 3
commit 1be52416099c47c7370ecdcafe2f207c264b51ee
Author: Antonia Lopes <mal@di.fc.ul.pt>
Date: Fri Sep 11 11:12:48 2015 +0100
 adicionados exercícios 1 e 2
commit c80649daadb5363630a99c845f7c02d4752c121b
Author: Antonia Lopes <mal@di.fc.ul.pt>
Date: Thu Sep 10 23:54:20 2015 +0100
 melhorado o readme
commit dda55c7b16f1d30b1870eaaf556e0f4b1bba8f8e
Author: Antonia Lopes <mal@di.fc.ul.pt>
Date: Thu Sep 10 23:44:14 2015 +0100
 colocados primeiros ficheiros
```

## 2.6.9 git commit

O command `git commit` pode ser usado com as opções `-am`:

```
git commit -am "commit message"
```

Neste caso, todos ficheiros que fazem parte do repositório (para os quais já usou o commando `git add`) e que foram modificados serão incluídos no commit. Não é necessário correr outra vez o comando `git add`.

## 2.6.10 git remote

O “remote” designa o repositório remoto, na maioria dos casos trata-se do repositório principal que serve de referência. Todos os membros da equipa sincronisam o seu repositório local com este. Nos comandos `git` é referido por “**origin**”.

Para conhecer o endereço do repositório remoto pode usar o comando:

```
git remote -v
```

**Exemplo:**

```
git remote -v
origin git@github.com:manuelaribeiro/myproject.git (fetch)
origin git@github.com:manuelaribeiro/myproject.git (push)
```

Naturalmente quando um repositório é criado com `git init`, não tem remoto associado:

```
> git init
Initialized empty Git repository in /tmp/myproject/.git/
> git remote -v
... não dá nada !
```

Quando um repositório é criado com o comando `git clone` o remoto é automaticamente configurado com o endereço do repositório de origem.

É possível haver vários repositórios remotos associados a um repositório local. Não havendo um caso de uso para esta possibilidade (nesta fase de conhecimento), não vamos usa-la. Nessas condições o repositório remoto é隐式的. Não será necessário indicar o repositório remoto nos comandos `push`, `pull` e `fetch`. Por exemplo, `git push` vai empurrar o vosso repositório local para `origin`.

É possível atribuir um nome para o(s) repositório(s) remotos. Por exemplo pode renomiar o `origin` para mais clareza:

```
git remote rename origin gitlabdi
```

A seguir a referência ao remoto ser feita com a palavra “`gitlabdi`” e não “`origin`”.

A definição do remoto está guardada no ficheiro `.git/config`:

```
$ cat .git/config
[core]
repositoryformatversion = 0
filemode = true
bare = false
logallrefupdates = true
[remote "origin"]
url = git@github.com:manuelaribeiro/test.git
fetch = +refs/heads/*:refs/remotes/origin/*
[branch "master"]
remote = origin
merge = refs/heads/master

$ git remote rename origin github
$ cat .git/config
[core]
repositoryformatversion = 0
filemode = true
bare = false
logallrefupdates = true
[remote "github"]
url = git@github.com:manuelaribeiro/test.git
fetch = +refs/heads/*:refs/remotes/github/*
[branch "master"]
remote = github
merge = refs/heads/master
```

## 2.6.11 git revert

O comando `git revert` pode ser considerado um comando do tipo ‘desfazer’, no entanto, não é uma operação de desfazer tradicional. Em vez de remover o commit da história do repositório, descobre como inverter as alterações introduzidas pelo commit e acrescenta um novo commit com o conteúdo inverso resultante. Isto impede git de perder a história, o que é importante para a integridade da sua história de revisão e para uma colaboração confiável.

A reversão deve ser usada quando você deseja aplicar o inverso de um commit do histórico do seu projeto. Isso pode ser útil, por exemplo, se você estiver rastreando um bug e descobrir que ele foi introduzido por um único commit. Em vez de entrar manualmente, corrigi-lo e confirmar um novo snapshot, você pode usar o `git revert` para fazer tudo isso automaticamente para você.

O comando `git revert` é usado para desfazer alterações no histórico de commits de um repositório. Outros comandos ‘undo’, como `git checkout` e `git reset`, movem os ponteiros HEAD e branch ref

para um commit especificado. O comando `git revert` também aceita um commit específico, no entanto, `git revert` não move os ponteiros de ref para este commit. Uma operação de reversão pegará o commit especificado, inverterá as alterações desse commit e criará um novo “revert commit”. Os ponteiros de referência são atualizados para apontar para o novo commit de reversão, tornando-o a ponta do ramo corrente.

Passo um: Criamos um repositório e efetuamos três commits:

```
$ git init
Initialized empty Git repository in /tmp/gitrevert/.git/
$ echo "Ola" > README.md
$ git add .
$ git commit -m "First commit"
[master (root-commit) 7d71ff3] First commit
 1 file changed, 1 insertion(+)
 create mode 100644 README.md
$ echo "Mais uma linha" >> README.md
$ git commit -am "Added one line to README."
[master a3d3411] Added one line to README.
 1 file changed, 2 insertions(+)
$ echo "mias uam lniha" >> README.md
$ git commit -am "Added one more line to README."
[master 253d191] Added one more line to README.
 1 file changed, 1 insertion(+)
$ git log --oneline
253d191 (HEAD -> master) Added one more line to README.
a3d3411 Added one line to README.
7d71ff3 First commit
```

O terceiro commit contém um erro, queremos reverter a situação para o commit anterior:

```
$ git revert HEAD
hint: Waiting for your editor to close the file...
Waiting for Emacs...
[master ccac1c42] Revert "Added one more line to README."
 1 file changed, 1 deletion(-)
```

O git pega no commit indicado e descobre uma maneira de modificar o ficheiro README.md de forma a apagar as modificações introduzidas pelo commit. Resumindo, basta remover a última linha:

```
Staged changes
 1 file changed, 1 deletion(-)
 README.md | 1 -
modified README.md
@@ -1,4 +1,3 @@
 Ola
 Mais uma linha
-mias uam lniha
```

O comando anterior deve abrir um editor para permitir a definição da mensagem que será associada ao novo commit (para a escolha do editor, ver a secção `git config`).

O novo commit criado:

```
$ git log --oneline
ccac1c42 (HEAD -> master) Revert "Added one more line to README."
253d191 Added one more line to README.
a3d3411 Added one line to README.
7d71ff3 First commit
```

Observe que o 3º commit ainda está no histórico do projeto após a reversão. Em vez de excluí-lo, `git revert` adicionou um novo commit para desfazer suas alterações. Como resultado, o 2º e o 4º commits representam exatamente a mesma base de código e o 3º commit ainda está em nosso histórico caso queiramos voltar a ele no futuro.

No exemplo anterior o commit a corrigir era o último. Será que pode funcionar com qualquer commit ? Uma serie de comandos é efetuada de forma a acrescentar os commits b6972a7 5a74dfa e 80cfb15:

```
$ git log --oneline
80cfb15 (HEAD -> master) On more line (correct).
5a74dfa This commit contains an error.
b6972a7 more work on README.
ccalc42 Revert "Added one more line to README."
253d191 Added one more line to README.
a3d3411 Added one line to README.
7d71ff3 First commit
```

Para cancelar o commit 5a74dfa vamos usar o **git revert**:

```
$ git revert 5a74dfa
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
error: could not revert 5a74dfa... This commit contains an error.
hint: After resolving the conflicts, mark them with
hint: "git add/rm <pathspec>", then run
hint: "git revert --continue".
hint: You can instead skip this commit with "git revert --skip".
hint: To abort and get back to the state before "git revert",
hint: run "git revert --abort".
```

Não correu muito bem. O git não foi capaz resolver o conflito. Podemos ver o problema :

```
$ git diff
diff --cc README.md
index 4fee4e1,cbc379b..0000000
--- a/README.md
+++ b/README.md
@@@ -2,5 -2,3 +2,8 @@@ 01

Mais uma linha
outra linha
++<<<<< HEAD
+esta linha foi acrescentada por engano
+Esta linha é legitima.
=====
++>>>>> parent of 5a74dfa (This commit contains an error.)
```

A solução consiste em editar o ficheiro para corrigi-lo. Após edição fica assim:

```
$ cat README.md
Ola

Mais uma linha
outra linha
Esta linha é legitima.
```

As modificações devem agora estar incluídas num commit:

```
$ git add .
$ git revert --continue
hint: Waiting for your editor to close the file...
Waiting for Emacs...
[master f313134] Revert "This commit contains an error."
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Foi necessário indicar uma mensagem para o commit (dexei a mensagem por omissão).

```
$ git log --oneline
f313134 (HEAD -> master) Revert "This commit contains an error."
80cfb15 On more line (correct).
5a74dfa This commit contains an error.
b6972a7 more work on README.
ccalc42 Revert "Added one more line to README."
253d191 Added one more line to README.
a3d3411 Added one line to README.
7d71ff3 First commit
```

A reversão (`git revert`) tem duas vantagens importantes sobre `git reset`. Primeiro, ele não altera o histórico do projeto, o que o torna uma operação “segura” para commits que já foram publicados em um repositório partilhado.

O comando `git revert` é uma operação de desfazer que provoca a criação de um commit. Oferece um método seguro de desfazer alterações. Em vez de apagar ou deixar commits órfãos no histórico, uma reversão criará um novo commit que inverte as alterações especificadas. O comando `git revert` é uma alternativa mais segura para `git reset` em relação à perda de trabalho.

## 2.6.12 git show

O comando `git show` é bastante versátil. Um dos casos de usos é ver as versões antigas de um ficheiro. Por exemplo, usando o repositório do exercício “git revert” :

```
$ git log --oneline
f313134 (HEAD -> master) Revert "This commit contains an error."
80cfb15 On more line (correct).
5a74dfa This commit contains an error.
b6972a7 more work on README.
ccalc42 Revert "Added one more line to README."
253d191 Added one more line to README.
a3d3411 Added one line to README.
7d71ff3 First commit
```

Podemos ver como era o ficheiro README.md no commit 253d191 que foi revertido :

```
$ git show 253d191:./README.md
Ola

Mais uma linha
mias uam lniha
```

Ou no commit que continha um erro (5a74dfa):

```
$ git show 5a74dfa:./README.md
Ola

Mais uma linha
outra linha
esta linha foi acrescentada por engano
```

Se por exemplo quer recuperar um pedaço de código numa versão antiga de uma classe (que foi apagado entretanto) pode fazer :

```
git show 459fd4a:/src/mypackage/MyClass.java > /tmp/MyClassOLD.java
```

A seguir pode editar a vontade o ficheiro criado fora do seu repositório.

# Capítulo 3

## Desenvolvimento em camadas

Padrões de software p.63 • Aplicações Empresariais p.63 • Organização em camadas p.64 • Padrões para as várias camadas p.66 • Exercícios p.67

### 3.1 Padrões de software

Um padrão (do inglês, *pattern*) corresponde a uma solução geral para um problema comum na engenharia de software.

- Não corresponde a código. É uma ideia, uma receita para abordar o problema em questão
- Como provérbios que guiam o programador para soluções concretas
- Alguns desses padrões fazem parte das próprias linguagens (e.g., o padrão *Module* corresponde aos pacotes Java)

Livros como o do Martin Fowler organizam listas de padrões considerados úteis na construção de aplicações complexas.

Mas as aplicações não são apenas coleções de padrões...

### 3.2 Aplicações Empresariais

Os padrões estudados nesta disciplina encontrem o seu domínio de aplicação nos sistemas de software empresariais. A maior parte dos recursos gastos a desenvolver software é em aplicações empresariais. Eles se concentram na exibição, manipulação e armazenamento de grandes quantidades de dados (muitas vezes complexos) e no apoio à automação de processos de negócios com esses dados. Por exemplo:

- CRM Customer Relationship Management
- SCM Supply Chain Management
- SFA Sales Force Automation
- ERP Enterprise Resource Planning

As aplicações empresariais têm características específicas e desafios próprios:

- Desafios Tecnológicos
  - Muitos dados persistentes
  - Acessos concorrentes
  - Muitos ecrãs de interface com utilizador

- Escalabilidade e Segurança

- **Desafios de Negócio**

- Inconsistências entre os conceitos do negócio e os processos
- Lógica de domínio complexa, difícil de capturar

As soluções evoluíram a par com os avanços tecnológicos. As diferenças mais importantes são ao nível da **arquitetura do software**, ou seja, nos **elementos** de software em que se baseia a solução e nas **relações** existentes entre eles.

Particularmente relevante são:

- as **camadas** (layers) usadas para, do ponto de vista lógico, decompor as funcionalidades da aplicação
- a forma como estas camadas são, do ponto de vista de processos, combinadas e distribuídas em diferentes **níveis**.

### 3.3 Organização em camadas

O desenvolvimento de software em camadas é uma abordagem de design e arquitetura de software que envolve a **separação do sistema em diferentes camadas ou níveis funcionais**, cada uma com responsabilidades específicas. O código do sistema é organizado numa **pilha de camadas**, cada uma encapsulando um grupo de funcionalidades num particular nível de abstração :

- A camada  $k$  oferece serviços à  $k + 1$  servindo-se da  $k - 1$  e não tem qualquer dependência de camadas superiores
- A camada 1, no **nível de abstração mais baixo**, é a base/núcleo do sistema
- A camada no topo fornece as **funcionalidades de alto nível**.

O desenvolvimento de software em camadas oferece várias vantagens:

- **Separação de preocupações**: Cada camada tem um **conjunto específico de responsabilidades**, o que torna o software mais organizado e mais fácil de compreender. Isso facilita a manutenção, a depuração e a escalabilidade.
- **Reusabilidade de código**: As camadas bem definidas permitem a **reutilização de código** em diferentes partes do sistema ou em projetos futuros. Isso economiza tempo e esforço de desenvolvimento.
- **Facilidade de manutenção**: Como as camadas estão separadas, as **mudanças em uma camada geralmente não afetam as outras**. Isso torna as atualizações e correções de bugs mais fáceis de realizar sem impactar todo o sistema.
- **Escalabilidade**: É **mais simples dimensionar um sistema** em camadas, pois você pode adicionar ou remover camadas conforme necessário para lidar com requisitos crescentes de carga ou funcionalidades adicionais.
- **Testabilidade**: A separação de camadas facilita a **criação de testes unitários** e de integração, pois você pode isolar e testar cada camada separadamente. Isso ajuda a garantir a qualidade do software.
- **Colaboração**: Equipes de desenvolvimento podem **trabalhar em paralelo** em diferentes camadas do sistema, o que melhora a eficiência e acelera o desenvolvimento.
- **Segurança**: Uma arquitetura em camadas permite a **implementação de medidas de segurança** em camadas específicas, protegendo melhor o sistema como um todo.
- **Flexibilidade tecnológica**: Pode usar diferentes tecnologias em cada camada, escolhendo aquelas mais adequadas para a tarefa em questão. Isso possibilita a adoção de tecnologias mais recentes e melhores práticas de desenvolvimento.

- **Isolamento de falhas:** Se ocorrer um problema em uma camada, ele tende a afetar apenas essa camada, **minimizando o impacto nas outras partes** do sistema.
- **Gestão da complexidade:** Dividir o sistema em camadas **torna-o mais fácil gerir**, especialmente em sistemas grandes e complexos, facilitando a compreensão da estrutura e o planejamento do desenvolvimento.

Em resumo, o desenvolvimento em camadas é uma abordagem que promove a **modularidade**, a **reutilização** e a **manutenção** eficiente do software, tornando-o mais **robusto** e **flexível**. No entanto, é importante projetar as camadas cuidadosamente e definir interfaces claras entre elas para colher plenamente esses benefícios.

Embora o desenvolvimento em camadas ofereça muitas vantagens, também apresenta algumas desvantagens, especialmente em certos contextos ou quando não é implementado corretamente. Aqui estão algumas das desvantagens mais comuns do desenvolvimento em camadas:

1. **Complexidade inicial:** A criação de várias camadas requer mais tempo e esforço na fase inicial do projeto. É necessário planejamento e análise cuidadosos para definir as interfaces e as responsabilidades de cada camada.
2. **Overhead de comunicação:** Em sistemas com muitas camadas, a comunicação entre essas camadas pode introduzir algum overhead de desempenho. Isso ocorre devido à necessidade de passar dados e mensagens entre as camadas, o que pode impactar o desempenho em sistemas de alto tráfego.
3. **Maior consumo de recursos:** Cada camada adiciona uma sobrecarga adicional de recursos, como memória e CPU. Em sistemas com recursos limitados, isso pode ser um problema.
4. **Complexidade de depuração:** Em sistemas em camadas, pode ser mais complexo depurar problemas, pois é necessário acompanhar o fluxo de dados e a lógica através de várias camadas. Identificar a origem de um erro pode ser desafiador.
5. **Sobrecarga de gestão:** Gerir várias camadas pode aumentar a complexidade da administração do sistema, incluindo a implantação e a manutenção. Isso também pode resultar em custos operacionais mais altos.
6. **Possível degradação de desempenho:** O desenvolvimento em camadas pode aumentar a complexidade do código, especialmente se houver muitas camadas interdependentes. Isso pode dificultar a compreensão do sistema.
7. **Aumento da complexidade do código:** O desenvolvimento em camadas pode aumentar a complexidade do código, especialmente se houver muitas camadas interdependentes. Isso pode dificultar a compreensão do sistema.
8. **Potencial para acoplamento excessivo:** Se as camadas não forem adequadamente desacopladas e as interfaces não forem bem definidas, pode haver acoplamento excessivo entre as camadas, o que torna as mudanças em uma camada mais difíceis de implementar sem afetar outras partes do sistema.
9. **Custo de desenvolvimento:** A implementação de um sistema em camadas geralmente requer mais recursos de desenvolvimento do que abordagens mais simples, o que pode aumentar os custos de desenvolvimento.
10. **Complexidade de escalabilidade:** À medida que um sistema em camadas cresce, a escalabilidade pode se tornar um desafio, especialmente se não houver uma estratégia clara de escalonamento e distribuição.

As camadas usadas em uma arquitetura de software podem variar dependendo do tipo de aplicação e dos requisitos específicos do projeto. No entanto, muitas arquiteturas de software adotam uma abordagem de **três camadas básicas**:

- **Camada de Lógica de Negócios (Lógica de Aplicação):**

- Esta camada **contém a lógica de negócios** da aplicação. Processa dados, realiza cálculos, aplica regras de negócios e toma decisões com base nas entradas da camada de apresentação.

- É independente da interface do utilizador e pode ser reutilizada por várias interfaces de apresentação.
- Também é conhecida como camada de serviço ou camada de aplicação.

- **Camada de Dados (Acesso a Dados):**

- Esta camada lida com o armazenamento e a recuperação de dados. Ela interage com bases de dados, sistemas de arquivos ou serviços externos para armazenar e recuperar informações.
- Geralmente, fornece uma abstração para aceder aos dados, o que permite que a camada de lógica de negócios seja independente da fonte de dados específica.
- Também é chamada de **camada de persistência**.

- **Camada de Apresentação (Interface com o Utilizador):**

- Esta camada lida com a interação direta com o utilizador. É responsável pela apresentação de informações, coleta de entrada do utilizador e exibição de resultados.
- Pode incluir interfaces com o utilizador, como GUIs (interfaces gráficas do usuário), páginas da web ou APIs que interagem com aplicações front-end.
- Seu principal objetivo é fornecer uma experiência amigável e eficiente ao utilizador.

Com o aparecimento das tecnológicas da internet surgiu a necessidade de transportar os sistemas cliente-servidor para os browsers. Com um sistema de três camadas era mais fácil adaptar-se às diferentes UI's existentes.

Existem várias formas de implementar as responsabilidades de cada uma destas camadas, usando padrões de software.

Estes fazem parte do catálogo de *Patterns of Enterprise Application Architecture*.

#### Onde executar as diferentes camadas ?

As opções normais são entre o **servidor** e um **cliente** no computador do utilizador. Existe a opção de correr tudo no servidor. Usa-se o browser apenas como uma interface HTML para a interação com o utilizador:

- Tem a vantagem de ser fácil de manter e atualizar.
- Tem a desvantagem dos tempos de resposta e da necessidade de conexão permanente.

Correr parte da aplicação no programa cliente garante alguma independência mas é necessária a sincronização dos estados entre o cliente e o servidor.

A respeito da execução das várias camadas:

- A camada de persistência encontra-se no lado do servidor.
- A camada de apresentação pode estar no lado do cliente (quando existe uma aplicação cliente relativamente complexa) ou do lado do servidor (como no uso dos browsers)
  - Para diminuir o fluxo de dados é normal a existência de scripts no HTML que realizam tarefas, como validação de dados
  - Segundo Fowler deve-se apostar numa apresentação web se for possível, ou numa apresentação no cliente se for exigido.
- A camada lógica é normal estar no servidor e apenas deve ser considerada no cliente quando a questão da conexão (ou falta dela) é relevante.

## 3.4 Padrões para as várias camadas

Para cada camada existem várias implementações possíveis que fazem uso dos padrões de software seguintes:

- Negócio
  - Transaction Script

- Table Module
- Domain Model
- Dados
  - Row data gateway
  - Table data gateway
  - Active Record
  - Data Mapper
- Apresentação
  - Front controller
  - Page controller
  - Page template

## 3.5 Exercícios

### 3.5.1 Quiz

1. O que é o desenvolvimento em camadas em engenharia de software?
  - (a) Um método para criar aplicações com uma única camada de código.
  - (b) Uma abordagem que divide um sistema de software em várias camadas funcionais ou lógicas.
  - (c) Um conceito que promove a mistura de código de front-end e back-end em um único arquivo.
2. Quais são as principais camadas em uma arquitetura típica de desenvolvimento em camadas?
  - (a) Interface Utilizador (UI), base de dados e servidor web.
  - (b) Apresentação, lógica de negócios e acesso a dados.
  - (c) Front-end, back-end e middleware.
3. Qual é o objetivo da camada de apresentação em uma arquitetura em camadas?
  - (a) Realizar operações de CRUD no base de dados.
  - (b) Apresentar informações ao utilizador e coletar entradas.
  - (c) Processar transações de pagamento.
4. Qual é o papel da camada de lógica de negócios?
  - (a) Gerir a interface utilizador e o design gráfico.
  - (b) Implementar a funcionalidade principal da aplicação e as regras de negócio.
  - (c) Fornecer acesso direto à base de dados.
5. O que a camada de acesso a dados faz numa arquitetura em camadas?
  - (a) Lida com a interface utilizador.
  - (b) Realiza cálculos matemáticos complexos.
  - (c) Gere a interação com a base de dados.



# Capítulo 4

## Padrões para a camada de negócio

Existem vários padrões para a implementação da camada de negócio. Nas seções seguintes vamos estudar as vantagens e desvantagens de cada um.

Transaction Script p.69 • Table Module p.72 • Domain Model p.76 • Como escolher ? p.80 • Exercícios p.81

### 4.1 Transaction Script

O **Transaction Script** é um padrão de design de software que se concentra em organizar a lógica de negócios **em torno das transações** ou operações do sistema. Ele é especialmente útil em **sistemas simples ou pequenos**, onde a complexidade não justifica a adoção de arquiteturas mais elaboradas, como o Modelo-Visão-Controlador (MVC) ou o Modelo de Camadas.

A ideia fundamental por trás do **Transaction Script** é organizar o código de maneira que cada transação ou operação específica seja tratada por um script separado. Aqui estão os principais conceitos associados ao **Transaction Script**:

- **Scripts Transacionais:** Cada transação ou operação importante do sistema é representada por um script. **Esses scripts contêm a lógica de negócios necessária para executar a transação**, incluindo validações, cálculos e interações com a base de dados.
- **Procedural:** O código dentro de cada script é geralmente organizado de forma **procedural**, ou seja, segue uma sequência linear de instruções. Isso torna o *Transaction Script* adequado para operações simples e diretas.
- **Foco em Transações:** O padrão *Transaction Script* concentra-se nas transações individuais e em suas regras de negócios associadas. Cada script é **responsável por uma tarefa específica**.
- **Não há Divisão Rígida:** Ao contrário de padrões como o Modelo-Visão-Controlador (MVC), o *Transaction Script* não exige uma divisão rígida entre a lógica de negócios e a apresentação. Em sistemas simples, as operações podem incluir diretamente a interação com a interface do usuário.
- **Facilidade de Entendimento:** O *Transaction Script* pode ser fácil de entender e é frequentemente usado em sistemas pequenos ou projetos de protótipo, onde a simplicidade e a rapidez de desenvolvimento são prioridades.

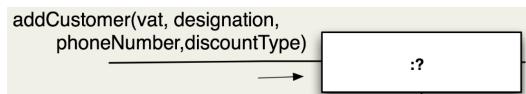
No entanto, o *Transaction Script* pode ter algumas desvantagens, especialmente em sistemas complexos:

- **Dificuldade de Manutenção:** Conforme o sistema cresce e se torna mais complexo, a manutenção pode se tornar problemática, pois não há uma separação clara entre a lógica de negócios e a apresentação.

- Repetição de Código:** Em sistemas com muitas transações semelhantes, pode haver repetição de código, pois cada script tende a incluir a mesma lógica de validação ou processamento de dados.
- Escalabilidade Limitada:** O padrão *Transaction Script* pode não ser adequado para sistemas que precisam de alta escalabilidade ou que requerem compartilhamento de lógica de negócios entre várias partes do sistema.

O padrão *Transaction Script* é uma abordagem simples e direta para organizar a lógica de negócios em torno de transações individuais em sistemas de software. Ele é adequado para sistemas pequenos e simples, mas pode se tornar menos eficaz em sistemas maiores e mais complexos, onde a separação de preocupações e a reutilização de código são mais importantes.

Exemplo: numa aplicação que entre outras tarefas gere uma carteira de clientes temos a operação `addCustomer`:

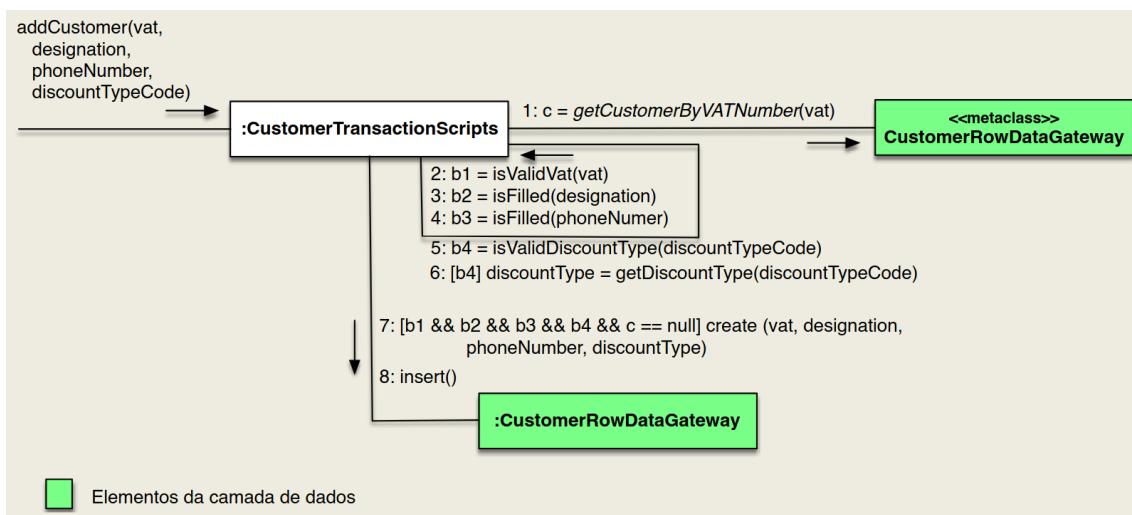


Operação `addCustomer` representada por um esboço de **Diagrama de Interação**.

O algoritmo a implementar é:

Se não houver um CLIENTE com este VAT na BD E  
 o CÓDIGO DE TIPO DE DESCONTO for válido E  
 o VAT for válido E  
 o NOME e TELEPHONE estiverem preenchidos:  
 então adicionar CLIENTE.

A tarefa é executada por um objeto de tipo *Transaction Script*. O script trata das validações e trata de criar e guarda o registo na base de dados (recorre ao serviço da camada de persistência).



O diagrama pode-se traduzir no código seguinte:

```

public void addCustomer(int vat, String denomination, int phoneNumber,
 int discountCode) throws ApplicationException {
 // Verificar VAT:
 if (!isValidVat(vat)) {
 throw new ApplicationException("Invalid VAT number: " + vat);
 }
 // Verificar o código de desconto:
 if (discountCode <= 0 || discountCode > DiscountType.values().length) {
 throw new ApplicationException("Invalid Discount Code :" + discountCode);
 }
 // Verificar denominação e telefone:
 if (!isFilled(denomination) || phoneNumber == 0) {
 }
}

```

```

 throw new ApplicationException("Both denomination and phoneNuber must be filled.");
}
// Instruções para criar e inserir o registo na BD:
copy

```

Segue um segundo exemplo de implementação do padrão *Transaction Script* para uma aplicação de gestão de inventário de produtos. O *Transaction Script* é uma abordagem em que as operações de negócios são tratadas diretamente em scripts ou classes, sem a necessidade de uma separação rígida de camadas ou uma representação de objetos mais complexa.

Neste exemplo, suponha que tenhamos uma classe `Produto` com atributos como nome, quantidade em armazém e preço. Vamos criar um *Transaction Script* para lidar com a adição de produtos ao armazém e o cálculo do valor total em reserva.

```

import java.util.HashMap;
import java.util.Map;

public class EstoqueTransactionScript {
 private static Map<String, Produto> estoque = new HashMap<>();

 // Método para adicionar um produto ao estoque
 public static void adicionarProduto(String nome, int quantidade, double preco) {
 if (estoque.containsKey(nome)) {
 // Se o produto já existe no estoque, atualiza a quantidade e o preço
 Produto produtoExistente = estoque.get(nome);
 produtoExistente.setQuantidade(produtoExistente.getQuantidade() + quantidade);
 produtoExistente.setPreco(preco);
 } else {
 // Caso contrário, cria um novo produto
 Produto novoProduto = new Produto(nome, quantidade, preco);
 estoque.put(nome, novoProduto);
 }
 }

 // Método para calcular o valor total em estoque
 public static double calcularValorTotalEmEstoque() {
 double valorTotal = 0;
 for (Produto produto : estoque.values()) {
 valorTotal += produto.getPreco() * produto.getQuantidade();
 }
 return valorTotal;
 }

 public static void main(String[] args) {
 // Adicionando produtos ao estoque
 adicionarProduto("Produto A", 10, 25.0);
 adicionarProduto("Produto B", 5, 15.0);
 adicionarProduto("Produto A", 3, 30.0); // Atualizando a quantidade e o preço do Produto A

 // Calculando o valor total em estoque
 double valorTotal = calcularValorTotalEmEstoque();
 System.out.println("Valor Total em Estoque (Eur): " + valorTotal);
 }
}

class Produto {
 private String nome;
 private int quantidade;
}

```

```

private double preco;

public Produto(String nome, int quantidade, double preco) {
 this.nome = nome;
 this.quantidade = quantidade;
 this.preco = preco;
}

// Getters e setters

public String getNome() {
 return nome;
}

public int getQuantidade() {
 return quantidade;
}

public void setQuantidade(int quantidade) {
 this.quantidade = quantidade;
}

public double getPreco() {
 return preco;
}

public void setPreco(double preco) {
 this.preco = preco;
}
}

copy

```

Neste exemplo:

- A classe `EstoqueTransactionScript` contém os métodos `adicionarProduto` para adicionar produtos ao estoque e `calcularValorTotalEmEstoque` para calcular o valor total em estoque. A lógica de negócios para essas operações é implementada diretamente nessa classe.
- A classe `Produto` representa um produto com atributos como nome, quantidade e preço.
- No método `main`, adicionamos produtos ao estoque, atualizamos a quantidade e o preço de um produto existente e, em seguida, calculamos o valor total em estoque.

Este é um exemplo simples do padrão *Transaction Script* em Java para gestão de estoque de produtos. O *Transaction Script* é adequado para operações simples e diretas, como as representadas neste exemplo. Para sistemas mais complexos, pode ser mais apropriado considerar abordagens de design mais estruturadas, como o *Domain Model* ou o padrão *Table Module*.

## 4.2 Table Module

O padrão **Table Module** é um padrão de desenho de software que se concentra na organização da lógica de negócios em torno das **tabelas da base de dados**. Em vez de dividir a lógica de negócios em classes que correspondem a objetos individuais (como no padrão **Domain Model**), o **Table Module** associa a lógica diretamente às tabelas da base de dados. Esse padrão é mais comunamente usado em sistemas que têm uma forte correspondência entre os objetos do sistema e as tabelas da base de dados.

Aqui estão os principais conceitos associados ao padrão **Table Module**:

- **Entidades ou Tabelas:** Cada tabela da base de dados ou entidade de negócios é representada por um módulo de tabela. Isso significa que há uma correspondência direta entre os módulos de tabela e as tabelas/entidades da base de dados.

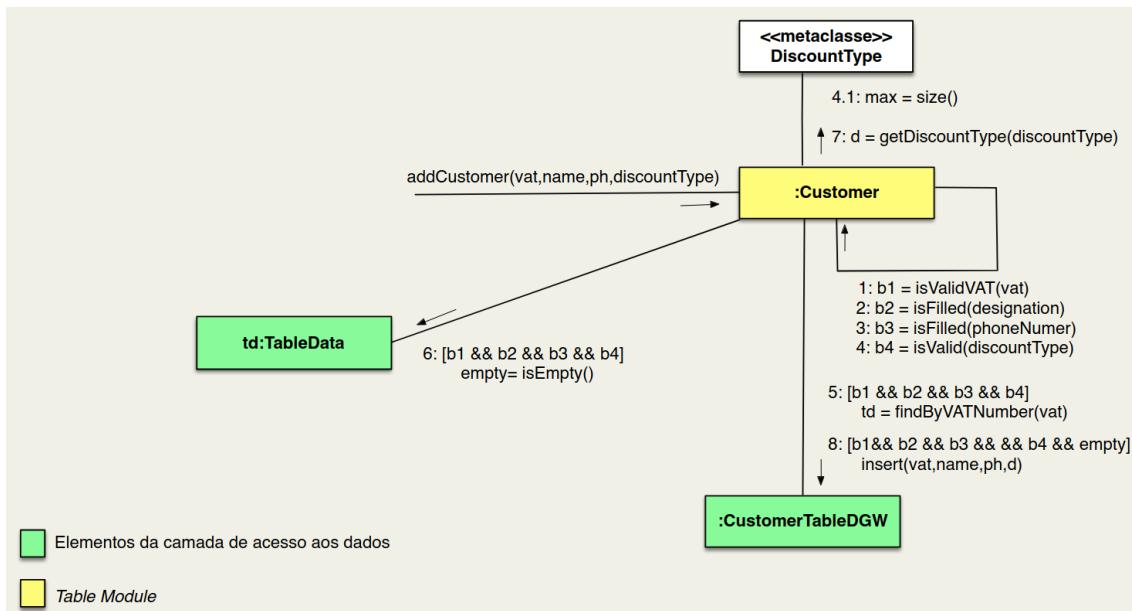
- **Lógica de Negócios Centralizada:** A lógica de negócios relacionada a uma tabela específica é centralizada no módulo de tabela correspondente. Isso inclui operações como validação, cálculos e manipulação de dados.
- **Encapsulamento:** O *Table Module* encapsula tanto os dados quanto a lógica de negócios associada à tabela. Isso significa que os métodos dentro do módulo de tabela podem aceder diretamente os dados da tabela e aplicar a lógica de negócios necessária.
- **Não há Divisão de Camadas:** Ao contrário de padrões que separam rigidamente a lógica de negócios das camadas de apresentação e acesso a dados (como o Modelo-Visão-Controlador ou o Modelo de Camadas), o *Table Module* não impõe essa separação. A lógica de negócios e o acesso aos dados podem ser combinados em um único módulo de tabela.
- **Simplicidade:** O *Table Module* pode ser uma abordagem simples e direta para sistemas que têm uma estrutura de base de dados bem definida e não requerem complexidade adicional na lógica de negócios.

Embora o *Table Module* possa ser adequado para sistemas simples ou pequenos, ele tem algumas desvantagens, especialmente em sistemas complexos:

- **Objetos:** Os módulos são diferentes dos objetos, em particular não têm identidade.
- **Mistura de Responsabilidades** A falta de separação entre lógica de negócios e acesso a dados pode resultar em módulos de tabela complexos que misturam muitas responsabilidades diferentes.
- **Dificuldade de Teste Unitário** Testar unidades individuais de lógica de negócios em um módulo de tabela pode ser desafiador, pois essas unidades podem depender diretamente da conexão com a base de dados.
- **Menos Flexibilidade** À medida que a complexidade do sistema aumenta ou se torna necessário escalar para diferentes tecnologias de acesso a dados, o *Table Module* pode se tornar menos flexível.

Em resumo, o padrão *Table Module* é uma abordagem que centraliza a lógica de negócios em torno de tabelas ou entidades da base de dados. Embora seja simples e direto, pode ser menos adequado para sistemas complexos que exigem uma separação mais clara de responsabilidades e uma maior flexibilidade de design.

**Exemplo 1:** Reutilizamos o exemplo usado na secção sobre o *Transaction Script 69*: a operação `addCustomer`. O *Table Module* (aqui `Customer`) tem a responsabilidade desta operação. Trata das verificações e interage com a camada de persistência para guardar os dados:



A implementação encontra-se no esboço seguinte:

```

public void addCustomer(int vat, String denomination, int phoneNumber,
 int discountCode) throws ApplicationException {
 // Verificar VAT:
 if (!isValidVAT(vat)) {
 throw new ApplicationException("Invalid VAT number: " + vat);
 }
 // Verificar o código de desconto:
 if (discountCode <= 0 || discountCode > DiscountType.values().length) {
 throw new ApplicationException("Invalid Discount Code :" + discountCode);
 }
 // Verificar denominação e telefone:
 if (!isFilled(denomination) || phoneNumber == 0) {
 throw new ApplicationException("Both denomination and phoneNuber must be filled.");
 }
 // Instruções para criar e inserir o registo na BD:
 try {
 table.insert(vat, designation, phoneNumber, DiscountType.values()[discountCode-1]);
 } catch (PersistenceException e) {
 throw new ApplicationException("Internal error adding a customer", e);
 }
}
copy

```

**Exemplo 2:** Suponha que tenhamos uma tabela no banco de dados chamada "Livros" com as seguintes colunas: "ID", "Título", "Autor" e "Ano de Publicação".

Aqui está uma implementação básica do *Table Module*:

```

class Livro:
 def __init__(self, livro_id, titulo, autor, ano_publicacao):
 self.id = livro_id
 self.titulo = titulo
 self.autor = autor
 self.ano_publicacao = ano_publicacao

 def salvar(self):
 # Lógica para salvar ou atualizar os dados do livro no banco de dados
 pass

 def excluir(self):
 # Lógica para excluir o livro do banco de dados
 pass

class LivroTableModule:
 @staticmethod
 def buscar_por_id(livro_id):
 # Lógica para buscar um livro pelo ID no banco de dados
 pass

 @staticmethod
 def buscar_por_titulo(titulo):
 # Lógica para buscar livros por título no banco de dados
 pass

 @staticmethod
 def listar.todos():
 # Lógica para listar todos os livros no banco de dados
 pass

```

Neste exemplo:

- A classe `Livro` representa um livro individual e contém atributos para o ID, título, autor e ano de publicação, bem como métodos para salvar e excluir o livro na base de dados.
- A classe `LivroTableModule` atua como o **módulo de tabela** para lidar com operações relacionadas aos livros no banco de dados. Ela fornece métodos estáticos para buscar livros por ID, buscar livros por título e listar todos os livros. Esses métodos encapsulam a lógica de acesso ao banco de dados e retornam objetos `Livro` para representar os resultados.

Essa é uma implementação simplificada do padrão *Table Module*. Num sistema real, usaria um ORM (*Object-Relational Mapping*) ou camada de acesso a dados para interagir com a base de dados de maneira mais eficiente. O *Table Module* é mais adequado para sistemas simples ou para operações em tabelas específicas que não requerem a complexidade de um *Domain Model* completo.

Agora um exemplo em Java: Suponha que tenhamos uma tabela na base de dados chamada "Pedidos" com as seguintes colunas: "ID do Pedido", "Cliente", "Data do Pedido" e "Valor Total".

Aqui está uma implementação simples em Java do *Table Module* para lidar com operações de pedidos:

```

import java.util.ArrayList;
import java.util.List;

// Classe que representa um pedido individual
class Pedido {
 private int id;
 private String cliente;
 private String dataPedido;
 private double valorTotal;

 // Construtor
 public Pedido(int id, String cliente, String dataPedido, double valorTotal) {
 this.id = id;
 this.cliente = cliente;
 this.dataPedido = dataPedido;
 this.valorTotal = valorTotal;
 }

 // Getters e setters

 public int getId() {
 return id;
 }

 public String getCliente() {
 return cliente;
 }

 public String getDataPedido() {
 return dataPedido;
 }

 public double getValorTotal() {
 return valorTotal;
 }
}

// Classe Table Module para operações de pedidos
class PedidoTableModule {
 // Simulando um base de dados em memória
 private static List<Pedido> baseDados = new ArrayList<>();

 // Método para adicionar um novo pedido ao base de dados
 public static void adicionarPedido(String cliente, String dataPedido, double valorTotal)
 int novoId = baseDados.size() + 1;
}

```

```

 Pedido novoPedido = new Pedido(novoId, cliente, dataPedido, valorTotal);
 baseDeDados.add(novoPedido);
}

// Método para buscar um pedido pelo ID
public static Pedido buscarPedidoPorId(int id) {
 for (Pedido pedido : baseDeDados) {
 if (pedido.getId() == id) {
 return pedido;
 }
 }
 return null; // Pedido não encontrado
}

// Método para listar todos os pedidos
public static List<Pedido> listarTodosPedidos() {
 return new ArrayList<>(baseDeDados); // Retorna uma cópia da lista
}
}

public class Main {
 public static void main(String[] args) {
 // Adicionando pedidos
 PedidoTableModule.adicionarPedido("Cliente A", "2023-09-01", 100.0);
 PedidoTableModule.adicionarPedido("Cliente B", "2023-09-02", 150.0);

 // Listando todos os pedidos
 List<Pedido> pedidos = PedidoTableModule.listarTodosPedidos();
 for (Pedido pedido : pedidos) {
 System.out.println("ID: " + pedido.getId());
 System.out.println("Cliente: " + pedido.getCliente());
 System.out.println("Data do Pedido: " + pedido.getDataPedido());
 System.out.println("Valor Total: " + pedido.getValorTotal());
 System.out.println();
 }
 }
}
}

copy

```

Neste exemplo:

- A classe `Pedido` representa um pedido individual com atributos como ID, cliente, data do pedido e valor total.
- A classe `PedidoTableModule` atua como o módulo de tabela para lidar com operações relacionadas a pedidos. Ela inclui métodos para adicionar pedidos, buscar pedidos por ID e listar todos os pedidos. Os dados são armazenados numa lista simulando uma base de dados em memória.
- No método `main`, adicionamos alguns pedidos, listamos todos os pedidos e exibimos suas informações.

Este é um exemplo simples do padrão `Table Module` em Java para operações de pedidos. Um sistema real, usaria uma base de dados real em vez de uma lista em memória e poderia implementar funcionalidades mais avançadas, como atualização e exclusão de pedidos.

## 4.3 Domain Model

O padrão **Domain Model** (Modelo de Domínio) é uma abordagem de design de software que visa criar uma representação da lógica de negócios de um sistema de software de forma a refletir fielmente

as entidades e os conceitos do domínio do problema que está sendo abordado. É frequentemente associado à arquitetura orientada a objetos e é usado para modelar e organizar a lógica de negócios de maneira mais próxima possível da realidade do domínio do problema.

Aqui estão os principais conceitos associados ao padrão *Domain Model*:

- **Entidades do Domínio:** No *Domain Model*, as entidades do domínio do problema, como Clientes, Pedidos, Produtos, etc., são mapeadas diretamente para classes no código. Cada classe representa uma entidade do domínio com seus atributos e comportamentos associados.
- **Comportamento Incluso:** Além de representar os dados do domínio (os atributos das entidades), as classes também incluem o comportamento relacionado a essas entidades. Isso significa que métodos e lógica de negócios específicos são implementados nas classes de entidade.
- **Encapsulamento:** O *Domain Model* enfatiza o encapsulamento de dados e comportamento relacionado em classes. Isso significa que as operações que afetam uma entidade específica estão encapsuladas dentro da classe daquela entidade.
- **Relacionamentos do Domínio:** Os relacionamentos entre as entidades do domínio são modelados diretamente no código, refletindo as associações e agregações presentes no domínio do problema.
- **Modelo Rico:** O *Domain Model* permite que o modelo de domínio seja rico e expressivo, capturando com precisão os conceitos do domínio e suas interações.
- **Separação de Responsabilidades:** Embora a lógica de negócios esteja intimamente relacionada com as classes de entidade, a separação de responsabilidades é alcançada ao manter a lógica de apresentação e o acesso a dados em camadas distintas (por exemplo, usando o padrão MVC ou arquitetura de camadas).
- **Testabilidade:** Como a lógica de negócios é encapsulada em classes de entidade, é mais fácil criar testes unitários para validar o comportamento das entidades individualmente.

O *Domain Model* é frequentemente usado em aplicações empresariais ou sistemas complexos em que a representação precisa e flexível do domínio do problema é fundamental. No entanto, ele também pode ser mais complexo de implementar do que outras abordagens, especialmente em sistemas pequenos ou simples.

O *Domain Model* é aplicado no contexto de uma abordagem orientada aos objetos que consiste em várias etapas (esses conceitos foram ensinados na disciplina de Análise e Desenvolvimento de Sistemas (ADS)) :

1. **Análise de Requisitos:** Durante essa fase, são identificados os requisitos do sistema e as necessidades dos utilizadores. São elaborados **diagramas de casos de uso** para representar as interações entre os utilizadores e o sistema.
2. **Modelo de Domínio:** um **Modelo de Domínio** é construído representando os principais conceitos e objetos do problema, incluindo atributos das classes de domínio e relacionamentos entre classes.
3. **Estrutura do sistema:** Projeção da estrutura do sistema incluindo **Modelo de Classes** (ou **Diagramas de Classes**), **Diagramas de Interação**, **Diagramas de Sequência** e de colaboração.
4. **Implementação.**
5. **Testes:**
6. etc...

É importante notar que o *Domain Model* não é apropriado para todos os tipos de sistemas. Em sistemas menores ou com requisitos mais simples, abordagens mais leves, como o *Transaction Script* ou o *Table Module*, podem ser mais adequadas. A escolha da abordagem de design depende das necessidades específicas do projeto.

Segue um exemplo simples do uso do padrão *Domain Model* em Java para uma aplicação de gestão de pedidos. No *Domain Model*, a lógica de negócios é modelada como classes de domínio que representam objetos do mundo real e suas interações. Vamos criar um exemplo de domínio de *Pedido*.

```
import java.util.ArrayList;
import java.util.List;

// Classe de domínio para representar um Pedido
class Pedido {
 private int numeroPedido;
 private String cliente;
 private List<ItemPedido> itens;

 public Pedido(int numeroPedido, String cliente) {
 this.numeroPedido = numeroPedido;
 this.cliente = cliente;
 this.itens = new ArrayList<>();
 }

 // Método para adicionar um item ao pedido
 public void adicionarItem(Produto produto, int quantidade) {
 ItemPedido item = new ItemPedido(produto, quantidade);
 itens.add(item);
 }

 // Método para calcular o valor total do pedido
 public double calcularValorTotal() {
 double valorTotal = 0;
 for (ItemPedido item : itens) {
 valorTotal += item.calcularSubtotal();
 }
 return valorTotal;
 }

 // Getters
 public int getNumeroPedido() {
 return numeroPedido;
 }

 public String getCliente() {
 return cliente;
 }

 public List<ItemPedido> getItens() {
 return itens;
 }
}

// Classe de domínio para representar um Produto
class Produto {
 private int codigo;
 private String nome;
 private double preco;

 public Produto(int codigo, String nome, double preco) {
 this.codigo = codigo;
 this.nome = nome;
 this.preco = preco;
 }

 // Getters
 public int getCodigo() {
 return codigo;
 }
```

```
}

public String getNome() {
 return nome;
}

public double getPreco() {
 return preco;
}

}

// Classe de domínio para representar um Item de Pedido
class ItemPedido {
 private Produto produto;
 private int quantidade;

 public ItemPedido(Produto produto, int quantidade) {
 this.produto = produto;
 this.quantidade = quantidade;
 }

 // Método para calcular o subtotal do item
 public double calcularSubtotal() {
 return produto.getPreco() * quantidade;
 }

 // Getters
 public Produto getProduto() {
 return produto;
 }

 public int getQuantidade() {
 return quantidade;
 }
}

public class Main {
 public static void main(String[] args) {
 // Criando produtos
 Produto produto1 = new Produto(1, "Produto A", 25.0);
 Produto produto2 = new Produto(2, "Produto B", 15.0);

 // Criando um pedido
 Pedido pedido = new Pedido(101, "Cliente X");

 // Adicionando itens ao pedido
 pedido.adicionarItem(produto1, 3);
 pedido.adicionarItem(produto2, 5);

 // Calculando o valor total do pedido
 double valorTotal = pedido.calcularValorTotal();

 // Exibindo informações do pedido
 System.out.println("Número do Pedido: " + pedido.getNumeroPedido());
 System.out.println("Cliente: " + pedido.getCliente());
 System.out.println("Itens do Pedido:");
 for (ItemPedido item : pedido.getItens()) {
 System.out.println("- Produto: " + item.getProduto().getNome());
 System.out.println(" Quantidade: " + item.getQuantidade());
 }
 }
}
```

```

 System.out.println(" Subtotal (Eur) : " + item.calcularSubtotal());
 }
 System.out.println("Valor Total do Pedido (Eur) : " + valorTotal);
}
}

copy

```

Neste exemplo:

- A classe `Pedido` representa um pedido com atributos como número do pedido, cliente e uma lista de `ItemPedido`. A lógica de negócios para adicionar itens e calcular o valor total do pedido é encapsulada nesta classe.
- A classe `Produto` representa um produto com atributos como código, nome e preço.
- A classe `ItemPedido` representa um item do pedido, incluindo o produto e a quantidade. A lógica de negócios para calcular o subtotal do item está nesta classe.

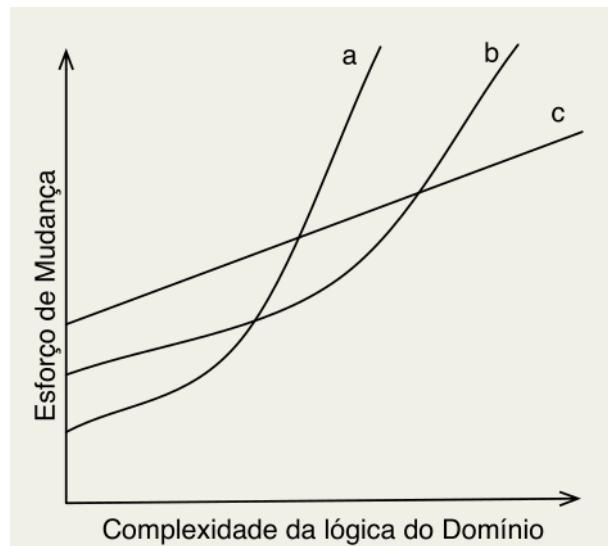
No método `main`, criamos produtos, um pedido e adicionamos itens ao pedido. Em seguida, calculamos o valor total do pedido e exibimos as informações do pedido.

Este é um exemplo simples do padrão *Domain Model* em Java para gerenciamento de pedidos. O *Domain Model* é adequado para sistemas onde a representação precisa de objetos do mundo real e suas interações é fundamental para a lógica de negócios.

## 4.4 Como escolher ?

A escolha entre os padrões de software ***Transaction Script***, ***Table Module*** e ***Domain Model*** depende da complexidade do seu sistema, dos requisitos do projeto e das preferências de desenho. Cada padrão tem suas próprias vantagens e é mais adequado para situações específicas. Aqui estão algumas diretrizes gerais para ajudar na escolha:

- ***Transaction Script***
  - Use o padrão *Transaction Script* quando sua aplicação tiver lógica de negócios simples e direta.
  - É adequado para sistemas onde a maioria das operações de negócios pode ser implementada como procedimentos simples.
  - É uma escolha sólida para aplicativos pequenos a médios com requisitos de lógica de negócios não muito complexos.
  - É mais adequado quando a separação rigorosa de responsabilidades não é uma prioridade.
- ***Table Module***
  - Considere o padrão *Table Module* quando sua aplicação envolver um número limitado de tabelas de banco de dados que representam entidades principais.
  - É útil quando as regras de negócios estão fortemente vinculadas aos dados em tabelas específicas e você deseja encapsular essas regras em classes dedicadas (*Table Modules*).
  - Funciona bem em sistemas onde o foco é principalmente em operações em torno de registros individuais em tabelas.
- ***Domain Model***
  - Escolha o padrão *Domain Model* quando sua aplicação tiver uma lógica de negócios complexa e uma variedade de objetos de domínio interconectados.
  - É mais apropriado para sistemas grandes e complexos, onde a ênfase está na modelagem rica e em objetos de domínio com comportamento sofisticado.
  - É adequado quando você deseja uma separação clara entre a lógica de negócios e a camada de persistência, permitindo flexibilidade na escolha de tecnologias de acesso a dados.
  - É uma escolha sólida quando os requisitos incluem regras de negócios complexas, validações extensas e cenários de uso diversificados.



## 4.5 Exercícios

Salesys (Domain Model) p.82 • Salesys (Transaction Script) p.85 • Salesys (Table Module) p.87

O **Salesys** é um sistema empresarial de vendas e administração de inventário. Propósito da aplicação é fazer a gestão da carteira de clientes e de vendas de uma empresa. É uma aplicação fictícia, muito simples, que é usada na disciplina ao longo do semestre para ilustrar os conceitos e as técnicas que vão sendo apresentadas. Para a primeira iteração do sistema, como é habitual, vamos considerar um fragmento simplificado do problema, cnetrado nas vendas que a empresa realiza aos seus clientes.

**Descrição do sistema.** A empresa possui uma carteira de clientes com quem comercializa produtos. Nesta primeira fase, cada cliente é caracterizado por um número de pessoa coletiva, uma denominação (nome do cliente) e um contacto telefónico. Cada produto comercializado pela empresa tem um código que o identifica univocamente, uma descrição e um preço. Um produto tem, em cada momento, uma quantidade de artigos disponíveis em stock.

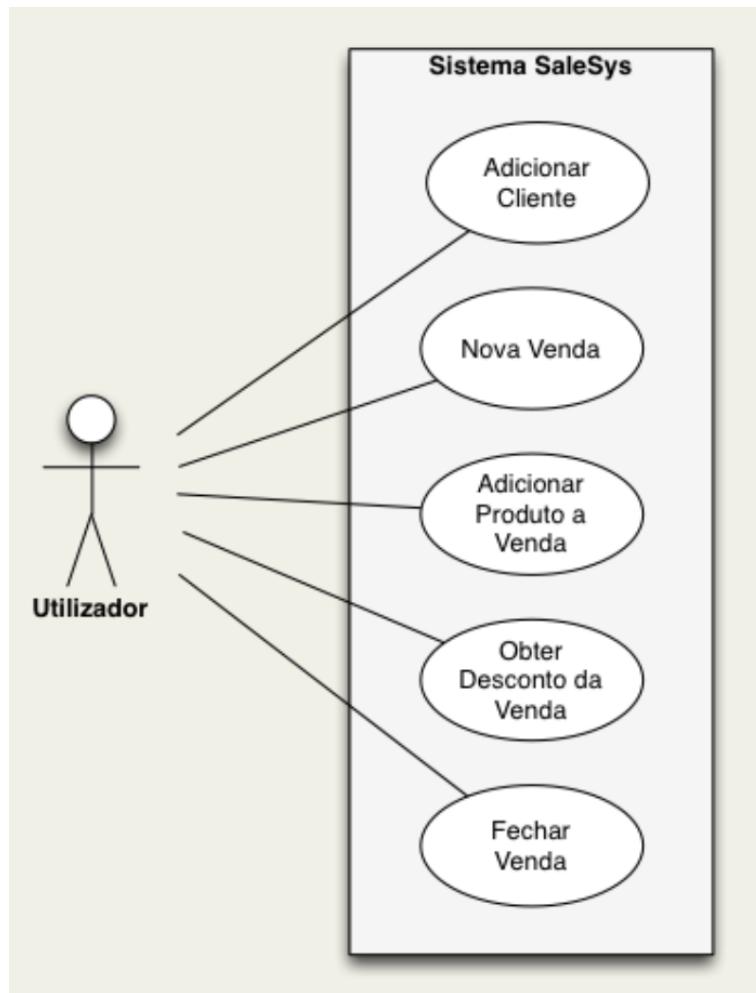
As vendas registam transações efetuadas com os clientes. Cada venda grava a data em que a operação foi efetuada, o cliente, a lista de produtos (e respetivas quantidades) transacionados e o valor total. A empresa pratica nas vendas que efetua, variados tipos de descontos. Assim, cada venda regista também o valor do desconto aplicado.

Atualmente a empresa pratica dois tipos de desconto nas suas vendas: o primeiro resulta da aplicação de uma percentagem sobre o valor global, caso o cliente atinja um determinado valor total na compra, o segundo corresponde a uma percentagem sobre o total dos produtos que estiverem marcados como elegíveis para desconto. O tipo de desconto a que uma venda está sujeita é determinada pelo cliente a quem a venda é efetuada. Um cliente pode não ter direito a nenhum tipo de desconto.

A empresa prevê num futuro próximo enriquecer as formas de desconto com o objetivo de acompanhar a concorrência no setor. Para tal, além de novos modelos de desconto, prevê também poder vir a combinar vários descontos numa mesma venda, mas para já ainda não é um requisito da aplicação.

Os casos de uso definidos para a primeira versão do sistema são:

- Adicionar cliente
- Criar venda
- Acrescentar produtos a uma venda em aberto
- Obter valor de desconto da venda
- Concluir venda



Salesys: Casos de Uso.

- Aplicações empresariais** Enumere um conjunto de características das aplicações emresariais que o Salesys irá possuir, justificando.
- Arquitectura em camadas** Explique o que implica em concreto a decisão de que a aplicação deve ter uma arquitectura em camadas. Enumere, justificando, as camadas que devem ser concretizadas.
- Modelo de dados** Suponha que foi decidido usar uma base de dados relacional para persistir os dados da aplicação. Apresente um esboço do modelo de dados correspondente.

Salesys (Domain Model) p.[82](#) • Salesys (Transaction Script) p.[85](#) • Salesys (Table Module) p.[87](#)

#### 4.5.1 Salesys (Domain Model)

Suponha que se pretende desenhar a camada de negócio da aplicação Salesys de acordo com o padrão Domain Model, ou seja pretende-se organizar o código desta camada com classes que são inspiradas pelos conceitos do domínio, nas suas características e associações.

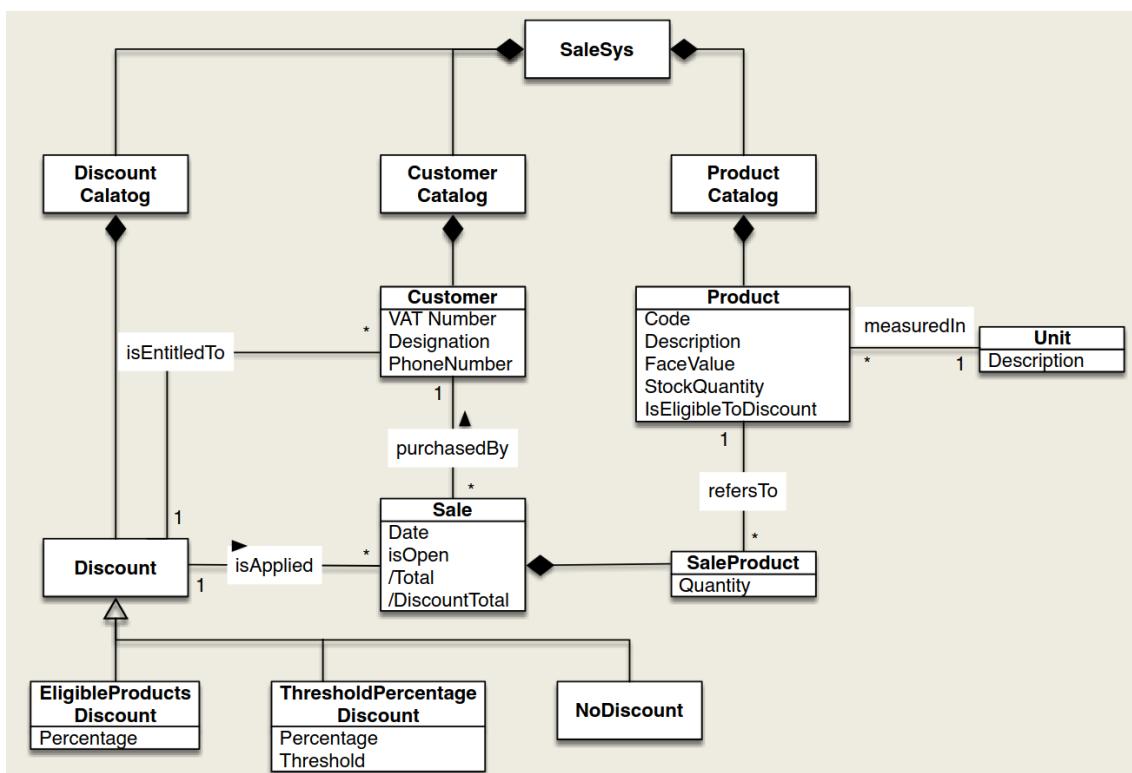
- Elabore um modelo de domínio a fim de identificar os principais conceitos do sistema.
- Aplicando os padrões GRASP, elabore um diagrama de interação (de sequência ou de comunicação) que defina o comportamento da operação addCustomer que trata da adição de um novo cliente.
- Repita que fez anteriormente para as operações

- `newSale`
- `addProductToSale`
- `getSaleDiscount`
- `closeSale`

4. Construa um diagrama de classes que reflita as decisões tomadas no desenho das operações das duas alíneas anteriores.

## Modelo de domínio

Elaboramos um Modelo do Domínio, um mapa com os principais conceitos do domínio e relações entre eles.



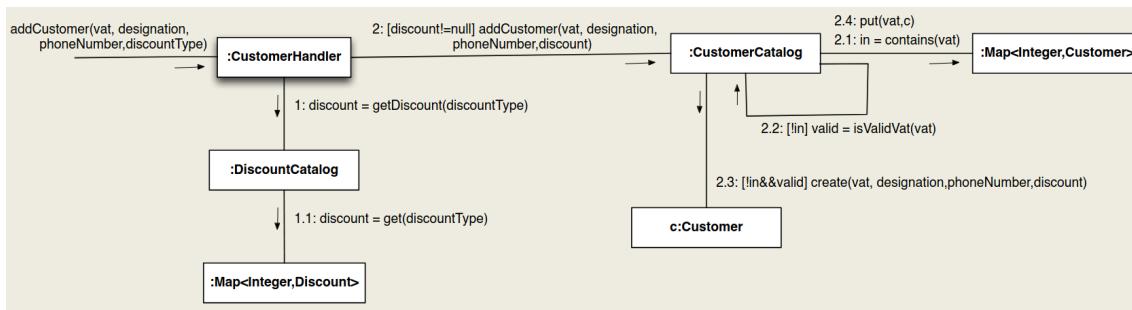
Salesys: Modelo de domínio (esboço).

A seguir, inspirados pelo modelo de domínio e recorrendo aos padrões GRASP desenhamos uma solução orientada a objetos para a camada de negócio do sistema:

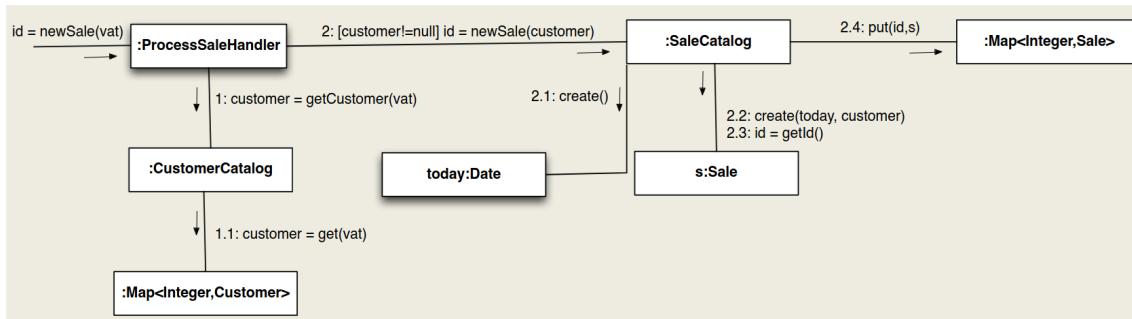
1. contratos das operações
2. diagramas de interação para cada operação
3. modelo de classes

Vamos ignorar por agora a necessidade de persistir os dados e guardar os dados apenas em memória

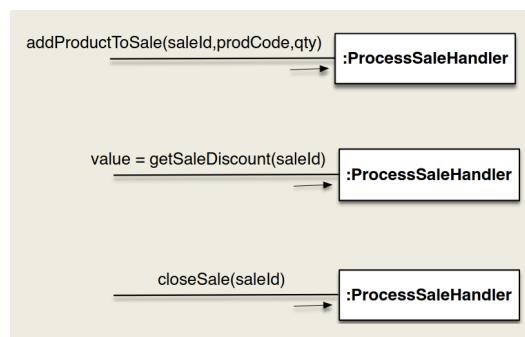
## Diagrama de interação (addCustomer)



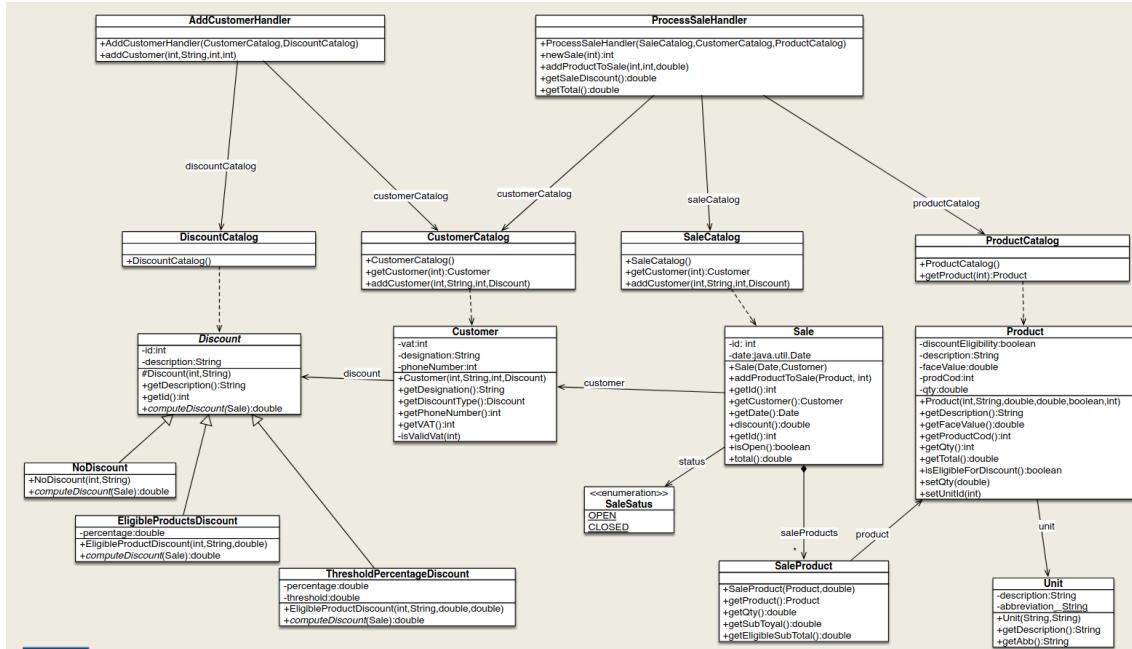
### Diagrama de interação (newSale)



### Diagrama de interação (outros casos de uso)

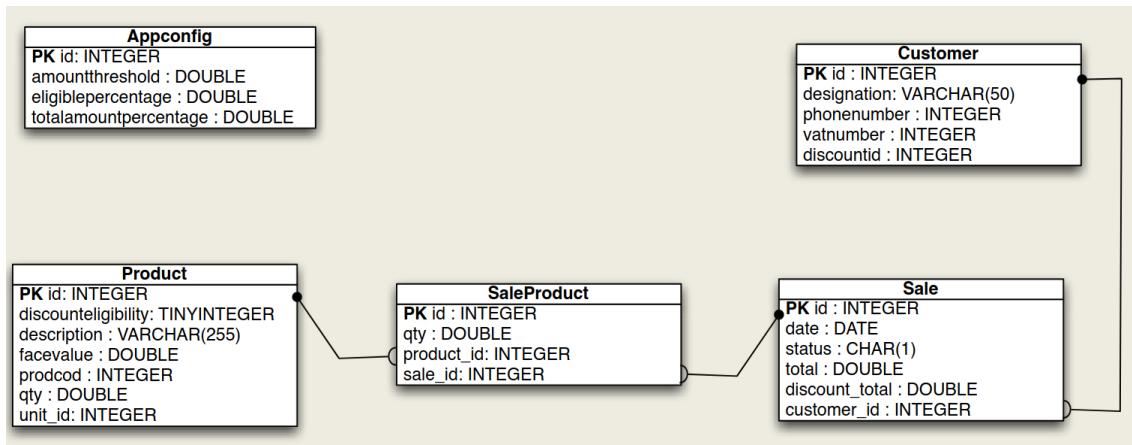


## Modelo de classes



## Modelo de dados

Suponha que foi decidido usar uma base de dados relacional para persistir todos os dados da aplicação. Apresente um modelo de dados apropriado.



### 4.5.2 Salesys (Transaction Script)

Suponha agora que se pretende desenhar a camada de negócio da aplicação Salesys recorrendo ao padrão *Transaction Script*. Esta solução deve basear-se numa camada de acesso aos dados desenhada de acordo com o padrão ??.

1. Elabore um diagrama de interação que defina o comportamento da operação `addCustomer` de acordo com o padrão *Transcation Script*.
2. Repita o que fez anteriormente para as operações `newSale`, `addProductToSale`, `getSaleDiscount` e `closeSale`.

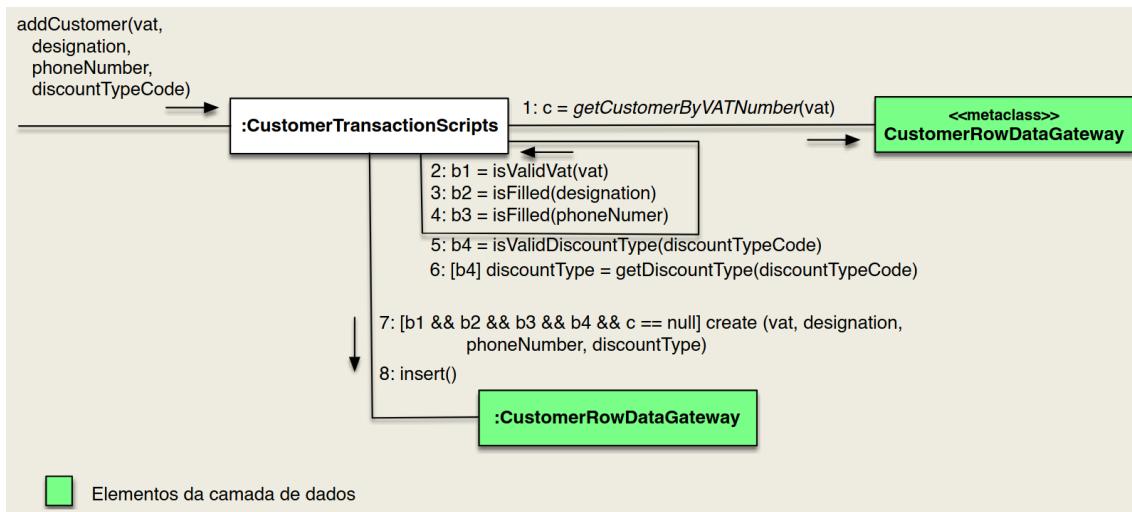
O script `CustomerTransactionScript` será responsável por executar a tarefa:

#### 1. Validação:

- (a) Recorrendo aos **serviços prestados pela camada de persistência**, verificar se já há um cliente com o VAT dado

- (b) Verificar se código do tipo de desconto é válido,
- (c) Verificar se o VAT é válido,
- (d) Verificar se o nome e o telefone estão preenchidos.

**2. Processamento:** Recorrendo aos serviços prestados pela **camada de persistência**, inserir uma linha na tabela **Customer**.



### Implementação:

```

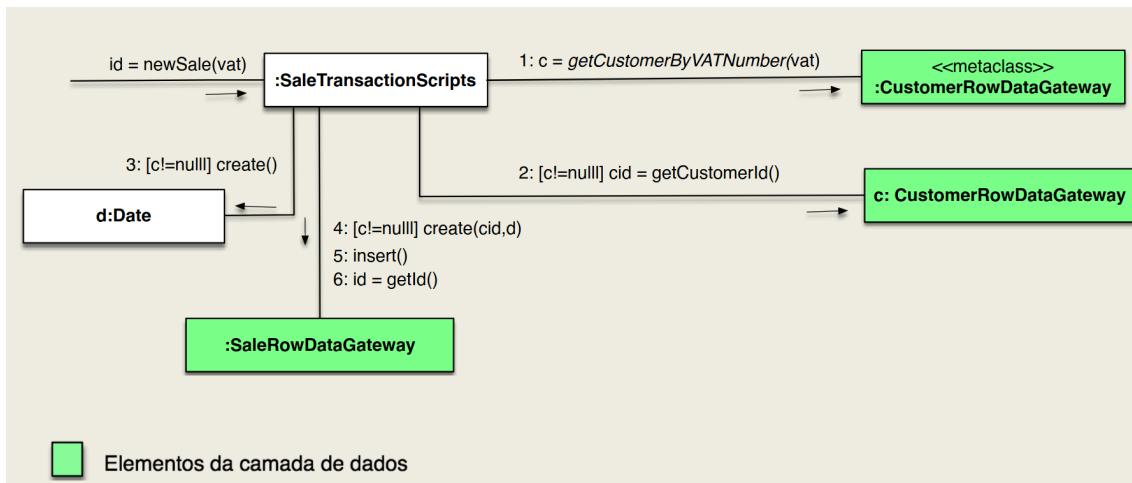
public void addCustomer(int vat, String denomination, int phoneNumber,
 int discountCode) throws ApplicationException {
 // Verificar VAT:
 if (!isValidVAT(vat)) {
 throw new ApplicationException("Invalid VAT number: " + vat);
 }
 // Verificar o código de desconto:
 if (discountCode <= 0 || discountCode > DiscountType.values().length) {
 throw new ApplicationException("Invalid Discount Code :" + discountCode);
 }
 // Verificar denominação e telefone:
 if (!isFilled(denomination) || phoneNumber == 0) {
 throw new ApplicationException("Both denomination and phoneNuber must be filled.");
 }
 // Instruções para criar e inserir o registo na BD:
 try {
 CustomerRowDataGateway newCustomer = new CustomerRowDataGateway(vat, denomination,
 phoneNumber, DiscountType.values()[discountCode - 1]);
 newCustomer.insert();
 } catch (PersistenceException e) {
 throw new ApplicationException("Error inserting the customer into the database.", e);
 }
}

```

copy

Agora para a operação **newSale**, o script é : **SaleTransactionScript**. A operação consiste em:

- 1. Validação:** Recorrendo aos serviços prestados pela **camada de persistência**, verificar se há um cliente com aquele VAT
- 2. Processamento:** 1) Recorrendo aos serviços prestados pela **camada de dados**, registrar uma nova venda com da data corrente na base de dados; notar que isto implica ter de obter a chave do registo do cliente com o VAT dado. 2) Devolver o número de registo da venda (chave primária da tabela Sale).

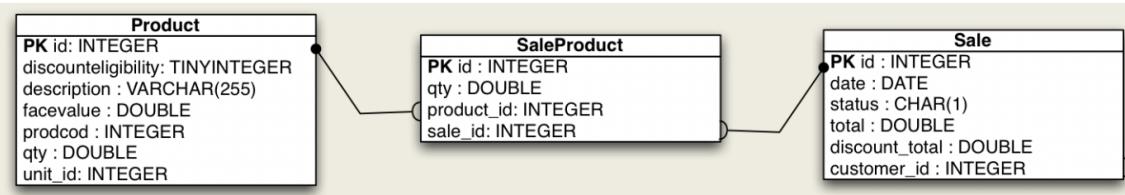


A operação `addProductToSale` consiste em:

### 1. Validação

- verificação do stock do produto
- existe venda com identificador `saleId` e não está fechada,
- existe produto com código `prodCode`,

As tabelas da base de dados envolvidas são:



Segue-se um esboço do código :

```

void addProductToSale(int saleID, int productCode, double qty)
 throws ApplicationException {
 // Checks if sale exists and is still open
 SaleRowDataGateway sale = getSale(saleId);
 if (sale.getStatus() == SaleStatus.CLOSED) {
 throw new ApplicationException("Products cannot be added to closed sales.");
 }
 // get the product ID:
 ProductRowDataGateway product = getProduct(productCode);
 if (product.getQty() >= qty) {
 product.setQty(product.getQty() - qty);
 product.updateStock();
 SaleProductRowDataGateway saleProduct = new SaleProductRowDataGateway(sale.getId(),
 product.getProductID(), qty);
 saleProduct.insert();
 }
}
copy

```

### 4.5.3 Salesys (Table Module)

Vamos agora estudar a implementação da aplicação em torno do padrão *Table Module*. Esta solução deve basear-se numa amada de acesso aos dados desenhada de acordo com o padrão *Table Data Gateway*. Vai-se considerar o caso em que esta camada trata da persistência numa base de dados relacional utilizando o JDBC.

1. Elabore um diagrama de interação que defina o comportamento da operação `addCustomer` de acordo com o padrão Table Module

2. Repita o que fez para as operações `newSale`, `addProductToSale`, `getSaleDiscount` e `closeSale`.

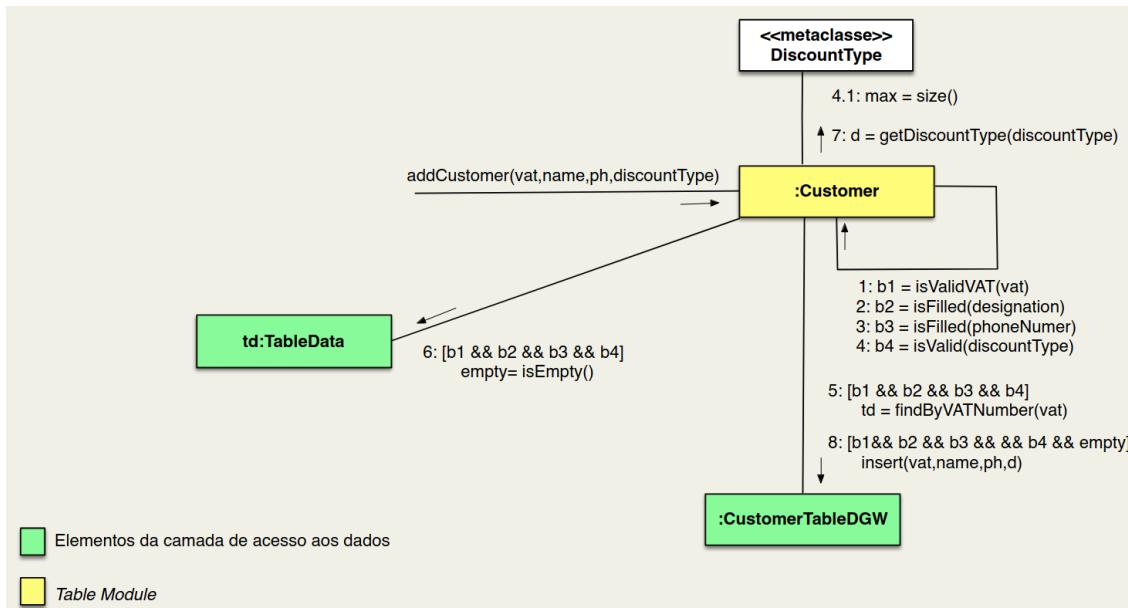
Para efetuar a acção `addCustomer` é necessário:

### 1. Validação

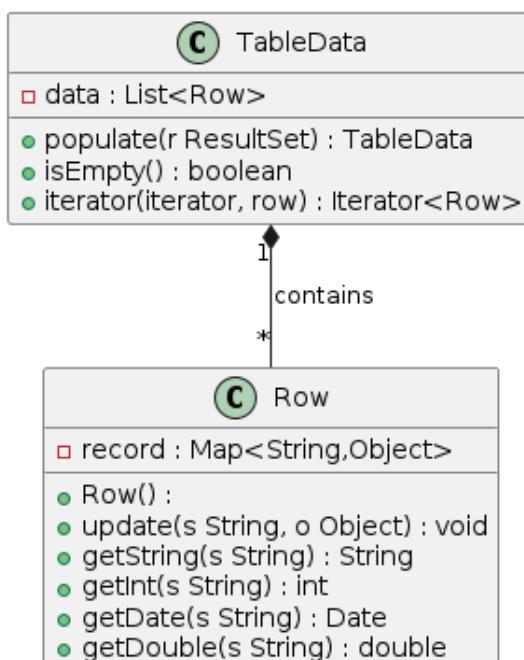
- Verificar se já há um cliente com o VAT dado (*fail fast*),
- Verificar se código do tipo de desconto é válido,
- Verificar se o VAT é válido,
- Verificar se o nome e o telefone estão preenchidos.

2. **Processamento** Recorrendo aos serviços prestados pela **camada de dados**, guardar o novo cliente de forma persistente.

O diagrama de interação é:



**TableData** é uma classe que representa tabelas genéricas:



Operação **newSale**: a operação consiste em:

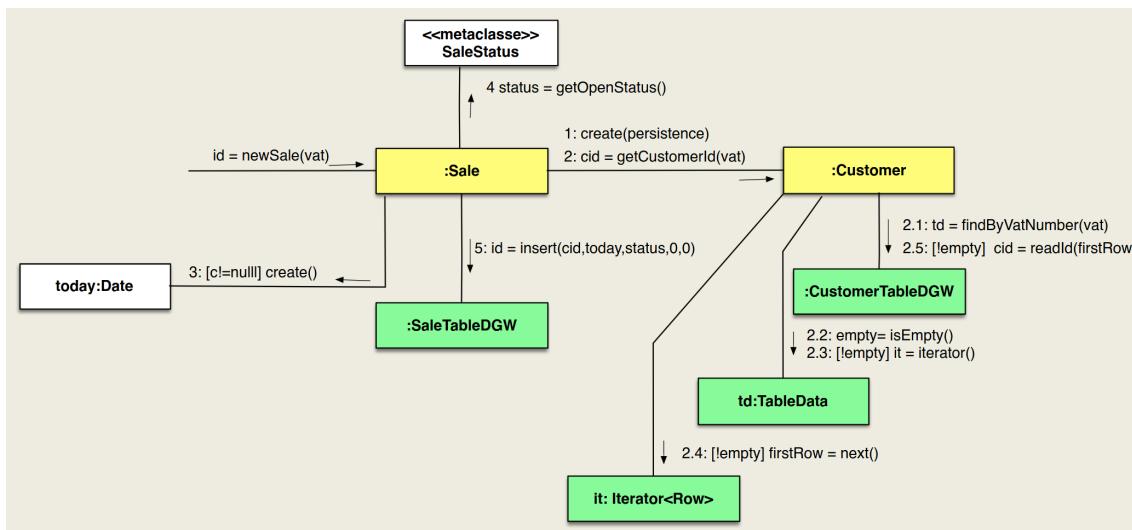
- Validação** Recorrendo aos serviços prestados pela camada de dados, verificar se há um cliente com aquele vat

## 2. Processamento

- Recorrendo aos serviços prestados pela camada de dados, registar uma nova venda com da data corrente na base de dados; notar que isto implica ter de obter a chave do registo do cliente com o VAT dado
- Devolver o número de registo da venda

Responsabilidades mais importantes dependem de dados guardados na tabela `Sale` pelo que são atribuídas ao **SaleTableModule**.

- Verificar se há um cliente com aquele VAT e obter a respetiva chave do registo: dependem de dados guardados na tabela `Customer` pelo que são atribuídas ao **CustomerTableModule**
- Registrar uma nova venda com a data corrente na base de dados e devolver o número de registo da venda





# Capítulo 5

## Padrões para a camada de persistência

Como no caso da camada de negócio, existem vários padrões que ajudam a implementação da camada de persistência.

- **Row Data Gateway:** O padrão **Row Data Gateway** é um padrão de design que encapsula o acesso a uma única linha de um banco de dados relacional, representando-a como um objeto em código.
- **Table Data Gateway:** O **Table Data Gateway** é um padrão que representa uma classe que atua como uma “porta de entrada” para uma tabela específica em a base de dados. Fornece métodos para interagir diretamente com os dados da tabela, como inserção, atualização, remoção e consulta.
- **Active Record:** O padrão **Active Record** associa diretamente objetos a registros de uma base de dados relacional. Cada classe de objeto tem métodos para realizar operações de CRUD (Create, Read, Update, Delete) na base de dados. Este padrão é frequentemente usado em ORMs (*Object-Relational Mapping*).
- **Data Mapper:** O **Data Mapper** separa completamente o domínio da base de dados. Mapeia objetos de domínio para registros da base de dados usando uma classe intermediária chamada “mapper”. Permite um maior desacoplamento entre o domínio e a camada de persistência.
- **Object-Relational Mapping (ORM):** Um ORM é uma estrutura que mapeia objetos de domínio para tabelas numa base de dados relacional. Alguns dos ORMs populares incluem Hibernate (Java), Entity Framework (.NET), Django ORM (Python) e Sequelize (Node.js).

A escolha do padrão de persistência depende da complexidade do sistema, dos requisitos de desempenho, da escalabilidade e da preferência da tecnologia. Em muitos casos, um ORM é uma escolha popular, pois simplifica a interação com o banco de dados e oferece um mapeamento objeto-relacional. No entanto, em sistemas mais complexos ou com requisitos específicos, outros padrões, como o *Data Mapper* ou *Repository*, podem ser mais adequados.

Padrão Gateway p.91 • Row data gateway p.92 • Table data gateway p.94  
• Active Record p.97 • Data Mapper p.99 • JDBC p.101 • Mapeamento Orientado Objeto p.107 • Java Persistence API p.112 • Exercícios p.126  
• Guião JDBC & Transaction Script p.128 • Guião JPA p.143

### 5.1 Padrão Gateway

Nenhum sistema funciona isolado do exterior. Para aceder a recursos externos é usual usar uma API (application programming interface) para comunicação com os serviços desses recursos.

As APIs são, normalmente, complexas de lidar. Se várias classes têm de lidar com a API, esta complexidade espalha-se pelo sistema, exigindo que todos os programadores a compreendam. Se for necessário mudar de API, é preciso refletir a alteração em todas estas classes.

O padrão Gateway diz-nos para arrumar a interacção com a API numa classe especial (a classe gateway) que se preocupa em estabelecer as comunicações necessárias. As restantes classes terão apenas de interagir com a classe gateway através de métodos apropriados.

Fowler aconselha, nos casos de API mais complexas, a definição de duas gateway, uma classe back-end e outra front-end.

- A classe back-end funciona como uma gateway de baixo nível onde se espelham os serviços da API. Se a API tem um serviço X, esta classe terá o método X(). Esta classe não é usada pela camada lógica.
- A classe front-end usa a back-end para definir os métodos adequados à camada de negócio.

Uma classe gateway tem a vantagem de se **poder substituir mais facilmente uma API por outra**. A gateway **impede que a necessidade de mudanças se espalhe para o resto do sistema**. Facilita a fase de testes, onde se pode substituir um serviço remoto (cujo acesso é lento) por um serviço local que finge as funcionalidades do primeiro, para que se possa testar mais rapidamente a camada de negócio.

Estes serviços falsos correspondem, no livro do Fowler, ao padrão **Service Stub**.

A estratégia de convergir funcionalidades numa classe corresponde ao padrão **Facade**. Qual é a diferença entre o padrão Gateway e o padrão Facade ?

- **Gateway**: o objetivo é abstrair uma API externa para consumo interno (simplificar uma API complexa)
- **Facade**: produzir uma API simplificada para consumo externo (facilitar a vida do lado do cliente) uma Facade pode combinar features de vários modulos internos para fornecer uma API simplificada.

## 5.2 Row data gateway

O padrão **Row Data Gateway** é um padrão de desenho de software que fornece uma interface simples para aceder a um único registo de uma base de dados relacional, tratando-o como um objeto. Cada registo na base de dados é representado por uma classe específica que atua como uma “porta de entrada” (gateway) para aquele registo. O padrão **Row Data Gateway** é frequentemente usado em sistemas onde a interação direta com registros de banco de dados é necessária.

Aqui estão os principais conceitos associados ao padrão Row Data Gateway:

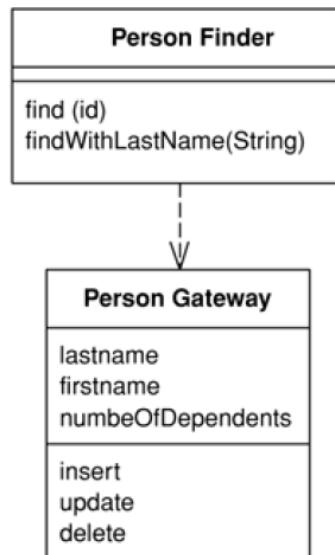
- **Classe Gateway**: Cada tabela ou entidade na base de dados tem uma classe específica que atua como o Row Data Gateway. Essa classe encapsula todas as operações relacionadas a esse registo, como consulta, atualização, inserção e exclusão.
- **Representação dos Dados**: A classe Gateway mapeia os campos do registo da base de dados para atributos da classe. Por exemplo, um campo `Nome` na base de dados pode ser mapeado para um atributo `nome` na classe Gateway.
- **Métodos de Acesso**: A classe Gateway geralmente fornece métodos para recuperar, atualizar e excluir o registo associado no banco de dados. Esses métodos são usados para realizar operações CRUD no registo.
- **Encapsulamento**: O Row Data Gateway encapsula detalhes específicos da base de dados, como consultas SQL ou instruções de atualização. Os clientes que usam o Gateway não precisam se preocupar com esses detalhes.
- **Um Registo por Objeto**: Cada instância da classe Gateway representa um único registo na base de dados. Isso significa que, se quiser trabalhar com vários regtos, precisará criar várias instâncias da classe.
- **Eficiência**: O padrão Row Data Gateway pode ser eficiente para operações que envolvem apenas um registo da base de dados, pois evita a carga desnecessária de dados em memória.

Neste padrão cada objeto da classe gateway corresponde a **uma linha** (i.e., um registo) da query actual da respetiva tabela.

- Existem **atributos** para representar as **colunas** da tabela da BD.
- Os construtores recebem os valores dos atributos necessários para a construção
- tem **getters** e **setters** para os diferentes atributos
- A classe possui métodos de instância `insert()`, `delete()` e `update()`/`updateXYZ()` para inserir um registo, apagar ou alterar o registo representado pela instância.
- A classe *gateway* deverá conter apenas o código para acesso à BD e não deve conter qualquer código relativo ao domínio

Para efetuar a procura existem duas possibilidades quanto a localização do métodos responsáveis

- Definir **métodos de classe** na classe **RowDataGateway** o que impede o uso de polimorfismo, que é útil se quisermos ter diferentes implementações dos métodos de procura para diferentes fontes de dados
- Definir **métodos de instância** numa classe *Finder* separada. Os objetos desta classe tem a responsabilidade de fazer as pesquisas e construir os respetivos *gateways* (i.e., objetos do tipo **RowDataGateway**)



## Exemplos de Utilização

- Inserir um novo cliente
  1. Criar o objeto `CustomerRowDataGateway` correspondente ao cliente
  2. Preencher os atributos do objeto, via construtor ou através de chamadas aos **setters**
  3. chamar o método `insert()`
- Alterar ou apagar cliente
  1. Tendo o objeto `CustomerRowDataGateway` correspondente ao registo do cliente a alterar ou modificar
  2. Chamar **setters** para alterar o objeto (caso se pretenda alterar)
  3. Chamar o método `update()` ou `delete()` para se alterar ou apagar o correspondente registo na tabela da BD

Aqui está um exemplo simples em Java de um *Row Data Gateway* para uma tabela de “Clientes” numa base de dados:

```

public class CustomerRowDataGateway {
 private int id;
 private String name;
 private String email;

 // Construtor
 public CustomerRowDataGateway(int id, String name, String email) {
 this.id = id;
 this.name = name;
 this.email = email;
 }

 // Métodos para aceder à base de dados
 public static CustomerRowDataGateway findById(int id) {
 // Lógica para consultar o banco de dados e criar um objeto CustomerRowDataGateway
 return new CustomerRowDataGateway(id, "Cliente Teste", "cliente@teste.com");
 }

 public void insert() {
 // Lógica para atualizar o registo no banco de dados com os dados deste objeto
 }

 public void delete() {
 // Lógica para remover o registo do banco de dados
 }

 // Getters e setters
 // ...
}

```

**copy**

Neste exemplo:

- A classe `CustomerRowDataGateway` representa um registo na tabela `Customer` da base de dados. Possui atributos que mapeiam os campos do registo, como `id`, `name` e `email`.
- Os métodos estáticos como `findById` são usados para consultar a base de dados e criar objetos `CustomerRowDataGateway` com base nos dados encontrados.
- Os métodos de instância `inserir` e `delete` são usados para atualizar ou remover o registo da base de dados.

O padrão *Row Data Gateway* é uma abordagem simples e direta para acessar registos de uma base de dados como objetos. No entanto, pode ser mais adequado para sistemas com requisitos simples de persistência de dados, e pode não ser a escolha ideal para sistemas mais complexos que requerem mapeamento objeto-relacional avançado e gestão de relacionamentos entre entidades.

## 5.3 Table data gateway

O padrão **Table Data Gateway** é um padrão de design de software que fornece uma interface para aceder e manipular dados numa tabela específica de uma base de dados relacional. Em vez de representar cada registo como um objeto separado, como no padrão **Row Data Gateway**, o **Table Data Gateway** trata toda a tabela como uma entidade única, fornecendo métodos para realizar operações CRUD (Create, Read, Update, Delete) na tabela como um todo.

Aqui estão os principais conceitos associados ao padrão *Table Data Gateway*:

- **Classe Gateway:** Cada tabela da base de dados tem uma classe específica que atua como o *Table Data Gateway*. Essa classe é responsável por encapsular todas as operações relacionadas a essa tabela, como consultas, inserções, atualizações e remoções.

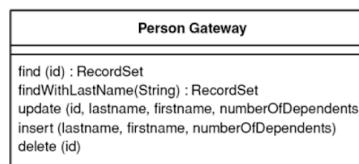
- **Operações de CRUD:** O *Table Data Gateway* fornece métodos para realizar operações de CRUD na tabela. Por exemplo, métodos para buscar registros, inserir novos registros, atualizar registros existentes e remover registros da tabela.
- **Representação de Dados:** A classe Gateway mapeia os campos da tabela da base de dados para atributos ou propriedades da classe.
- **Encapsulamento:** O *Table Data Gateway* encapsula os detalhes específicos da base de dados, como consultas SQL ou instruções de atualização. Os clientes que usam o Gateway não precisam se preocupar com esses detalhes.
- **Eficiência:** O *Table Data Gateway* pode ser eficiente para operações que envolvem toda a tabela, como consultas em massa ou atualizações em lote.
- **Um Gateway por Tabela:** Em sistemas que usam o padrão *Table Data Gateway*, geralmente há uma classe de Gateway dedicada para cada tabela da base de dados.
- **Desacoplamento:** O *Table Data Gateway* promove um certo grau de desacoplamento entre a lógica de negócios do aplicativo e os detalhes da persistência de dados.

Neste padrão uma instância da classe *gateway* gere a respetiva tabela. A classe inclui nos seus métodos públicos, **todas as operações SQL** necessárias à gestão da tabela. O padrão *Table Data Gateway* permite que os clientes acedam e manipulem os registos da tabela usando as primitivas da linguagem de programação, encapsulando todas as especificidades da API que permite aceder à fonte de dados. Esta abordagem para a camada de dados é especialmente adequada para o *Table Module* na camada de negócio, mas também é uma alternativa para o *Transaction Script*.

**Em termos de organização do código**, para cada tabela xxx da BD cria-se uma classe `xxxTableDataGateway` tem métodos que permitem fazer **as operações de procura** requeridos pela lógica de negócio e que permitem **inserir novos registos, apagar um registo ou alterar um registo**. Os métodos de procura emitem **queries SQL**. Os resultados das queries contêm usualmente **vários registos**

- Para devolver esta informação estruturada pode-se embrulhá-la num mapa (e.g., `HashMap`) usando a chave primária como chave do mapa.
- Outra alternativa é devolver o objeto *Record Set* devolvido pela API na query SQL (e.g., classe `ResultSet` no JDBC).

Um **Record Set** é uma representação em memória do resultado de uma *query*. Tem a vantagem de não estar ligado à BD, podendo ser enviado entre aplicações.



### Exemplos de utilização:

- Inserir um novo cliente:
  1. Obter o objeto `CustomerTableDataGateway`
  2. Chamar o método `insert` que trata da criação com os dados de input necessários (notar que pode haver mais do que um)
- Apagar cliente:
  1. Obter o objeto `CustomerTableDataGateway`
  2. Chamar o método `delete` com o valor do `id` (chave) como input

Aqui está um exemplo simples em Java de um *Table Data Gateway* para uma tabela de “Produtos” em uma base de dados:

```

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class ProdutoTableGateway {
 private Connection connection;

 public ProdutoTableGateway() throws SQLException {
 // Inicialização da conexão com a base de dados
 connection = DriverManager.getConnection("jdbc:mysql://localhost/minhabd", "utilizad...
 }

 public ResultSet listarTodosProdutos() throws SQLException {
 String sql = "SELECT * FROM produtos";
 PreparedStatement statement = connection.prepareStatement(sql);
 return statement.executeQuery();
 }

 public void inserirProduto(String nome, double preco) throws SQLException {
 String sql = "INSERT INTO produtos (nome, preco) VALUES (?, ?)";
 PreparedStatement statement = connection.prepareStatement(sql);
 statement.setString(1, nome);
 statement.setDouble(2, preco);
 statement.executeUpdate();
 }

 public void atualizarProduto(int id, String nome, double preco) throws SQLException {
 String sql = "UPDATE produtos SET nome = ?, preco = ? WHERE id = ?";
 PreparedStatement statement = connection.prepareStatement(sql);
 statement.setString(1, nome);
 statement.setDouble(2, preco);
 statement.setInt(3, id);
 statement.executeUpdate();
 }

 public void removerProduto(int id) throws SQLException {
 String sql = "DELETE FROM produtos WHERE id = ?";
 PreparedStatement statement = connection.prepareStatement(sql);
 statement.setInt(1, id);
 statement.executeUpdate();
 }

 public void fecharConexao() throws SQLException {
 connection.close();
 }
}

copy

```

Neste exemplo:

- A classe `ProdutoTableGateway` representa o *Table Data Gateway* para a tabela “produtos” da base de dados. Possui métodos para listar todos os produtos, inserir, atualizar e remover produtos.
- Os métodos usam SQL para interagir com a base de dados. A classe também gere a conexão com a base de dados.
- O *Table Data Gateway* fornece uma interface simples para que os clientes do sistema possam

realizar operações na base de dados (na tabela de produtos), sem precisar lidar diretamente com o SQL.

O padrão *Table Data Gateway* é uma abordagem eficaz quando se trata de operações em uma tabela específica da base de dados, mas pode se tornar complexo em sistemas com várias tabelas e relacionamentos complexos. Nesses casos, outras abordagens, como ORMs (*Object-Relational Mapping*) ou o padrão *Repository*, podem ser mais apropriadas.

## 5.4 Active Record

O padrão **Active Record** é um padrão de design de software que combina os dados (estado) e a lógica de negócios (comportamento) de uma entidade numa única classe. Essa classe representa um registo ou uma linha em uma tabela de uma base de dados relacional. O padrão **Active Record** é amplamente usado em sistemas que requerem uma interação direta e simplificada com a base de dados.

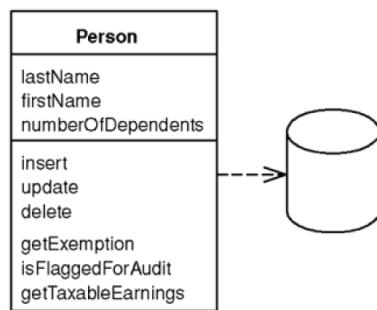
Aqui estão os principais conceitos associados ao padrão *Active Record*:

- **Classe Active Record:** Cada entidade ou registo na base de dados é representado por uma classe específica, conhecida como classe *Active Record*. Essa classe encapsula tanto os dados (campos da tabela) como as operações (métodos) relacionadas a essa entidade.
- **Mapeamento Objeto-Relacional (ORM):** O *Active Record* é frequentemente implementado usando um ORM (*Object-Relational Mapping*), que automatiza a correspondência entre os objetos em código e as tabelas na base de dados.
- **Operações de CRUD:** A classe *Active Record* geralmente fornece métodos para realizar operações de CRUD (Create, Read, Update, Delete) diretamente na base de dados. Inclui métodos para criar, recuperar, atualizar e remover registos.
- **Encapsulamento:** O *Active Record* encapsula tanto os detalhes de acesso aos dados (consultas SQL) como as regras de negócios relacionadas à entidade. Isso significa que os clientes que usam a classe *Active Record* não precisam escrever SQL ou entender os detalhes de como os dados são armazenados na base de dados.
- **Um Objeto por Registro:** Cada instância da classe *Active Record* representa um único registo ou linha na tabela correspondente da base de dados. Isso permite que os clientes manipulem os registos individualmente como objetos.
- **Identificação Única:** Cada registo em uma tabela geralmente possui uma chave primária exclusiva (como um ID) que é usado para identificar e acessar o registo por meio da classe *Active Record*.

No padrão *Active Record* há uma fusão de funcionalidades ao contrário dos padrões anteriores (gateways) que eram puros. O *Active Record* é um padrão híbrido já que se aplica à organização de código que trata simultaneamente da lógica de negócio e da lógica de acesso aos dados. Cada objecto corresponde a **um registo** da respetiva tabela. A classe inclui métodos para gerir a BD e métodos que incluem lógica do domínio.

- Neste caso o objecto da classe inclui **dados e comportamento**,
- É apropriado quando as classes reflectem a estrutura da BD (como no *Domain Model*),
- **Os atributos da classe devem corresponder à colunas da tabela,**
- Cada classe *Active Record* é responsável pelo I/O da sua tabela bem como pela lógica do domínio que influencia estes dados.

Uma instância representa uma única linha da tabela. Após a criação de um objeto, uma nova linha é adicionada à tabela (após a operação de escrita). Qualquer objeto carregado obtém suas informações da BD. Quando uma instância é atualizada, a linha correspondente na tabela também é atualizada. A classe implementa métodos de acesso para cada coluna da tabela. Esse padrão é comumente usado por ferramentas de persistência de objetos e no mapeamento relacional de objetos (ORM).



Métodos típicos numa classe Active Record:

- Construtor que recebe **um registo de um SQL result set**
- Construtor de uma nova instância trata de **inserir na tabela**
- Métodos **finder de classe** que devolvem um ou mais objectos Active Record
- Métodos para **inserção e update** do objeto corrente na tabela
- Getters e setters para os campos da tabela/atributos da classe (fazem automaticamente as instruções SQL adequadas)
- Métodos específicos da lógica do domínio que tenham a ver com a respectiva tabela

No padrão Active Record a separação entre a BD e a lógica do domínio é menor.

- Se o domínio for pequeno, isto não representa uma grande desvantagem
- Além dos métodos de negócio, a classe tem responsabilidade de persistir os seus objetos
- Adequado igualmente para usar com o padrão Transaction Script

Veremos como automatizar este padrão usando a tecnologia JPA.

Aqui está um exemplo simples em Java de uma classe *Active Record* para uma tabela de “Clientes” numa base de dados:

```

public class Cliente {
 private int id;
 private String nome;
 private String email;

 // Construtor
 public Cliente(int id, String nome, String email) {
 this.id = id;
 this.nome = nome;
 this.email = email;
 }

 // Métodos para operações de CRUD
 public static Cliente procurarPorId(int id) {
 // Lógica para consultar a base de dados e criar um objeto Cliente
 return new Cliente(id, "Cliente Teste", "cliente@teste.com");
 }

 public void salvar() {
 // Lógica para inserir ou atualizar o registo na base de dados com os dados deste objeto
 }

 public void remover() {
 // Lógica para remover o registo da base de dados
 }

 // Getters e setters
 // ...
}

```

copy

Neste exemplo:

- A classe `Cliente` representa um registo da tabela “Clientes” da base de dados. Possui atributos que mapeiam os campos do registo, como `id`, `nome` e `email`.
- Os métodos estáticos como `procurarPorId` são usados para consultar a base de dados e criar objetos `Cliente` com base nos dados encontrados.
- Os métodos de instância `salvar` e `remover` são usados para inserir ou atualizar o registo na base de dados e para excluir o registo, respectivamente.

O padrão *Active Record* é adequado para sistemas onde a interação direta com regtos da base de dados é necessária e onde a complexidade do mapeamento objeto-relacional é relativamente baixa. No entanto, para sistemas mais complexos com relacionamentos complicados entre tabelas e regras de negócios mais sofisticadas, outras abordagens, como ORMs (*Object-Relational Mapping*) ou o padrão *Domain Model*, podem ser mais apropriadas.

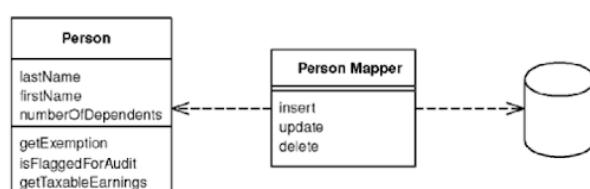
## 5.5 Data Mapper

O padrão **Data Mapper** é um padrão de desenho de software que separa a lógica de negócios de uma aplicação da camada de persistência de dados. Ele trata de mapear objetos de domínio em estruturas de dados usadas para armazenamento ou vice-versa. A ideia central do *Data Mapper* é garantir que as classes de domínio (objetos de negócios) não tenham conhecimento direto de como os dados são armazenados ou recuperados da base de dados. Em vez disso, essa responsabilidade é delegada a uma classe separada chamada “mapper” ou “data mapper.”

Aqui estão os principais conceitos associados ao padrão *Data Mapper*:

- **Classe Mapper:** Uma classe *Mapper*, muitas vezes chamada “*Data Mapper*”, é responsável por mapear objetos de domínio em estruturas de dados de armazenamento, como tabelas da base de dados, e vice-versa. O *Mapper* encapsula a lógica de persistência e recuperação de dados.
- **Separação de Responsabilidades:** O *Data Mapper* promove a separação de responsabilidades, garantindo que as classes de domínio não tenham conhecimento de como seus dados são armazenados ou recuperados. Isso permite uma melhor manutenção e testabilidade do código.
- **Mapeamento Bidirecional:** O *Data Mapper* suporta o mapeamento bidirecional, o que significa que ele permite a conversão de objetos de domínio em estruturas de dados de armazenamento e vice-versa. Isso é útil para operações de leitura e escrita.
- **Encapsulamento:** A classe *Mapper* encapsula detalhes específicos da camada de persistência, como consultas SQL ou instruções de atualização. Isso mantém os detalhes de armazenamento de dados isolados do restante do código.
- **Facilidade de Teste:** A separação de responsabilidades facilita a criação de testes unitários para as classes de domínio, pois elas não têm dependências diretas da base de dados.
- **Flexibilidade:** O *Data Mapper* permite a flexibilidade na escolha do mecanismo de armazenamento, permitindo a troca de uma base de dados por outro ou até mesmo a adoção de diferentes formas de armazenamento, como armazenamento em memória ou sistemas de arquivos.

Existe uma camada intermédia (o mapper) que mantém independentes a informação armazenada nos objectos e a informação da BD.



O mapper tem a responsabilidade de transferir a informação entre a parte OO e a parte relacional. Os objectos não precisam sequer saber que existe uma BD, eles não incluem qualquer código SQL nem conhecimento sobre a estrutura das tabelas (o inverso, a BD ser ignorante da camada de domínio, é sempre verdade).

Assim, ambas as camadas podem evoluir e ser testadas separadamente.

Um data mapper é um exemplo do padrão **Mapper** (Fowler, cap.18). É uma classe que estabelece uma comunicação entre dois objectos/sistemas independentes. A classe envia informação para os dois lados sem que qualquer um dos lados saiba sequer que ele exista! Isto significa que não há referências ao mapper nos dois objectos em questão.

Para implementar pode-se usar um terceiro objecto que gere a execução do mapper, ou fazer do mapper um observador de um ou dos dois sub-sistemas (que reage quando há uma alteração nos estados dos sistemas que interliga)

Este padrão é apropriado para modelos de domínio complexos. É possível ignorar os detalhes da BD no desenho, na construção e no teste da camada lógica.

O cliente para carregar uma pessoa da BD, invoca o método `insert` do *mapper*. Este vai conferir se essa pessoa já existe em memória e, se não, vai buscar a informação à BD. Depois cria um novo objecto `Person` com a informação recolhida, e retorna essa referência para o cliente.

Aqui está um exemplo simples em Java de um *Data Mapper* para uma entidade de “Produto”:

```
public class Produto {
 private int id;
 private String nome;
 private double preco;

 // Getters e setters
 // ...
}

public class ProdutoMapper {
 public Produto buscarPorId(int id) {
 // Lógica para consultar a base de dados e criar um objeto Produto
 // A consulta SQL e a lógica de mapeamento estão encapsuladas aqui
 return new Produto();
 }

 public void salvar(Produto produto) {
 // Lógica para inserir ou atualizar o registo na base de dados com base no objeto Produto
 // A inserção/atualização e a lógica de mapeamento estão encapsuladas aqui
 }

 public void remover(Produto produto) {
 // Lógica para remover o registro da base de dados com base no objeto Produto
 // A remoção está encapsulada aqui
 }
}

copy
```

Neste exemplo:

- A classe `Produto` é uma classe do domínio que representa um produto com atributos `id`, `nome` e `preço`.
- A classe `ProdutoMapper` é responsável por mapear objetos de domínio da classe `Produto` em registros da base de dados e vice-versa. Encapsula a lógica de persistência, incluindo consultas SQL e operações de inserção/atualização/remoção.

O padrão *Data Mapper* é particularmente útil em sistemas complexos onde a lógica de negócios e os requisitos de persistência são extensos e onde a separação de responsabilidades é fundamental para manter um código organizado e facilmente mantido. Ele é frequentemente usado em conjunto com ORMs (Object-Relational Mapping) para simplificar ainda mais o mapeamento entre objetos de domínio e bases de dados relacionais.

## 5.6 JDBC

Java DataBase Connectivity (JDBC) é a API que permite ligar uma aplicação desenvolvida em Java a uma base de dados relacional ou a dados sob forma de folhas de cálculo. A API estabelece um contrato entre

1. quem programa a aplicação e
2. quem fornece a ligação à base de dados específica (Oracle, MySQL, Derby, ObjectDB etc...)

A biblioteca JDBC inclui APIs para cada uma das tarefas mencionadas abaixo que são comumente associadas ao uso de uma base de dados.

- Criação uma conexão com uma base de dados,
- Criação de instruções SQL ou MySQL,
- Execução de queries SQL na base de dados,
- Visualização e modificação os registos.

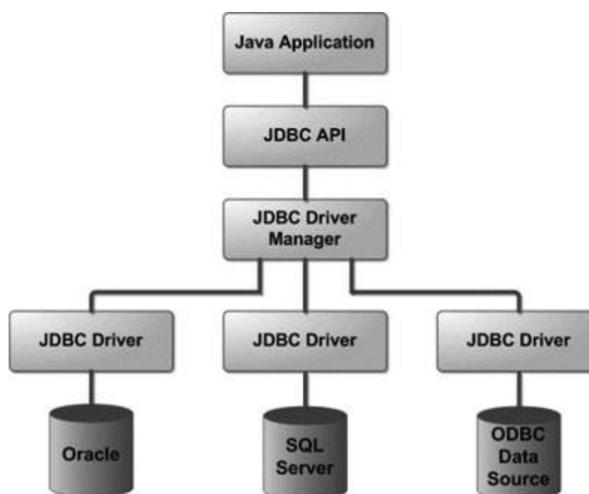
O modelo usado é de tipo cliente/servidor:

1. Cliente cria uma **ligação** à fonte de dados física (servidor de base de dados)
2. Cliente envia **comandos** ou **interrogações** SQL para serem executados através dessa ligação
3. Servidor envia de volta um **conjunto de resultados**
4. Cliente processa os resultados
5. Cliente **fecha a ligação**

Todos os erros são reportados através de exceções e os conjuntos de resultados têm associados metadados.

Geralmente a arquitectura do JDBC consiste em duas camadas:

- JDBC API: gera interface entre a aplicação e o JDBC driver Manager
- JDBC Driver API: abstrai a interface com vários tipos de bases de dados.



Componentes do JDBC p.102 • Exemplo básico de uso do JDBC p.102  
 • Statements p.103 • ResultSet p.106

### 5.6.1 Componentes do JDBC

A API JDBC fornece as seguintes interfaces e classes:

- **DriverManager**: Esta classe faz a gestão dos drivers de base de dados. Transmite as solicitações de conexão da aplicação java para o driver de base de dados apropriado usando o subprotocolo de comunicação. O primeiro driver que reconhece um determinado subprotocolo em JDBC será usado para estabelecer uma conexão de base de dados.
- **Driver**: Esta interface lida com as comunicações com o servidor de base de dados. Geralmente o programador da aplicação não vai interagir diretamente objetos do tipo Driver. Em vez disso, instâncias de **DriverManager** são usadas.
- **Connection**: Esta interface contém todos os métodos para estabelecer a conexão uma base de dados e gerir os recursos associados. Toda a comunicação com a base de dados é feita apenas através do objeto de **Connection**.
- **Statement**: Os objetos criados a partir desta interface representam instruções SQL para o base de dados. Algumas interfaces derivadas aceitam parâmetros além de executar procedimentos armazenados.
- **ResultSet**: Esses objetos contêm dados recuperados da base de dados, a seguir a execução de uma query. O **ResultSet** atua como um iterador para permitir navegar pelos resultados obtidos.
- **SQLException**: Esta classe é usada para assinalar qualquer erro ocorrido na interação com a base de dados.

### 5.6.2 Exemplo básico de uso do JDBC

Existem cinco etapas a seguir efetuar uma interação com a base de dados usando o JDBC:

1. **Abrir uma conexão** Requer o uso do método `DriverManager.getConnection()` para criar um objeto `Connection`, que representa uma conexão física com a base de dados.
2. **Executar uma consulta** Requer o uso de um objeto do tipo `Statement` para construir e enviar uma instrução SQL para a base de dados.
3. **Extraír dados do conjunto de resultados** Requer o uso do método `ResultSet.getXXX()` apropriado para recuperar os dados do conjunto de resultados.
4. **Limpar o ambiente** Requer o fecho explícito de todos os recursos da base de dados.

Mais em detalhes:

1. **Importar os pacotes necessários:**

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

copy
```

2. **Definir constantes para os dados de acesso:**

```
// definir o endereço da base de dados:
private final String jdbcURL = "jdbc:mysql://localhost:3306/base_de_dados";
//
private final String username = "username_no_servidor_de_BD";
private final String password = "password";

copy
```

3. **Estabelecer a conexão:** Use o método `DriverManager.getConnection()` para estabelecer uma conexão com a base de dados. A chamada a este método deve estar inserida num bloco `try-catch` para lidar com exceções :

```

Connection connection = null;
try {
 connection = DriverManager.getConnection(jdbcURL, username, password);
 if (connection != null) {
 System.out.println("Conexão establecida!");
 }
} catch (SQLException e) {
 System.err.println("Erro ao conectar ao banco de dados: " + e.getMessage());
}
copy

```

#### 4. Fechar a Conexão

```

try {
 if (connection != null) {
 connection.close();
 System.out.println("Conexão fechada.");
 }
} catch (SQLException e) {
 System.err.println("Erro ao fechar a conexão: " + e.getMessage());
}
copy

```

Para realizar uma *query* é necessário usar um objeto de tipo **Statement**.

### 5.6.3 Statements

Os dois principais tipos de **Statement** são:

- **Statement** Útil quando você está usando instruções SQL estáticas em tempo de execução. Uma instância de **Statement** não pode aceitar parâmetros.
- **PreparedStatement** Uma versão mais segura e mais eficiente, que admite parâmetros nas queries.

As instâncias de **Statement** são obtidas usando o método **createStatement()** da classe **Connection**. A criação e o uso de uma inscrução é feito dentro de um bloco try-catch:

```

Statement stmt = null;
try {
 stmt = conn.createStatement();
 // ...
}
catch (SQLException e) {
 // ...
}
finally {
 stmt.close();
}
copy
ou
try (Statement stmt = conn.createStatement()) {
 // ...
} catch (SQLException e) {
 // ...
}
copy

```

Neste segundo caso a instrução é fechada automaticamente.  
O objeto de tipo **Statement** é usado para executar uma query:

```
ResultSet result = stmt.executeQuery(sql);
```

**copy**

O parâmetro do método é uma string que contém a query SQL. O resultado é colocado numa instância de **ResultSet**. A chamada a este método pode provocar o lançamento da exceção **SQLException** será portanto necessário incluir a chamada num bloco **try-catch** ou propagar a exceção.

Aqui está um exemplo:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class JDBCExample {
 public static void main(String[] args) {
 // definir o endereço da base de dados:
 private final String jdbcURL = "jdbc:mysql://localhost:3306/base_de_dados";
 //
 private final String username = "username_no_servidor_de_BD";
 private final String password = "password";
 try {
 Connection connection = DriverManager.getConnection(jdbcURL, username, password);
 Statement statement = connection.createStatement();

 String sql = "SELECT * FROM exemplo";
 ResultSet resultSet = statement.executeQuery(sql);

 while (resultSet.next()) {
 int id = resultSet.getInt("id");
 String nome = resultSet.getString("nome");
 System.out.println("ID: " + id + ", Nome: " + nome);
 }

 resultSet.close();
 statement.close();
 connection.close();
 } catch (SQLException e) {
 e.printStackTrace();
 }
 }
}
```

**copy**

Vamos ver mais em detalhe o funcionamento da classe **ResultSet** na próxima secção.

No caso dos **PreparedStatements**, em vez de usar o método **createStatement**, usa-se o método **prepareStatement**:

```
try (PreparedStatement pstmt = connection.prepareStatement(SQL)) {
 // ...
} catch (SQLException e) {
 // ...
}
```

**copy**

Onde **SQL** é uma instrução onde são assinalados com **?** os parâmetros da query ex.:

```
Update Employees SET age = ? WHERE id = ?
```

Os métodos **setXXX()** associam valores aos parâmetros, em que **XXX** representa o tipo de dados Java do valor que deseja vincular ao parâmetro de entrada. Se esquecer de fornecer os valores, receberá um erro (**SQLException**).

Cada marcador de parâmetro é referido por sua posição ordinal. O primeiro marcador representa a posição 1, o próximo a posição 2 e assim por diante. Este método difere dos índices de array Java, que começa em 0.

Exemplo de consulta com **PreparedStatement**:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class PreparedStatementExample {
 public static void main(String[] args) {
 // definir o endereço da base de dados:
 private final String jdbcURL = "jdbc:mysql://localhost:3306/base_de_dados";
 //
 private final String username = "username_no_servidor_de_BD";
 private final String password = "password";
 try {
 Connection connection = DriverManager.getConnection(jdbcURL, username, password);

 // Consulta SQL parametrizada
 String sql = "SELECT * FROM exemplo WHERE id = ?";

 int idParaConsultar = 1; // Parâmetro da consulta

 PreparedStatement preparedStatement = connection.prepareStatement(sql);
 preparedStatement.setInt(1, idParaConsultar); // Define o valor do parâmetro

 ResultSet resultSet = preparedStatement.executeQuery();

 while (resultSet.next()) {
 int id = resultSet.getInt("id");
 String nome = resultSet.getString("nome");
 System.out.println("ID: " + id + ", Nome: " + nome);
 }
 resultSet.close();
 preparedStatement.close();
 connection.close();
 } catch (SQLException e) {
 e.printStackTrace();
 }
 }
}

copy
```

Exemplo de modificação da base de dados. Note que neste caso, usa-se o método **executeUpdate**:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;

public class PreparedStatementInsertExample {
 public static void main(String[] args) {
 // definir o endereço da base de dados:
 private final String jdbcURL = "jdbc:mysql://localhost:3306/base_de_dados";
 //
 private final String username = "username_no_servidor_de_BD";
 private final String password = "password";
```

```
try {
 Connection connection = DriverManager.getConnection(jdbcURL, username, password);

 // Inserção de dados parametrizada
 String sql = "INSERT INTO exemplo (id, nome) VALUES (?, ?)";

 int novoID = 4; // Novo ID a ser inserido
 String novoNome = "Novo Nome"; // Novo nome a ser inserido

 PreparedStatement preparedStatement = connection.prepareStatement(sql);
 preparedStatement.setInt(1, novoID); // Define o valor do primeiro parâmetro
 preparedStatement.setString(2, novoNome); // Define o valor do segundo parâmetro

 int linhasAfetadas = preparedStatement.executeUpdate();

 System.out.println("Linhas afetadas: " + linhasAfetadas);

 preparedStatement.close();
 connection.close();
} catch (SQLException e) {
 e.printStackTrace();
}
}
```

copy

Os principais métodos das classes Statement e PreparedStatement são:

- `int executeUpdate()` Retorna o número de linhas afetadas pela execução da instrução SQL. Use este método para executar instruções SQL para as quais você espera obter um número de linhas afetadas - por exemplo, uma instrução `INSERT`, `UPDATE` ou `DELETE`.
  - `ResultSet executeQuery()` Retorna um objeto de tipo `ResultSet`. Use este método quando espera obter um conjunto de resultados, como faria com uma instrução `SELECT`.

#### **5.6.4 ResultSet**

As instruções SQL que leem dados de uma consulta da base de dados, retornam os dados num conjunto de resultados. A instrução `SELECT` é a maneira padrão de selecionar linhas da base de dados e visualizá-las em um conjunto de resultados. A interface `java.sql.ResultSet` representa o conjunto de resultados de uma consulta à base de dados.

Um objeto **ResultSet** mantém um cursor que aponta para a linha atual no conjunto de resultados. O termo “conjunto de resultados” refere-se aos dados de linha e coluna contidos em um objeto **ResultSet**. Os métodos da interface **ResultSet** podem ser divididos em três categorias :

- Métodos de navegação - usados para mover o cursor.
  - Métodos Get - Usados para visualizar os dados nas colunas da linha atual apontada pelo cursor.
  - Métodos de atualização - usados para atualizar os dados nas colunas da linha atual. As atualizações também podem ser atualizadas na base de dados subjacente.

O cursor pode ser movido com base nas propriedades do `ResultSet`. Essas propriedades são designadas quando o `Statement` correspondente que gera o `ResultSet` é criado.

Métodos para deslocar o cursor:

- public boolean absolute(int row) throws SQLException **colocar o cursos na linha indicada.**
  - public boolean relative(int row) throws SQLException **desloca o cursor relativamente a posição corrente. O parâmetro pode ser positivo ou negativo.**

- `public boolean previous() throws SQLException` desloca o cursor para a linha anterior. Retorna `false` se estiver na primeira linha.
- `public boolean next() throws SQLException` desloca o cursor para a linha seguinte. Retorna `false` se estiver na última linha.

A interface `ResultSet` contém **dezenas de métodos** para obter os dados da linha atual. Existe um método `get` para cada um dos tipos de dados possíveis, e cada método `get` tem duas versões:

- uma que aceita um nome de coluna.
- uma que leva como parâmetro um índice de coluna.

Modificação de um `ResultSet`.

A interface `ResultSet` contém uma coleção de métodos de atualização para atualizar os dados de um conjunto de resultados.

Tal como acontece com os métodos `get`, existem dois métodos de atualização para cada tipo de dados:

- Um que aceita um nome de coluna.
- Um que leva em um índice de coluna.

Por exemplo, para atualizar uma coluna de tipo `String` da linha atual de um conjunto de resultados, pode-se usar um dos seguintes métodos `updateString()`:

- `public void updateString(int columnIndex, String s) throws SQLException`
- `public void updateString(String columnName, String s) throws SQLException`

Existem métodos de atualização para os oito tipos de dados primitivos, bem como `String`, `Object`, `URL` e os tipos de dados SQL no pacote `java.sql`.

Atualizar uma linha no conjunto de resultados altera as colunas da linha atual no objeto `ResultSet`, mas não na base de dados subjacente. Para atualizar suas alterações na linha da base de dados, precisa invocar um dos seguintes métodos.

- `public void updateRow()`
- `public void deleteRow()`
- `public void refreshRow()` Atualiza os dados no conjunto de resultados para refletir quaisquer alterações recentes na base de dados.
- `public void cancelRowUpdates()` Cancela as alterações feitas nesta linha do `ResultSet`.

## 5.7 Mapeamento Orientado Objeto

O uso de uma base de dados relacional é um elemento essencial de uma aplicação empresarial. A linguagem SQL é quase universal e constitui o protocolo usado para interagir com as bases de dados. No entanto o mundo “Orientado Objetos” é distinto do mundo de BD relacionais. A representação, os tipos de associações são diferentes. Equanto os objetos guardam referências para outros objetos associados, as tabelas associam-se via chaves estrangeiras. Os objetos facilmente guardam nos seus atributos coleções de outros objetos. A normalização das tabelas exige que as relações entre tabelas sejam representadas por valores únicos.

É portanto necessário definir uma estratégia para mapear entre esses dois mundos, é o objetivo do Object Relational Model (ORM).

É uma técnica de programação usada no desenvolvimento de software para criar uma ponte entre linguagens de programação orientadas a objetos (OOP) e bases de dados relacionais.

O principal propósito do ORM é simplificar operações sobre uma base de dados, abstraiendo as interações com o banco de dados e mapeando registros da base de dados para objetos no código da aplicação. Isso significa que os programadores podem manipular dados da base de dados usando conceitos familiares de programação orientada objetos, como classes, objetos e métodos. Alguns pontos-chave sobre ORM:

- **Mapeamento:** As ferramentas ORM definem como os dados numa base de dados correspondem a objetos no código da aplicação. Isso é frequentemente chamado de “mapeamento objeto-relacional” porque estabelece um mapeamento entre a base de dados relacional e a aplicação orientada a objetos.

- **Abordagem Orientada a Objetos:** O ORM permite que os programadores trabalhem com bases de dados usando princípios orientados a objetos, tornando mais fácil desenvolver e manter aplicações.
- **Abstração SQL:** As ferramentas ORM geram consultas SQL nos bastidores, eliminando a necessidade de os desenvolvedores escreverem código SQL manualmente para a maioria das operações.
- **Portabilidade:** O ORM pode tornar os aplicativos mais independentes da base de dados, permitindo que os programadores escrevam código que funcione com vários sistemas de gestão de bases de dados (DBMS) sem fazer alterações extensas no código.
- **Relacionamentos Complexos:** O ORM lida com relacionamentos complexos entre tabelas da base de dados, fornecendo mecanismos para navegar e manipular dados relacionados como objetos e coleções.
- **Cache :** Muitos frameworks ORM incluem mecanismos de cache para melhorar o desempenho, reduzindo o número de consultas ao banco de dados.

Frameworks ORM populares em várias linguagens de programação incluem o Hibernate (Java), o Entity Framework (C#), o Django ORM (Python) e o Sequelize (Node.js), entre outros. Esses frameworks simplificam as interações com bases de dados e ajudam os programadores a construir aplicações mais fáceis de manter e eficientes.

### 5.7.1 Mapeamento das relações

As relações entre objetos em um ORM (Object-Relational Mapping) são mapeadas usando técnicas específicas que permitem ao ORM entender e representar as associações e relacionamentos entre as tabelas da base de dados como objetos e classes no código da aplicação. Existem várias maneiras de mapear relações entre objetos num ORM, e a escolha da técnica depende da natureza dos relacionamentos. Aqui estão algumas das técnicas mais comuns:

- **Relação Um-para-Um**

- Neste tipo de relação, uma entidade no lado “Um” está associada a exatamente uma entidade no lado “Um”. Conforme os casos o dono da relação pode ser um
- No ORM, isso pode ser mapeado usando um atributo em numa classe para representar a relação direta entre duas tabelas.

- **Relação Um-para-Muitos**

- Este é uma das relações mais comuns. Numa base de dados, é frequentemente representado usando uma chave estrangeira numa tabela (lado “Muitos”) que faz referência à chave primária noutra tabela (lado “Um”).
- No ORM, isso é mapeado usando coleções ou listas numa classe. Por exemplo, uma classe `Cliente` pode ter uma coleção de pedidos representando a relação “um cliente possui muitos pedidos”.

- **Relação Muitos-para-Muitos**

- Este tipo de relação envolve várias entidades de ambos os lados.
- Na base de dados, é geralmente representado por uma tabela de associação que liga as tabelas das duas entidades.
- No ORM, isso é frequentemente mapeado usando coleções em ambas as classes envolvidas na relação. Por exemplo, uma classe `Estudante` pode ter uma coleção de `Cursos`, e a classe `Curso` também pode ter uma coleção de `Estudantes`.

Num ORM, é possível criar relações bidirecionais, onde as duas classes envolvidas na relação podem aceder uma à outra. Para isso, pode usar mapeamentos específicos em ambas as classes para indicar a direção da relação.

### 5.7.2 Mapeamento da herança

Até agora apenas falámos de relações de associação como a composição. Mas o mundo Orientado Objetos também inclui associações por herança.

- **Tabela única (Single Table Inheritance)**
- **Tabela por Classe Concreta (Concrete Table Inheritance)**
- **Tabela por Subclasse (Table per Subclass)**

#### Single Table Inheritance

Neste padrão é apenas usada uma tabela para representar todas as instâncias de todas as classes na hierarquia da herança. Quando se cria um objeto a partir de um registo da tabela é necessário saber a que classe pertence. Para isso existe uma **coluna na tabela** com essa informação (type) cujo valor pode ser o nome da classe.

Aqui está um exemplo simplificado de um diagrama de classes UML que ilustra a estratégia de *Single Table Inheritance*:

```
+-----+
| Vehicle | (Tabela Única no Banco de Dados)
+-----+
| - id: int |
| - type: string |
| - make: string |
| - model: string |
| - ... (outros campos comuns a todos os veículos)
+-----+
|
|
| (Herança)
|
+
+-----+
| Car |
+-----+
| - numDoors: int |
| - color: string |
| - ... (outros campos específicos de Carro)
+-----+
|
|
| (Herança)
|
+
+-----+
| Motorcycle |
+-----+
| - hasFairing: bool|
| - ... (outros campos específicos de Motocicleta)
+-----+
```

Neste exemplo:

- **Vehicle** é a classe base que contém os campos comuns a todos os tipos de veículos.
- **Car** e **Motorcycle** são classes derivadas que herdam de **Vehicle** e incluem campos específicos de carro e motocicleta, respectivamente.
- Na base de dados, todos os dados de veículos (carros e motocicletas) são armazenados em uma única tabela chamada **Vehicle**. O campo **type** é usado para determinar o tipo específico de veículo.

Esta é uma estratégia típica de herança para o mapeamento de herança em ORM quando se utiliza a abordagem de *Single Table Inheritance*. A tabela única na base de dados contém todos os campos de todas as classes envolvidas na herança, com um campo de tipo (como `type` no exemplo) para distinguir entre os tipos específicos de objetos.

- **Vantagens**

- **Apenas existe uma tabela,**
  - Não é preciso fazer *joins* para recolher os dados,
  - Um *refactoring* que transfira um atributo entre as classes não altera a BD.

- **Desvantagens**

- **Gasto de memória:** muitos campos dos registo estarão vazios, cada campo só faz sentido no contexto da classe a que o registo pertence.
- O âmbito dos nomes das classes é comum: **não se pode ter atributos com o mesmo nome nas classes** (pode resolver-se prefixando o nome dos campos com o nome da respetiva classe).

## Concrete Table Inheritance

Neste padrão é usado uma tabela por **classe concreta**. As classes abstratas (como a superclasse) não são representadas por tabelas (dado não se declarar objetos de classes abstratas).

Mais em detalhes:

- **Tabela por Classe Concreta**

- Cada classe concreta na hierarquia de herança é mapeada para sua **própria tabela** na base de dados. Não existe uma tabela para a classe base (superclasse).
- Cada tabela contém **todos os campos** específicos dessa classe concreta, **incluindo os campos herdados** da classe base, se houver.
- A estrutura da base de dados reflete diretamente a estrutura da hierarquia de classes no código da aplicação.

- **Independência entre Tabelas**

- As tabelas das classes concretas são independentes umas das outras. Não há chaves estrangeiras entre elas.
- Significa que cada tabela é autossuficiente em termos de representar os objetos da classe concreta.

Esta abordagem tem vantagens e desvantagens :

- **Vantagens**

- Cada tabela é **auto-contida** e não há campos irrelevantes.
- Não há a necessidade de joins
- Cada tabela só é acedida quando a respetiva classe é acedida, não há *bottlenecks* à partida

- **Desvantagens**

- Complexidade de Consulta: Consultas que envolvem objetos de várias classes concretas na hierarquia podem ser complexas, pois requerem a junção de várias tabelas.
- Na BD não são expressas as relações com as classes abstratas,
- Um refactoring que transfira um atributo entre as classes altera a BD,
- Se existe uma mudança na superclasse, todas as tabelas têm de ser modificadas,

## Class Table Inheritance

Neste padrão é usado **uma tabela por classe**. A consequência é que um objeto pode ter a sua informação **distribuída entre várias tabelas**. Uma solução é usar a chave da superclasse como chave estrangeira nas restantes tabelas.

Por exemplo suponha que tem uma hierarquia de classes de funcionários que inclui três tipos de funcionários: Funcionario (classe base), Gerente e Programador. Cada um deles tem atributos específicos, que devem ser mapeados para uma base de dados usando a estratégia de *Class Table Inheritance*:

```
+-----+
| Employee | (Tabela Employee)
+-----+
| - id: int |
| - name: string |
| - salary: decimal|
+-----+
 ^
 |
 |
+-----+
| Manager | (Tabela Manager)
+-----+
| - id: int |
| - teamSize: int |
| - ... (outros campos específicos de Gerente)|
+-----+
 ^
 |
 |
+-----+
| Developer | (Tabela Developer)
+-----+
| - id: int |
| - programmingLang: string |
| - ... (outros campos específicos de Desenvolvedor)|
+-----+
```

Neste exemplo:

- `Employee` é a classe base que contém atributos comuns a todos os funcionários, como ID, nome e salário.
- `Manager` e `Developer` são classes derivadas que herdam de `Employee` e incluem atributos específicos de gerente e programador, respectivamente.
- A tabela `Employee` contém os campos comuns a todos os funcionários, incluindo um ID exclusivo.
- As tabelas `Manager` e `Developer` contêm apenas os campos específicos de cada tipo de funcionário e também incluem uma chave estrangeira para a tabela `Employee` para estabelecer a relação de herança.

Permite manter uma estrutura de base de dados normalizada e eficiente, ao mesmo tempo em que reflete a hierarquia de herança de classes no código da aplicação. Cada tabela representa uma classe específica e seus campos exclusivos, enquanto a chave estrangeira para a tabela base mantém a relação entre as classes.

No entanto esta abordagem tem algumas desvantagens:

- **Complexidade da Consulta** Consultas que envolvem todos os tipos de objetos da hierarquia de herança podem ser complexas, pois requerem a junção de várias tabelas. Pode afetar o desempenho e a legibilidade das consultas SQL geradas pelo ORM.
- **Desempenho em Gravação** Operações de gravação, como inserção, atualização e remoção, podem ser mais lentas devido à necessidade de operar em várias tabelas para manter a consistência dos dados.

É importante destacar que a escolha entre a estratégia de *Class Table Inheritance* e outras estratégias, como *Single Table Inheritance* ou *Table per Subclass*, depende dos requisitos específicos do projeto, do modelo de dados e das prioridades de desempenho e manutenção. Cada abordagem tem suas vantagens e desvantagens, e a seleção deve ser feita considerando cuidadosamente as necessidades da aplicação.

## 5.8 Java Persistence API

A Java Persistence API (JPA) é composta por três partes:

- A API (cf. pacote `javax.persistence`)
- Uma linguagem para queries designada JPQL
- Um ORM via metadados (uso de anotações)

Os conceitos do modelo de domínio irão corresponder a tabelas da BD. No JPA isto corresponde a uma **entidade**. Uma entidade é uma classe Java cujo estado de cada instância pode ser armazenado como um registo da respetiva **tabela**. Esta informação bem como a relação entre entidades será descrita por **anotações** e/ou num ficheiro XML que acompanha a aplicação. Existem várias implementações: ObjectDB, OpenJPA (desenvolvido pela Apache), EclipseLink.

**Já estudamos o JDBC para criar aplicações conectadas a uma base de dados relacional, qual é a diferença entre o JDBC et o JPA ?** O JDBC (Java Database Connectivity) e o JPA (Java Persistence API) são duas tecnologias relacionadas à persistência de dados em aplicações Java, mas eles têm abordagens diferentes e são usados em contextos diferentes. Aqui está a diferença fundamental entre os dois:

### JDBC:

- **Nível de Abstração:** O JDBC fornece um nível mais baixo de abstração para a interação com bases de dados. É uma API do Java que permite que os programadores escrevam código Java para comunicar diretamente com uma base de dados relacional.
- **Programação Manual:** No JDBC, é necessário escrever código manualmente para abrir conexões com a base de dados, criar consultas SQL, processar resultados e gerir transações.
- **Controle Completo:** O JDBC oferece um controle completo sobre as operações da base de dados, permitindo otimizar consultas e transações de acordo com os requisitos específicos da aplicação.
- **Mais Complexo:** Por ser mais baixo nível, o JDBC pode ser mais complexo e requer mais código para realizar tarefas comuns de persistência.

### JPA:

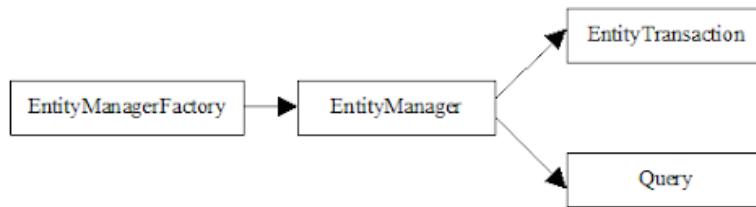
- **Nível de Abstração Superior:** O JPA é uma API de alto nível que simplifica o acesso a bancos de dados relacionais em aplicativos Java. Ele fornece uma camada de abstração sobre o JDBC.
- **Mapeamento Objeto-Relacional:** Uma das principais funcionalidades do JPA é o mapeamento objeto-relacional (ORM). Permite que os objetos Java sejam mapeados diretamente para tabelas da base de dados, eliminando a necessidade de escrever consultas SQL manualmente.
- **Uso de Entidades:** No JPA, trabalha-se com entidades, que são classes Java que representam tabelas da base de dados. As operações de leitura/gravação são realizadas em objetos de entidade, em vez de consultas SQL diretas.
- **Gestão de Transações Simplificado:** O JPA simplifica a gestão de transações por meio de anotações, tornando mais fácil lidar com operações transacionais.

Arquitectura JPA p.[113](#) • EntityManagerFactory p.[113](#) • EntityManager p.[114](#) • EntityTransaction p.[114](#) • Ciclo de vida de uma Entidade p.[115](#) • Entidades JPA p.[116](#) • Relações entre Entidades p.[118](#) • Persistir Entidades p.[120](#) • Persistência por Cascata p.[120](#) • Atributos Temporários p.[121](#) • Recolher Entidades p.[121](#) • Atualizações p.[122](#) • Remoções p.[122](#) • Queries p.[123](#)

### 5.8.1 Arquitectura JPA

Os principais elementos da API são:

- **EntityManager** é uma interface que providencia serviços para interagir com as entidades (e.g., **persist**, **merge**, **remove** que gravam, alteram e apagam entidades da BD).
  - Corresponde a uma ligação à BD. Uma aplicação que precisa de várias ligações, usa várias instâncias desta classe.
  - Serve de fábrica à criação de transações e queries (cf. abaixo).
- **EntityManagerFactory** cria instâncias de **EntityManager**s à medida das necessidades. É parametrizada para lidar com uma BD específica. Normalmente há apenas uma instância por cada BD usada pela aplicação.
- **EntityTransaction** é uma interface que gere as transações. É preciso uma transação activa para inserir/remover dados na BD.
- **Query** Instâncias que permitem fazer queries à BD.



### 5.8.2 EntityManagerFactory

Para criar uma instância de **EntityManagerFactory** é necessário fornecer os dados sobre a BD e definir que classes são entidades, i.e., definir uma **Persistence Unit** (no ficheiro de configuração **persistence.xml**). A instância é criada pelo método de classe **createEntityManagerFactory**:

```
EntityManagerFactory emf =
 Persistence.createEntityManagerFactory("domain-model-jpa");
```

**copy**

O argumento do método é o nome da unidade de persistência definida no ficheiro **persistence.xml**. O ficheiro tem a estrutura seguinte:

```
<persistence
 xmlns="http://xmlns.jcp.org/xml/ns/persistence"
 schemaLocation="http://xmlns.jcp.org/xml/ns/persistence http://xmlns.jcp.org/xml/ns/persistence_2_1.xsd"
 version="2.1">
 <!-- O nome da unidade de persistência é domain-model-jpa -->
 <persistence-unit name="domain-model-jpa" transaction-type="RESOURCE_LOCAL">
 <!-- As classes enumeradas aqui são as entidades da BD: -->
 <class>domain.Cliente</class>
 <class>domain.Desconto</class>
 ...
 <!-- a seguir estão as definições da BD: -->
 <properties>
 <!-- O URL da BD -->
 <property name="javax.persistence.jdbc.url"
 value="jdbc:derby:data/derby/css000;create=true"/>
 <!-- O driver. Aqui estamos a usar uma base de dados local de tipo Derby -->
 <property name="javax.persistence.jdbc.driver"
 value="org.apache.derby.jdbc.EmbeddedDriver"/>
 <!-- As credenciais para aceder à BD -->
 <property name="javax.persistence.jdbc.user" value="css000"/>
```

```

<property name = "javax.persistence.jdbc.password" value="css000"/>
</properties>
</persistence-unit>
</persistence>
```

copy

Num projeto Maven este ficheiro **deve estar** a pasta `src/main/resources/META-INF/`.

### 5.8.3 EntityManager

O `EntityManager` representa uma conexão à BD. Uma instância de `EntityManagerFactory` deve ser usada para obter um `EntityMabager`:

```

EntityManager em = emf.createEntityManager();
try {
 // Use the EntityManager to access the database
} finally {
 em.close();
}
```

copy

O `EntityManager` deve ser fechado depois de ter sido usado. Outra maneira de fazer o mesmo do que no exerto anterior:

```

try (EntityManager em = emf.createEntityManager()) {
 // Use the EntityManager to access the database
}
copy
```

### 5.8.4 EntityTransaction

Uma instância do `EntityTransaction` representa a conexão activa à BD através da qual realizamos operações que mudam as tabelas dessa BD:

- É iniciada pelo método `begin()` que activa uma nova transição sendo esta terminada com `commit()`, onde as alterações pedidas são efetuadas na DB
- Se se usar `rollback()` desfaz-se as alterações e deixa-se tudo como antes na BD

O método `rollback()` será usado, por exemplo, caso ocorre uma exceção.

```

try {
 em.getTransaction().begin();
 Cliente novoClient = new Cliente(...); // Client must be an entity
 em.persist(novoCliente); // object will be saved in the next commit
 em.getTransaction().commit();
} catch (Exception e) { // some unforseen problem just happened!
 if (em.getTransaction().isActive()) { // is transaction still active?
 em.getTransaction().rollback(); // if so, then undo
 }
} finally {
 em.close();
}
```

copy

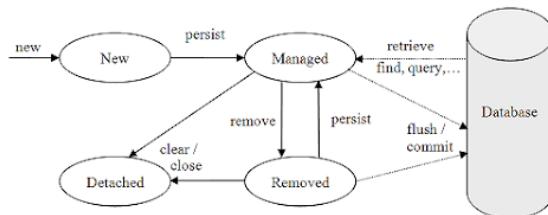
#### Transactions

Os métodos `persist`, `merge`, `remove` e `refresh` **DEVEM** ser invocados dentro de um contexto de transação. Caso contrário uma exceção de tipo `TransactionRequiredException` será lançada.

### 5.8.5 Ciclo de vida de uma Entidade

A entidade é quando criada não possui relação com o **EntityManager**

- Quando ocorre `em.persist(e)` durante uma transação ativa, a entidade passa ao **estado managed**
- Igualmente, se uma entidade é recolhida da BD (e.g., por uma query) ela também fica neste estado
- Qualquer alteração do estado da entidade é observada pelo EntityManager, e será armazenada no próximo `commit()`
- Uma entidade marcada para apagar muda de estado para **removed**, mas só será apagada da BD após o `commit` seguinte.
- Se o **EntityManager** for fechado, as entidades geridas por ele passam para o estado **detached**.
  - Existe uma forma explícita via o método `em.detach(obj)`.
  - É possível reverter, i.e., voltar a estar **managed** com `em.merge(obj)`



O **contexto de persistência** é a coleção de todos objetos geridos (**managed entities**) pelo respetivo **EntityManager** (EM). Se for pedido um objeto que já está no EM, ele é devolvido e **poupa-se uma operação** sobre a BD (funciona como cache).

- Toda esta gestão é automática e o EM certifica-se que o estado do contexto mantém-se coerente
- Pode-se verificar se um objeto pertence ao contexto usando:

```
boolean isManaged = em.contains(cliente);
```

*copy*

- Pode-se igualmente limpar o contexto com o método `clear()` neste caso, todos os objetos lá armazenados **passam ao estado detached**.

**Gestão da memória** O contexto de persistência tem memória limitada o que pode levantar problemas em transações muito grandes. Para colmatar este ponto pode-se utilizar uma combinação dos métodos `flush()` e `clear()`:

- `flush()` sincroniza o estado da BD com o contexto de persistência
- `clear()` limpa o contexto de persistência, todas as entidades **managed** passam a **detached** (sem `commit` ou `flush` prévio, não persistem na BD)

```
em.getTransaction().begin();
for (int i = 1; i <= 1000000; i++) {
 Point point = new Point(i, i);
 em.persist(point);
 if ((i % 10000) == 0) { // salva & limpa a cada 10 mil pontos
 em.flush();
 em.clear();
 }
}
em.getTransaction().commit();
```

**copy**

Uma alternativa a transações muito grandes, é a aplicação de múltiplas transações (esta é considerada uma solução melhor):

```
em.getTransaction().begin();
for (int i = 1; i <= 1000000; i++) {
 Point point = new Point(i, i);
 em.persist(point);
 if ((i % 10000) == 0) {
 em.getTransaction().commit();
 em.clear();
 em.getTransaction().begin();
 }
}
em.getTransaction().commit();
```

**copy**

## 5.8.6 Entidades JPA

Uma entidade é uma classe Java cujo estado de cada objecto irá ser armazenado como um registo na respetiva tabela. Uma classe entidade tem de cumprir o seguinte:

- Tem de ser anotada com `@Entity`

```
@Entity
public class MyClass { ... }
```

**copy**

- Tem de ter um construtor sem argumentos
- A classe não pode ser final
- Os métodos e atributos anotados também não podem ser finais
- Atributos persistentes não podem ser públicos

Os atributos como coleções, mapas e arrays podem ser persistidos.

**Entidades e chaves p.116 • Anotações de atributos p.118**

### Entidades e chaves

**Entidades e chaves:** Uma entidade tem um identificador único que irá corresponder à chave primária da respetiva tabela. Se a chave é simples (tipos primitivos ou *wrapper classes*, `String`, `java.util.Date` ou `sql.Date`) tem de ser anotada com `@Id`. Exemplo:

```
@Entity
public class Cliente {
 @Id
 private int idCliente;
```

**copy**

É possível indicar como as chaves devem ser geradas:

```
@Entity
public class Cliente {
 @Id @GeneratedValue(strategy=AUTO)
 private int idCliente;
}
```

**copy**

As diferentes estratégias de geração são:

- **AUTO** Indicates that the persistence provider should pick an appropriate strategy for the particular database.
- **IDENTITY** Indicates that the persistence provider must assign primary keys for the entity using a database identity column.
- **SEQUENCE** Indicates that the persistence provider must assign primary keys for the entity using a database sequence.
- **TABLE** Indicates that the persistence provider must assign primary keys for the entity using an underlying database table to ensure uniqueness.

Nota: as anotações estão disponibilizadas via imports ex:

```
import javax.persistence.Entity;
import javax.persistence.Id;

copy
```

O JPA permite igualmente obter a chave de uma dada entidade. Para isso, é necessário obter uma instância de um utilitário do **EntityManagerFactory**:

```
PersistenceUnitUtil util = emf.getPersistenceUnitUtil();
```

```
Object clienteId = util.getIdentifier(cliente);
```

**copy**

**Usar uma classe como chave primária.** No exemplo seguinte existe duas classes : uma classe Client e um classe ClientName. O objectivo é designar o atributo clientName da classe Client como chave primária. As anotações seguintes devem estar incluídas:

```
@Embedable
public class ClientName
 implements Serializable {
 @Column(name = "FISTNAME", nullable = false)
 private String firstname;
 @Column(name = "LASTNAME", nullable = false)
 private String lastname;
```

**copy**

```
@Entity
public class Client
 implements Serializable {

 @EmbeddedId
 protected ClientName client
```

**copy**

Exemplo:

```
ClientName clName = new ClientName("João", "Sousa");
Cliente cliente = new Cliente(clName);

em.getTransaction().begin();
em.persist(cliente);
em.getTransaction().commit();
```

**copy**

- O objeto ClientName é armazenado na tabela Client na codificação obtida após a serialização.

- Um objeto embedido não pode ser partilhado por outros registos da tabela (mesmo se não forem chaves)
- Não é necessário fazer-se o persist explícito dos objetos embebidos

É possível acrescentar uma anotação `@Table` que permite especificar o nome da tabela na base de dados:

```
@Entity
@Table(name="CLIENT")
public class Cliente {
 @Id @GeneratedValue(strategy=AUTO)
 private int idCliente;
}

copy
```

## Anotações de atributos

Um atributo sem anotação é considerado como tendo a anotação `@Basic` (anotação por defeito). Esta anotação admite uma opção `fetch` que indica ao sistema se o valor deve ser carregado de imediato (`EAGER`) ou pode ser carregado apenas quando necessário (`LAZY`). Sem opção, `EAGER` é assumido.

```
@Basic(fetch=FetchType.LAZY)
private String name;

copy
```

Para as coleções o valor por omissão é `LAZY`.

```
@Entity
class Employee {
 @ManyToMany(fetch=FetchType.EAGER)
 private Collection<Project> projects;
}

copy
```

Geralmente, cada atributo corresponderá a uma coluna na tabela.

Com a anotação `@Column` podemos dar informação específica sobre determinados atributos.

```
@Column(nullable = false, unique = true)
private int nif;
```

copy

Pode-se definir o nome da coluna (`name=...`), a sua dimensão (`length= ...`), entre outras informações.

### 5.8.7 Relações entre Entidades

O JPA define quatro tipos de multiplicidade entre as entidades A e B:

- `@OneToOne` cada instância de A está associada a uma instância de B

```
@Entity
public class Cliente {
 @OneToOne
 private Desconto desconto;

copy
```

```
@Entity
public abstract class Desconto {
 @Id
 private int tipoDesconto;

copy
```

- **@OneToMany** uma instância de A pode estar relacionada com múltiplas instâncias de B
- **@ManyToOne** múltiplas instâncias de A podem estar relacionadas com a mesma instância de B
- **@ManyToMany** múltiplas instâncias de A podem estar relacionadas com múltiplas instâncias de B

As relações podem ser uni ou bidirecionais:

- Uma relação entre as entidades A e B é **unidirecional** se **apenas uma delas tem atributos da outra**, mas não vice-versa.
- Numa relação **bidirecional** um dos lados é o “dono” (*owning side*), o outro lado é o inverso (*inverse side*).
- Numa relação **Many-To-One**, o lado **Many** é sempre o dono.
- O lado inverso usa a anotação **@mappedBy** no atributo correspondente ao dono da relação. Significa que essa informação não está na tabela inversa, mas que é preenchida com informação recolhida na tabela dono
- 

A classe **Employee** é “dona” da relação :

```
@Entity
public class Employee {
 String name;
 @ManyToOne
 Department department;
}
```

copy  
com a classe **Department**:

```
@Entity
public class Department {
 @OneToMany(mappedBy="department")
 Set<Employee> employees;
}
```

copy

### Relações entre entidades

- Numa relação 1-1 o “dono” é aquele lado que contém a chave estrangeira.
- Numa relação unidirecional há apenas o “dono” que é a entidade que tem um atributo da outra entidade

A anotação **@JoinColumn** define que uma dada coluna representa a associação das entidades, indicando que o respectivo atributo representa a chave estrangeira.

```
public class Cliente {
 @Id
 private int idCliente;
 ...
 @OneToOne
 @JoinColumn(nullable = false, name = "fkDesconto")
 private Desconto desconto;
```

copy

Neste exemplo, a coluna desconto (que será chamada `fkDesconto`) corresponde à chave estrangeira oriunda da tabela `Desconto`. Se fosse uma relação `@OneToMany` a chave estrangeira ficaria na outra classe:

```
public class Venda { ...
 @OneToMany
 @JoinColumn(nullable = false, name = "fkVenda")
 private List<VendaProduto> items;
```

copy

Se omitir os parametros (`nullable = false, name = "fkDesconto"`) a coluna será chamada `DESCONTO_ID`.

### 5.8.8 Persistir Entidades

Pode-se guardar entidades explicitamente:

```
Cliente cliente = new Cliente(...);
em.getTransaction().begin();
em.persist(cliente);
em.getTransaction().commit();
```

copy

O objecto `cliente` passa ao estado **managed** no *entity manager* `em`, e é guardado na BD após o `commit()`.

- Se o método `persist` receber um objeto que não é uma entidade é lançada a exceção `IllegalArgumentException`.
- Se já existir uma entidade com a mesma chave é lançada uma `EntityExistsException`.
- É necessário uma transação activa senão é lançada uma `IllegalStateException`.

É preciso que todos os objetos relacionados sejam guardados no `commit`, senão a operação dá erro (`IllegalStateException`). Para isso faz-se uma persistência explícita de todos esses objetos (trabalhoso e susceptível a erros) ou usa-se o mecanismos de cascata (cf. adiante).

### 5.8.9 Persistência por Cascata

Em vez de uma persistência explícita pode usar-se o mecanismo de cascata.

```
@Entity
public class Venda {
 @OneToMany(cascade = ALL)
 private List<ProdutoVenda> produtosVenda;
```

copy

Cada vez que uma venda é persistida, automaticamente todas as entidades `ProdutoVenda` relacionadas são igualmente persistidas, sem ser necessário explicitar esses comandos. Os outros valores possíveis para o atributo `cascade` são: `detach`, `merge`, `persist`, `refresh`, `remove` (que executam a respetiva operação em cascata). O valor `ALL` inclui todos os anteriores, ie, é equivalente a

```
cascade = {PERSIST, MERGE, REMOVE, REFRESH, DETACH}
```

copy

### 5.8.10 Atributos Temporários

Pode haver atributos que não sejam para armazenar. Estes incluem atributos estáticos, atributos finais ou outros atributos declarados pelo Java como *transient*. Em Java é possível declarar atributos *transient*, i.e., atributos que não são para guardar numa serialização:

```
class C implements Serializable {
 private transient int atr;
 ...
}
```

copy

A anotação JPA correspondente é:

```
@Entity
public class C {
 @Transient
 double seed;
}
```

copy

### 5.8.11 Recolher Entidades

O JPA pode ir buscar entidades à BD:

- Essas entidades até poderão estar já no contexto de persistência que, como referido, funciona como cache
- Podem ser realizadas queries usando a linguagem JPQL (cf. adiante)

Para procurar uma entidade com base a classe e chave primária (eg, cliente com id 123) deve-se usar o método **find**:

```
EntityManager em;
// ...
Cliente cliente = em.find(Cliente.class, 123);
```

copy

É devolvido o objeto da cache ou é criado um novo objeto. **Se a chave não existir, é devolvido null.**

#### Managed

A seguir a uma operação de consulta com **find**, se a entidade é encontrada, passa a estar no estado *managed*.

O método **e.getReference(...)** é uma versão lazy do **find()**. Neste caso, devolve um objeto oco (hollow object) tendo apenas inicializado o atributo chave. Tudo o resto só é carregado se houver uma consulta ao estado do objeto. Útil quando apenas se quer saber da existência da instância/registo. Pode fazer-se um **refresh** de uma entidade para sincronizar o estado do objeto em memória com aquele da BD:

```
em.refresh(client)
```

copy

Se a cascata estiver activa, o **refresh** é também realizado aos objetos referenciados. Seja o objeto preenchido ou oco, **se existir o acesso a um dos campos** (via um **get**), o conteúdo é **sempre carregado**.

Do ponto de vista do programador é como se toda a BD estivesse em memória **se as entidades não estão detached**. Caso o EntityManager for fechado, este comportamento deixa de ocorrer. Objetos ociosos já não podem ser preenchidos.

É possível verificar programaticamente se uma entidade está em memória usando o método **isLoaded**:

```
PersistenceUtil util = Persistence.getPersistenceUtil();
boolean isObjectLoaded = util.isLoaded(cliente);
boolean isFieldLoaded = util.isLoaded(cliente, "desconto");
copy
```

### 5.8.12 Atualizações

Para atualizar uma entidade na base de dados basta

1. assegurar-se que está em memória,
2. iniciar uma transação,
3. Efetuar a alteração dejeda ao estado da entidade,
4. terminar a transação:

```
Cliente cliente = em.find(Cliente.class, 4345);

em.getTransaction().begin();
cliente.setDesconto(novoDesconto);
em.getTransaction().commit();

copy
```

### 5.8.13 Remoções

O procedimento para remover uma entidade é semelhante, basta usar o método `remove`:

```
Cliente cliente = em.find(Cliente.class, 1);

em.getTransaction().begin();
em.remove(cliente);
em.getTransaction().commit();

copy
```

Os objetos embedidos são igualmente apagados. Os efeitos em cascata também funcionam (caso fazem parte da anotação):

```
@Entity
class Employee {

 @OneToOne(cascade=REMOVE) // ou cascade=ALL
 private Address address;
}

copy
```

**Remoção de Orfãos** Existe uma opção relacionada que tem a ver com a remoção de entidades que perderam todas as referências para elas:

```
@Entity
class Employee {
 @OneToOne(orphanRemoval=true)
 private Address address;
}

copy
```

Deste modo garante-se que não ficam na BD registos que não são úteis pois não podem mais ser consultados. Funciona para as relações `@OneToOne` e `@OneToMany`. Pode ser usado para coleções. Não havendo mais referências, quando um elemento é eliminado da coleção ele é eliminado da BD.

### 5.8.14 Queries

Equanto o JDBC permite definir queries com cadeias de caracteres que contêm instruções SQL, o JPA disponibiliza um meio mais sofisticado para definir queries : o Java Persistence Query Language (JPQL). O JPQL pode ser visto como uma versão orientada aos objetos do SQL.

A classe genérica **TypedQuery** é usada para definir queries. Por exemplo :

```
TypedQuery<Cliente> q =
 em.createQuery("SELECT c FROM Cliente c", Cliente.class);
```

**copy**

O objecto obtido é usado para executar a query de diversas formas:

- o método **getSingleResult()** retorna apenas um resultado :

```
TypedQuery<Long> q2 = em.createQuery("SELECT COUNT(c) FROM Cliente c",
 Long.class);
long clientCount = q2.getSingleResult();
```

**copy**

- o método **getResultList()** é usado para ir buscar vários resultados :

```
List<Cliente> results = q.getResultList();
for (Cliente c : results) {
 System.out.println(c.getName());
}
```

**copy**

Como pode ver nesses exemplos, o JPQL trata de converter os valores lidos das tabelas da base de dados para valores com tipo apropriado em Java.

#### JPQL não é SQL

As cadeias de caracteres usadas para descrever as queries podem aparecer como queries SQL mas não são. Têm uma sintaxe própria.

Neste exemplo:

```
TypedQuery<Cliente> q =
 em.createQuery("SELECT c FROM Cliente c", Cliente.class);
```

**copy**

Cliente **em** SELECT c FROM Cliente c designa o nome da tabela na base de dados e não o nome da classe (entidade). Este nome é *case-sensitive*, se o nome da tabela for CLIENTE deve usar para definir a query :

```
TypedQuery<Cliente> q =
 em.createQuery("SELECT c FROM CLIENTE c", Cliente.class);
```

**copy**

A operação de **remoção** é definida e executada da seguinte forma :

```
int count = em.createQuery("DELETE FROM Cliente").executeUpdate();
```

**copy**

apaga todos os regtos da tabela.

A atualização funciona de forma semelhante:

```
int count = em.createQuery("UPDATE Cliente SET telefone = 0").executeUpdate();
```

**copy**

Note que: É necessário que exista uma **transação activa** (senão é lançada uma exceção TransactionRequiredException).  
**Parâmetros nas queries.** As queries podem conter parâmetros. Permite a reutilização de queries: diferentes valores dos parâmetros correspondem a diferentes queries efetuadas sobre a base de dados. Um parâmetro é uma palavra prefixada pelo símbolo `:`. O método `setParameter()` define um valor para o parâmetro e devolve um objeto do tipo da query (`TypeQuery<T>`). Isto permite evadir uma chamada para executar a query (ou inicializar outro parâmetro):

```
public Cliente getCliente(EntityManager em, String name) {

 TypedQuery<Cliente> query = em.createQuery(
 "SELECT c FROM Cliente c WHERE c.name = :par",
 Cliente.class);

 return query.setParameter("par", name).getSingleResult();
}
```

**copy**

Os parâmetros podem também ser assinalados pela sua ordem:

```
public Cliente getCliente(EntityManager em, String name) {

 TypedQuery<Country> query = em.createQuery(
 "SELECT c FROM Cliente c WHERE c.name = ?1",
 Cliente.class);

 return query.setParameter(1, name).getSingleResult();
}
```

**copy**

Na query JPQL o número do parâmetro é prefixado com `?`.

**Named Queries.** Pode-se definir queries separadamente do código. Essas queries estão definidas em anotações colocadas antes da definição da classe:

```
@Entity
@NamedQuery(name="Cliente.findByVAT",
 query="SELECT c FROM Cliente c WHERE c.vat = :vat")

public class Cliente { ... }
```

**copy**

A definição tem de associar um nome à query (aqui é `Cliente.findByVAT`) e indicar a operação SQL a efetuar. Uma das vantagens é que as queries não ficam misturadas com o código.

Se uma classe tiver mais que uma `NamedQuery` é necessário fazer o seguinte:

```
@Entity
@NamedQueries({
 @NamedQuery(name="Cliente.findByVAT",
 query="SELECT c FROM Cliente c WHERE c.vat = :vat"),

 @NamedQuery(name="Cliente.findByName",
 query="SELECT c FROM Cliente c WHERE c.name = :name")
})
public class Cliente { ... }
```

**copy**

Uma vez declarada na anotação (antes da definição da classe), a query pode ser criada usando o método `createNamedQuery`:

```
TypedQuery<Cliente> query =
 em.createNamedQuery("Cliente.findByVAT", Cliente.class);
query.setParameter("VAT", 123456789);
Cliente cliente = query.getSingleResult();
copy
```

**JPQL Select.** A operação Select do JPQL é diferente do SELECT em SQL. Para selecionar colunas específicas da tabela usa-se:

```
SELECT c.name, c.vat FROM Cliente AS c
```

copy

A query vai retornar uma lista de arrays de objetos `List<Object[]>` (neste caso com dois elementos em cada array, para conter o nome e o VAT). Por exemplo:

```
TypedQuery<Object[]> query = em.createQuery(
 "SELECT c.name, c.npc FROM Country AS c",
 Object[].class);

List<Object[]> results = query.getResultList();
for (Object[] result : results) {
 System.out.println("None: " + result[0] +
 ", vat: " + result[1]);
}
copy
```

Note que deve usar o tipo `Object[].class` no segundo argumento, na criação da query.

Pode haver várias variáveis:

```
SELECT c1, c2 FROM Country c1, Country c2
 WHERE c2 MEMBER OF c1.neighbors
```

copy

Neste exemplo a entidade `Country` teria um atributo coleção designado `neighbors` de tipo `Collection<Country>` com todos os países que um dado país faz fronteira. Este select devolveria todos os pares de países com fronteira entre si. São efetuados dois ciclos (um com `c1`, e outro interno com `c2`) sobre todos os países da tabela `Country`.

Outra opção, mais eficiente seria realizar um *inner join*:

```
SELECT c1, c2 FROM Country c1 JOIN c1.neighbors c2
```

Neste caso o segundo ciclo (da variável `c2`) seria declarada no contexto mais limitado dos vizinhos do atual valor `c1`.

Assumindo que cada país tem apenas uma capital, as seguintes instruções são equivalentes:

```
SELECT c.name, p.name FROM Country c JOIN c.capital p
SELECT c.name, c.capital.name FROM Country c
```

No primeiro exemplo, a variável `p` fica associada ao valor (único) da capital do país actual.

No exemplo seguinte são devolvidos pares (país, língua) onde a população seja maior que o parâmetro `:p`, e pelo menos uma das linguagens oficiais pertença ao parâmetro `:languages` (cujo tipo é uma coleção Java):

```
SELECT c, l FROM Country c JOIN c.languages l
 WHERE c.population > :p AND l IN :languages
```

No seguinte devolvemos pares de moeda com o total de população dos países europeus que a usam, incluindo apenas moedas usadas em mais do que um país:

```
SELECT c.currency, SUM(c.population)
 FROM Country c
 WHERE 'Europe' MEMBER OF c.continents
 GROUP BY c.currency
 HAVING COUNT(c) > 1
```

A query seguinte devolve os nomes dos países com mais de 1000000 pessoas ordenados pela sua população por ordem crescente (e se têm a mesma população, ordenados pelo nome por ordem decrescente):

```
SELECT c.name
 FROM Country c
 WHERE c.population > 1000000
 ORDER BY c.population ASC, c.name DESC
```

**JPQL Delete.** Exemplo de remoção usando o JPQL:

```
int deletedCount = em.createQuery("DELETE FROM Country c").executeUpdate();
```

copy

apaga todos os registos da tabela.

Exemplo de remoção seletiva:

```
query = em.createQuery(
 "DELETE FROM Country c WHERE c.population < :p");
```

```
int deletedCount =
 query.setParameter("p", 100000).executeUpdate();
```

copy

apaga os países com menos de 100000 pessoas.

**JPQL Update.** Comandos que alteram a base de dados:

```
query = em.createQuery(
 "UPDATE Country c SET c.population = 0, c.area = 0");
int updateCount = query.executeUpdate();
```

copy

coloca a zero a população e área de todos os países. Exemplo de modificação seletiva:

```
query = em.createQuery(
 "UPDATE Country SET population = population * 1.1 " +
 "WHERE c.population < :p");
```

```
int updateCount = query.setParameter("p", 100000).executeUpdate();
```

copy

aumenta de 10% as populações dos países com meno de 100000 habitantes.

## 5.9 Exercícios

Row Data Gateway p.[126](#) • BoardGamesCafé p.[126](#) • Quiz p.[127](#)

### 5.9.1 Row Data Gateway

Explique em que consiste o padrão *Row Data Gateway* focando a sua explicação na forma como é feita a abstração da base de dados e nas responsabilidades atribuídas a cada objeto do padrão.

### 5.9.2 BoardGamesCafé

Pretende-se desenvolver uma aplicação **BoardGamesCafé** que ajude a gestão das atividades de um tipo de negócio que se está a tornar popular: lojas onde se pode jogar jogos de tabuleiro e em que se paga em função dos jogos usados e o tempo de utilização dos mesmos.

Para cada grupo de clientes que se senta numa mesa livre é criado um registo de ocupação da mesa, que regista obrigatoriamente o nome e telefone do elemento responsável pelo grupo e o número de ocupantes. Esse registo serve ainda para registrar os jogos requisitados pelo grupo. Um jogo que esteja disponível pode ser requisitado se o grupo que ocupa a mesa não tiver nenhum jogo requisitado.

Para cada jogo requisitado, é registada a hora de requisição e devolução e a classificação atribuída (0 a 5 estrelas). Na saída (check-out) é calculado o total a pagar e registada a forma como foi feito o pagamento. Cada jogo tem um código único e um nome (obrigatórios), uma descrição e pode estar em um de três estados: livre, ocupado e manutenção. Para ajudar os clientes a escolher, a aplicação permitirá ver a popularidade dos jogos (número de requisições) e a classificação média.

### 1. Modelo de dados

Suponha que foi decidido usar uma base de dados relacional para persistir todos os dados da aplicação **BoardGamesCafé**. Apresente um modelo de dados apropriado incluindo tabelas **RegistoOcupacao** e **Requisicao** para registar a ocupação das mesas e as requisições dos jogos.

### 2. Organização da camada de negócio: Domain Model

Suponha que se pretende desenhar a camada de negócio do **BoardGamesCafé** de acordo com o padrão Domain Model, ou seja, pretende-se organizar o código desta camada com classes que são inspiradas pelos conceitos do domínio, nas suas características e associações.

- Elabore um modelo de domínio a fim de identificar os principais conceitos do sistema.
- Aplicando os padrões **GRASP**, elabore um diagrama de interação que defina o comportamento da operação **requisitarJogo** que trata do registo de uma reserva de um jogo com um determinado código pelos ocupantes de uma mesa com um determinado número.
- Repita o que fez anteriormente para as operações:
  - checkin
  - devolverJogo
  - checkout
  - obterMesasDisponiveis
  - obterInfoJogo
- Desenha um diagrama de classes de acordo com as decisões tomadas.

### 3. Organização da camada de negócio: Transaction Script

Supondo que foi decidido usar o padrão Transaction Script para organizar o código da camada do domínio e o padrão Row Data Gateway para a camada de acesso aos dados, esboce uma solução que suporte a operação:

```
void requisitarJogo(int numMesa, int codigoJogo)
```

copy

para requisição de um novo jogo pelos ocupantes de uma mesa. Considere ainda que na classe onde define o script existem os métodos:

```
boolean jogoEstaLivre(JogoRDGW jogo)
boolean temJogoRequisitado(MesaRDGW mesa)
```

copy

### 5.9.3 Quiz

1. Qual padrão de desenho envolve a criação de classes de domínio que representam objetos do mundo real e suas interações, encapsulando a lógica de negócios ?
2. Qual padrão de desenho é comumente usado em sistemas onde a interação direta com registros da base de dados é necessária, representando cada registo como uma classe separada?
3. Qual padrão de desenho se concentra na separação das classes de domínio da base de dados, usando uma classe intermediária para mapear objetos para registo da base de dados e vice-versa?
4. Qual padrão de desenho é frequentemente implementado usando ORMs (Object-Relational Mapping) para automatizar o mapeamento entre objetos de domínio e tabelas da base de dados?
5. Qual padrão de desenho é especialmente útil em sistemas complexos onde a lógica de negócios e os requisitos de persistência são extensos, promovendo a separação de responsabilidades?

1. Qual padrão de design é usado para encapsular a lógica de negócios e suas interações com objetos de domínio?
  - (a) Active Record
  - (b) Table Data Gateway
  - (c) Data Mapper
  - (d) Domain Model
  
2. Qual padrão de desenho é adequado para sistemas onde a interação direta com registo da base de dados é necessária e representa cada registo como uma classe separada ?
  - (a) Active Record
  - (b) Data Mapper
  - (c) Repository
  - (d) Table Data Gateway
  
3. Qual padrão de desenho se concentra na separação das classes de domínio da base de dados, usando uma classe intermediária para mapear objetos para os registo da base de dados e vice-versa?
  - (a) Active Object
  - (b) Table Module
  - (c) Domain Model
  - (d) Data Mapper
  
4. Qual padrão de desenho é frequentemente implementado usando ORMs (*Object-Relational Mapping*) para automatizar o mapeamento entre objetos de domínio e tabelas da base de dados?
  - (a) Repository
  - (b) Table Module
  - (c) Data Mapper
  - (d) Active Object
  - (e)
  
5. Qual padrão de desenho é particularmente útil em sistemas complexos onde a lógica de negócios e os requisitos de persistência são extensos, promovendo a separação de responsabilidades?
  - (a) Domain Model
  - (b) Domain Model
  - (c) Table Data Gateway
  - (d) Active Object

## 5.10 Guião JDBC & Transaction Script

Esqueleto do projeto p.129 • DataSource p.129 • RunSQLScript p.131  
 • Configuração do pom.xml p.131 • CreateDatabase p.132 • Correr o programa com Maven p.132 • Inserir uma linha na tabela p.133 • Visualizar a BD no eclipse p.133 • JogadorRDGW p.135 • MesaRDGW p.136  
 • CheckinTransactionScript p.137 • Classe Menu p.138 • Acabou ! p.142  
 • Problemas e soluções p.142

### 5.10.1 Esqueleto do projeto

Começar por criar um repositório numa pasta nova. Associar o repositório chamado `boardgamecafe` à sua conta no servidor `git.alunos.di.fc.ul.pt`.

#### Cuidado com o nome do repositório !

Em caso de engano pode mudar o nome e o endereço do repositório no gitlab: Selecionar o projeto, **Settings > General > Advanced > Expand > Change path** > altere a parte final do URL para `boardgamecafe`. No topo da mesma página corrige igualmente o nome do projeto para `boardgamecafe`. Não se esquece modificar o endereço do repositório remoto no seu computador:

```
git remote remove origin
git remote add ``novo url``
```

- Criar um projeto de tipo Maven chamado `BoardGameCafe` na pasta do seu repositório.
- Criar os pacotes `main`, `persist`, `scripts` e `dbutils`.
- Criar uma classe `BoardGameCafe` no pacote `main`.
- Nesta classe definir uma constante que com o endereço da base de dados:

```
public static final String DB_CONNECTION_STRING =
 "jdbc:derby:data/derby/boardgamecafedb";
```

[copy](#)

#### Caso o maven não esteja instalado no seu computador

Pode o instalar facilmente na sua área. Faça download deste [link](#) por exemplo na sua pasta “Download”. Extrai o conteúdo do arquivo. Pode a seguir criar um alias para o executável do maven (é o ficheiro `mvn` que se encontra na pasta `bin` do arquivo.)

### 5.10.2 DataSource

A classe `Datasource` fornece um nível de abstração para uma conexão à base de dados.

```
package persist;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.sql.Statement;

/**
 * The class DataSource is an abstraction for a database connection.
 *
 */
public class DataSource {
 private static Connection connection;
 /**
 * Constructs a database connection given the connection url, the username, and its passw

```

```

* @param url The database connection URL
* @param username The username to login into the database
* @param password The user's password
* @return The data source.
* @throws PersistenceException In case the connection fails to establish
*/
public static void connect(String url, String username, String password)
 throws PersistenceException {
 try {
 connection = DriverManager.getConnection(url, username, password);
 } catch (SQLException e) {
 System.err.println(e.getMessage());
 e.printStackTrace();
 System.err.println("/** Just in case, is your eclipse connected to the DB ?");
 throw new PersistenceException("Cannot connect to database", e);
 }
}

/**
* @return The current database connection
*/
public static Connection getConnection() {
 return connection;
}

public static PreparedStatement prepareGetGenKey(String sql) throws SQLException {
 System.out.println("connection: " + connection);
 return connection.prepareStatement(sql, Statement.RETURN_GENERATED_KEYS);
}
/**
* Prepare an SQL statement from an SQL string
*
* @param sql The SQL text to prepare the command
* @return The prepared statement for the SQL text
* @throws PersistenceException In case the prepare statement
* encounters an error.
*/
public static PreparedStatement prepare(String sql) throws PersistenceException {
 try {
 return connection.prepareStatement(sql);
 } catch (SQLException e) {
 throw new PersistenceException("Error preparing comment", e);
 }
}

/**
* Close the database connection
*/
public static void close() throws PersistenceException {
 try {
 connection.close();
 } catch (SQLException e) {
 throw new PersistenceException("Cannot closing the database", e);
 }
}
}

```

⇒ [DataSource.java](#)

Tem também que definir uma classe **PersistenceException** no mesmo pacote. Para a definição desta

classe pode seguir as sugestões do eclipse quando aparece um erro relacionado com a mesma.

### 5.10.3 RunSQLScript

No pacote `dbutils` vamos colocar alguns utilitários para, por exemplo, criar a base de dados. Uma classe se destaca : `RunSQLScript`: é um utilitário que permite correr scripts escritos em SQL puro. Atenção ! o programa assume que os comandos SQL estão escritos numa linha. Não se pode quebrar um comando em várias linhas.

```
package dbutils;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.sql.Connection;
import java.sql.SQLException;
import java.sql.Statement;

public class RunSQLScript {

 /**
 * An utility class should not have public constructors
 */
 private RunSQLScript() {
 }

 public static void runScript(Connection connection, String scriptFilename) throws IOException {
 try (BufferedReader br = new BufferedReader(new FileReader(scriptFilename))) {
 String command;
 int i = 1;
 while ((command = br.readLine()) != null) {
 System.out.println(i + ":" + command);
 i++;
 try (Statement statement = connection.createStatement()) {
 statement.execute(command);
 }
 }
 }
 }
}
```

⇒ RunSQLScript.java

### 5.10.4 Configuração do pom.xml

Neste guião vamos usar uma base de dados “Derby” local. Não necessita aceder a um servidor de base de dados externo. Para isso precisamos acrescentar a seguinte dependência ao ficheiro `pom.xml` :

```
<dependencies>
 <dependency>
 <groupId>org.apache.derby</groupId>
 <artifactId>derby</artifactId>
 <version>10.12.1.1</version>
 </dependency>
</dependencies>
```

copy

### 5.10.5 CreateDatabase

A classe `CreateDatabase` (no pacote `dbutils`) corre um script SQL para criar a base de dados.

```
package dbutils;

import java.io.IOException;
import java.sql.SQLException;

import main.BoardGameCafe;
import persist.DataSource;
import persist.PersistenceException;

public class CreateDatabase {

 public void createCSSDerbyDB() throws IOException, SQLException, PersistenceException {
 DataSource.connect(BoardGameCafe.DB_CONNECTION_STRING + ";create=true", "BoardGameCafe");
 RunSQLScript.runScript(DataSource.getConnection(), "data/scripts/createDB-Derby.sql");
 DataSource.close();
 }

 public static void main(String[] args) throws PersistenceException, IOException, SQLException {
 new CreateDatabase().createCSSDerbyDB();
 }
}
```

⇒ `CreateDatabase.java`

É ainda necessário escrever o script SQL correspondente (`createDB-Derby.sql`), na pasta `data/scripts/`:

```
CREATE TABLE PLAYERS (PHONE_NUMBER INTEGER PRIMARY KEY NOT NULL, NAME VARCHAR(50) NOT NULL)
```

Nesta fase apenas uma tabela está criada. Fica ao seu cargo acrescentar aqui as instruções para criar todas as tabelas necessárias.

### 5.10.6 Correr o programa com Maven

Para correr o programa com o Maven é necessário acrescentar ao ficheiro `pom.xml`:

```
<build>
 <plugins>
 <plugin>
 <artifactId>maven-compiler-plugin</artifactId>
 <version>3.7.0</version>
 <configuration>
 <source>1.8</source>
 <target>1.8</target>
 </configuration>
 </plugin>
 </plugins>
</build>
```

copy

Caso esteja a usar uma versão do java mais posterior à 1.8 tem de substituir, por exemplo, 1.8 por 11 (se estiver a usar a versão 11).

O programa está pronto para ser executado:

1. mvn clean

este passo limpa a pasta target removendo os ficheiros criados por compilações ou utilizações anteriores do programa. Não é um passo obrigatorio.

2. mvn compile

compila o programa. É uma boa maneira de verificar que não há erros obvios no código e se todas as bibliotecas estão acessíveis. É necessário o computador estar ligado a internet neste passo. O Maven faz automaticamente *download* das bibliotecas especificadas no ficheiro `pom.xml`.

3. mvn exec:java -Dexec.mainClass=dbutils.CreateDatabase -Dexec.cleanupDaemonThreads=false

Este comando executa o programa. O segundo parâmetro especifica a classe que contém o método `main` que desejamos correr. Note que deve indicar o pacote onde esta classe está. Deste modo pode ter no projeto várias classes com um método `main` e corre-las desta maneira.

A execução deste comando deve resultar num sucesso, as últimas linhas a aparecer no terminal são:

```
1: CREATE TABLE PLAYER (PHONE_NUMBER INTEGER PRIMARY KEY NOT NULL, NAME VARCHAR(50) NOT NULL)
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
```

Note que se correr uma segunda vez o programa vai dar um erro porque a base de dados já foi criada.

### 5.10.7 Inserir uma linha na tabela

Com o objetivo de testar o acesso a base de dados criada, neste passo é preciso:

- Criar o script SQL `initDB-Derby.sql` (na pasta `data/scripts`) que contém instruções SQL para a inserção de algumas linhas na única tabela que a BD contém até agora.
- Definir uma classe `InitDatabase` no pacote `dbutils`, semelhante à classe `CreateDatabase` que exucuta o script anterior.
- Teste o seu programa. Deve poder correr o programa no terminal com o comando :

```
mvn exec:java -Dexec.mainClass=dbutils.InitDatabase -Dexec.cleanupDaemonThreads=false
```

- Para não ter que teclar um comando tão comprido pode criar um alias:

```
alias initdb="mvn exec:java -Dexec.mainClass=dbutils.InitDatabase -Dexec.cleanupDaemonThreads=false"
```

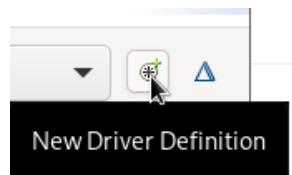
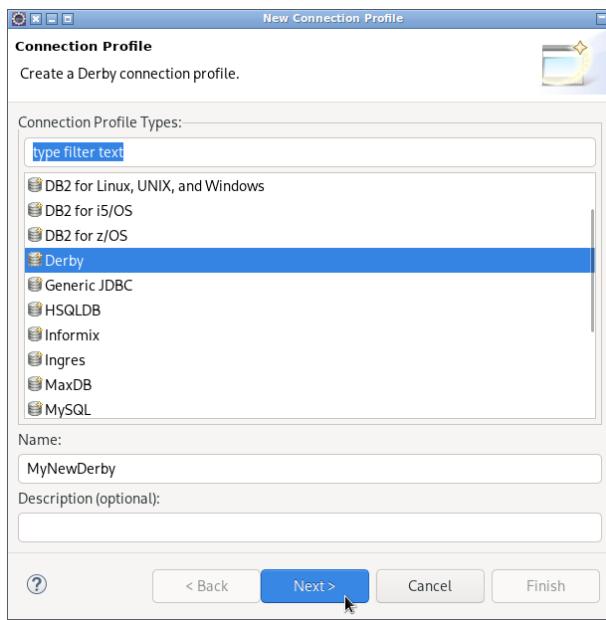
Uma vez criado o alias, para correr o programa, basta usar:

```
initdb
```

Para que este alias fique guardado nas sessões futuras deve acrescentar a linha ao seu ficheiro `.bashrc` na pasta raíz da sua área.

### 5.10.8 Visualizar a BD no eclipse

O eclipse fornece interfaces para interagir com as bases de dados. Para usa-la deve ecolher a opção **Window > Show View > Other ... > Data Management > Data Source Explorer** > Na janela do **Data Source Explorer**, clicar em **Database Connections** e escolher **New**. Na janela seguinte escolher como **Connection Profile Type** “Derby”, escolher um nome para a conexão e clicar em **Next**.



Na janela seguinte clicar no elemento mostrado na imagem seguinte para escolher o driver. Selecione a versão mais recente de “Derby **Embedded** JDBC Driver” e de seguida clicar em “Jar List”. Este passo serve para indicar ao eclipse a biblioteca JDBC que deve ser usada para esta base de dados. Felizmente usou primeiro o Maven que tratou de fazer download das bibliotecas necessárias. Basta seleciona-la. O Maven guarda as bibliotecas numa pasta escondida chamada .m2, na raíz da sua área. Clicar em “Add Jar/Zip”. O ficheiro que deve selecionar encontra-se aqui:

```
~/.m2/repository/org/apache/derby/derby/10.12.1.1/derby-10.12.1.1.jar
```

onde o ~ representa a raíz da sua área. Clicar em “OK”

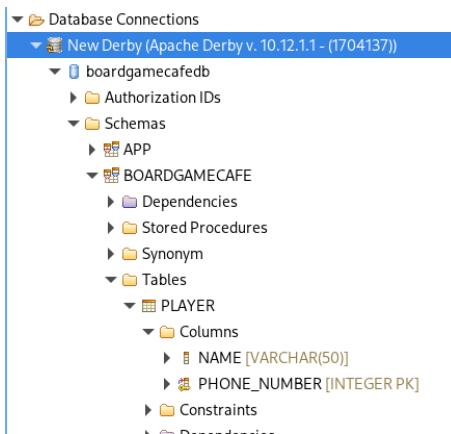
Na janela mostrada a seguir, indique onde é gravada a sua base de dados (Database location). Para condizer com a definição colocada na classe BoardGameCafe, escolhe a pasta **data/derby/boardgamecafedb**.



Agora deve aparecer a nova conexão na janela Data Source Explorer



Clicando na conexão com o botão direito do rato escolhe a opção “Ping” para testar a acesso à BD. Deve obter uma mensagem “Ping succeeded”. A seguir establece a ligação escolhendo a opção “connect”. Pode agora aproveitar o momento para admirar a beleza incomensurável da sua base de dados:



Clicando com o botão direito na tabela **PLAYER** pode escolher **Data > Edit** > para visulizar o seu conteúdo.

Note que este passo serve apenas para poder aceder à base de dados via o eclipse. Não é um passo necessário para a execução do programa.

### 5.10.9 JogadorRDGW

Nos passos seguintes vai ser implementado o Transaction Script “**CheckInTransactionScript**” que trata de reservar uma mesa para um jogador. Os scripts vão ser implementados recorrendo ao padrão **Row Data Gateway** para a camada de persistência.

Para este script será necessário usar uma classe **JogadorRDGW** e uma classe **MesaRDGW**. Nesta secção vamos tratar da primeira classe.

- No pacote persist cirar a classe **JogadorRDGW**. Os atributos da classe são o nome (String) e o número de tel (int).
- Criar um construtor e getters. Não será necessário criar setters numa primeira fase dado que as pessoas mudam raramente de nome e muito menos de número de telephone.
- Defina constantes para o nome da tabela e os nomes das colunas :

```
/**
 * Table name
 */
private static final String TABLE_NAME = "PLAYERS";

/**
 * Field names
 */
private static final String NAME_COLUMN_NAME = "NAME";
private static final String PHONE_NUMBER_COLUMN_NAME = "PHONE_NUMBER";
```

copy

- Defina constantes para as queries *insert* e *find*:

```
/**
 * A query for insertion
 */
private static final String INSERT_PLAYER_SQL =
 "INSERT INTO " + TABLE_NAME + " " +
 "(" + PHONE_NUMBER_COLUMN_NAME +
 ", " + NAME_COLUMN_NAME + ") " +
 "VALUES (?, ?);"

/**
```

```

 * A query to find a player by his phonenumber
 */
private static final String GET_PLAYER_BY_PNONE_NUMBER_SQL =
 "SELECT " + PHONE_NUMBER_COLUMN_NAME + ", " +
 NAME_COLUMN_NAME + " " +
 "FROM " + TABLE_NAME + " " +
 "WHERE " + PHONE_NUMBER_COLUMN_NAME + " = ?";

```

**copy**

- Defina o método `insert`:

```

public void insert () throws PersistenceException {
 try (PreparedStatement statement =
 DataSource.prepareStatement(INSERT_PLAYER_SQL)) {
 // set statement arguments
 statement.setInt(1, tel);
 statement.setString(2, nome);
 // executes SQL
 statement.executeUpdate();
 } catch (SQLException e) {
 System.err.println("Find error: " + e.getMessage());
 throw new PersistenceException ("Internal error!", e);
 }
}

```

**copy**

- Defina o método `find` cuja assinatura é:

```
public static JogadorRDGW find (int id)
 throws PersistenceException, RecordNotFoundException {
```

**copy**

- Defina o método `exists` com a seguinte assinatura:

```
public static boolean exists(int id) throws PersistenceException
```

**copy**

que testa a existência de um jogador na base de dados (sem criar uma instância).

- Finalmente escreva uma classe `TestJogadorRDGW` no pacote `dbutils`, com um método `main` que verifique o funcionamento da classe `JogadorRDGW`.

## 5.10.10 MesaRDGW

A etapa seguinte consiste em repetir os passos anteriores para escrever a classe `MesaRDGW`.

- Defina os atributos da classe (para representar o número da mesa, o número de ocupantes e o jogador)
- Defina a constante que representa o nome da tabela.
- Defina constantes para os nomes das colunas da tabela
- Defina constantes para as queries `insert`, `find` e `update`. Esta última deve permitir atualizar o jogador e o número de ocupantes de uma mesa na tabela.
- Escreva o construtor (só aceita o número da mesa como parâmetro), os `getters` e `setters` (para os atributos jogador e número de ocupantes) e um método `toString`.

## 6. Defina o método

```
public boolean isFree()
```

**copy**  
que retorna `true` se a mesa estiver livre e `false` caso contrário.

## 7. Implemente a seguir os métodos :

- `public void insert () throws PersistenceException`  
**copy**
- `public void update() throws PersistenceException`  
**copy**
- `public static MesaRDGW find (int id) throws PersistenceException, RecordNotFoundException`  
**copy**

## 5.10.11 CheckinTransactionScript

Nesta fase, a escrita do *transaction script* torna-se simples. A classe `CheckInTransactionScript`, colocada no pacote scripts contém um método

```
public void checkIn(int phoneNumber, int tableNumber) throws ApplicationException
```

**copy**  
responsável por executar o script. Os passos foram identificados no desenvolvimento do diagrama de interação :

- Obter o jogador J cujo número de telephone é fornecido como argumento do método. Se não existir inseri-lo na base de dados.

```
try {
 j = JogadorRDGW.find(phoneNumber);
} catch (RecordNotFoundException e) {
 // The problem of this approach is that the player inserted in
 // the DB has no name.
 System.err.println("Player " + phoneNumber + " not found");
 j = new JogadorRDGW(phoneNumber);
 j.insert();
}
```

**copy**

Como indicado em comentário, esta abordagem tem o defeito criar um elemento na tabela de jogadores que não tem nome. Tem alguma sugestão para melhorar esta situação ?

- Obter a mesa cujo número está indicado em argumento.
- Verificar se a mesa está livre.
- Atualizar a mesa de forma a ter como jogador associado o jogador J e o número de ocupantes (>1).
- Atualizar a tabela das mesas.

### 5.10.12 Classe Menu

A classe `Menu` (fornecida) apresenta um menu ao utilizador para executar os scripts da aplicação. A classe é usada pela classe principal `BoardGameCafe`, igualmente fornecida (ver mais abaixo).

```
package main;

import java.sql.SQLException;
import java.util.Scanner;

import persist.PersistenceException;
import scripts.CheckInTransactionScript;

public class Menu {

 private static CheckInTransactionScript checkIn = new CheckInTransactionScript();
 static void mainMenu(Scanner in) {
 boolean end = false;
 do {
 System.out.println("Choose an option: ");
 System.out.println("Check in.....1");
 System.out.println("Requisitar Jogo.....2");
 System.out.println("List DB (for debug)3");
 System.out.println("Exit.....4");
 System.out.println("> ");
 switch (nextInt(in)) {
 case 1:
 checkIn(in);
 break;
 case 2:
 requisitarJogo();
 break;
 case 3:
 listDB();
 break;
 case 4:
 end = true;
 break;
 }
 } while (!end);
 }

 private static void requisitarJogo() {
 }

 private static void listDB() {
 try {
 dbutils.ListDatabase.doListDB();
 } catch (PersistenceException e) {
 System.err.println(e.getMessage());
 e.printStackTrace();
 } catch (SQLException e) {
 System.err.println(e.getMessage());
 e.printStackTrace();
 }
 }

 private static void checkIn(Scanner in) {
 try {

```

```

 System.out.println("Phone number: ");
 int phoneNumber = nextInt(in);
 if (!checkIn.exists(phoneNumber)) {
 System.out.println("Your name ?");
 String name = nextLine(in);
 checkIn.createPlayer(phoneNumber, name);
 }
 System.out.println("Table number: ");
 int tableNumber = nextInt(in);
 checkIn.checkIn(phoneNumber, tableNumber);
 } catch (ApplicationException e) {
 System.err.println("Application error while checkIn.");
 System.err.println(e.getMessage());
 e.printStackTrace();
 }
}

/*
 * The following methods read several kinds of values from a Scanner.
 * The Scanner may correspond to System.in or to an input file. This allows
 * automatic testing of the application through "use cases" that are tested
 * using the executeUseCase method in the App class.
 * The reason for using these methods instead of Scanner's nextXXX() methods
 * is they allow comments in the use case files. Comments are begin with #
 * and end at the end of the line.
 *
 */
private static int nextInt(Scanner in) {
 String s = in.nextLine();
 while (s.startsWith("#")) {
 s = in.nextLine();
 }
 if (s.contains("#")) {
 try (Scanner sc = new Scanner(s)) {
 s = sc.nextInt();
 }
 }
 int value = 0;
 try {
 value = Integer.parseInt(s);
 } catch (NumberFormatException e) {
 System.out.println("> ");
 return nextInt(in);
 }
 return value;
}

private static double nextDouble(Scanner in) {
 String s = in.nextLine();
 while (s.startsWith("#")) {
 s = in.nextLine();
 }
 if (s.contains("#")) {
 try (Scanner sc = new Scanner(s)) {
 s = sc.nextDouble();
 }
 }
 return Double.parseDouble(s);
}

```

```

private static String nextLine(Scanner in) {
 String s = in.nextLine();
 while (s.startsWith("#")) {
 s = in.nextLine();
 }
 if (s.contains("#")) {
 int p = s.indexOf("#");
 s = s.substring(0, p).trim();
 }
 return s;
}

private static int[] nextDate(Scanner in) {
 String s = in.nextLine();
 while (s.startsWith("#")) {
 s = in.nextLine();
 }
 if (s.contains("#")) {
 int p = s.indexOf("#");
 s = s.substring(0, p).trim();
 }
 String[] a = s.split("/");
 int[] d = new int[3];
 d[0] = Integer.parseInt(a[0]);
 d[1] = Integer.parseInt(a[1]);
 d[2] = Integer.parseInt(a[2]);
 return d;
}

private static int[] nextTime(Scanner in) {
 String s = in.nextLine();
 while (s.startsWith("#")) {
 s = in.nextLine();
 }
 if (s.contains("#")) {
 int p = s.indexOf("#");
 s = s.substring(0, p).trim();
 }
 String[] a = s.split(":");
 int[] time = new int[2];
 time[0] = Integer.parseInt(a[0]);
 time[1] = Integer.parseInt(a[1]);
 return time;
}

```

⇒ Menu.java

```

package main;

import java.util.Scanner;

import persist.DataSource;
import persist.PersistenceException;

public class BoardGameCafe {

```

```

public static final String DB_CONNECTION_STRING = "jdbc:derby:data/derby/boardgamecafedb";
public static void main(String[] args) throws ApplicationException {
 try {
 DataSource.connect(DB_CONNECTION_STRING + ";create=false", "BoardGameCafe", "");
 Menu.mainMenu(new Scanner(System.in));
 } catch (PersistenceException e) {
 throw new ApplicationException("Error connecting database", e);
 }
}
}

```

⇒ BoardGameCafe.java

A terceira opção do menu pode ser usada para visualizar o conteúdo da base de dados. A classe `ListDataBase` está fornecida mas tem de verificar se os nomes da tabelas/colunas corresponde aos nomes escolhidos por si.

```

package dbutils;

import java.sql.SQLException;

import main.BoardGameCafe;

import java.sql.PreparedStatement;
import java.sql.ResultSet;

import persist.DataSource;
import persist.PersistenceException;

public class ListDatabase {

 public static void doListDB() throws PersistenceException, SQLException {
 DataSource.connect(BoardGameCafe.DB_CONNECTION_STRING,
 "BoardGameCafe", "");
 PreparedStatement statement = DataSource.prepare("SELECT * FROM PLAYERS");
 ResultSet rs = statement.executeQuery();
 System.out.println("++++++ Players ++++++");
 while (rs.next()) {
 String name = rs.getString("NAME");
 int tel = rs.getInt("PHONE_NUMBER");
 System.out.println("Name: " + name + " Phone: " + tel);
 }
 statement.close();
 rs.close();
 System.out.println("++++++ Tables ++++++");
 statement = DataSource.prepare("SELECT * FROM TABLES");
 rs = statement.executeQuery();
 while (rs.next()) {
 String num = rs.getString("NUM");
 int p = rs.getInt("PLAYER");
 int n = rs.getInt("N_OCCUPANTS");
 System.out.println("Table num: " + num + " Player: " +
 p + " N_OCCUPANTS " + n);
 }
 statement.close();
 rs.close();
 }
}

```

```

public static void main(String[] args) throws PersistenceException, SQLEception {
 doListDB();
 DataSource.close();
}
}

```

⇒ [ListDatabase.java](#)

### 5.10.13 Acabou !

Uma vez terminado este guião, deve dar acesso ao seu repositório ao utilizador `css-lti-000` com as permissões “Developper”.

### 5.10.14 Problemas e soluções

Target option 1.5 is no longer supported p.[142](#) • GitLab: You are not allowed to push code to protected branches on this project p.[142](#) • O eclipse modifica a sua área de trabalho sem avisar p.[143](#) • Como criar aliases p.[143](#) • O eclipse parece loco p.[143](#)

#### Target option 1.5 is no longer supported

Se obtiver o erro do tipo:

```
[ERROR] Source option 5 is no longer supported. Use 6 or later.
[ERROR] Target option 1.5 is no longer supported. Use 1.6 or later.
```

pode ser resolvido adicionando ao ficheiro pom.xls:

```

<properties>
 <maven.compiler.source>11</maven.compiler.source>
 <maven.compiler.target>11</maven.compiler.target>
</properties>
```

[copy](#)

para passar a usar o java 11 ou :

```

<properties>
 <maven.compiler.source>1.8</maven.compiler.source>
 <maven.compiler.target>1.8</maven.compiler.target>
</properties>
```

[copy](#)

para passar a usar o java 8.

#### GitLab: You are not allowed to push code to protected branches on this project

Por omissão, o git lab “protege” o ramo master. Para além de adicionar os seus colegas aos membros do seu repositório enquanto “Developpers” tem de configurar o repositório de forma a permitir os colegas alterar o master.

Na página “Settings” do seu repositório escolher **Repository > Protected Branches** > aparece a lista dos ramos protegidos. Escolhe a opção “unprotect” para o ramo master. Agora os seus colegas podem fazer push para o master.

## O eclipse modifica a sua área de trabalho sem avisar

O eclipse altera sem aviso alguns ficheiros de configuração como o ficheiro `.classpath` ou ficheiros na pasta `.settings`. Pode acontecer que se esqueça de incluir esses ficheiros num commit e posteriormente esta situação venha a provocar a rejeição de um merge (por exemplo) :

```
Auto-merging .settings/org.eclipse.jdt.core.prefs
CONFLICT (content): Merge conflict in .settings/org.eclipse.jdt.core.prefs
Auto-merging .classpath
CONFLICT (content): Merge conflict in .classpath
Automatic merge failed; fix conflicts and then commit the result.
```

Nesta altura os ficheiros foram modificados (pelo git) de forma a assinalar as áreas de conflito. **É necessário resolver a situação manualmente editando os ficheiros em causa.** Se deixar os ficheiros na sua versão alterada pelo git, os ficheiros de configuração estão corruptos e nada de bom vai acontecer pela frente. Se está a hesitar e não sabe como reslover os conflitos, fique com uma cópia dos originais numa pasta separada.

## Como criar aliases

Ningém gosta de teclar comandos compridos como por exemplo :

```
mvn exec:java -Dexec.mainClass=dbutils.CreateDatabase -Dexec.cleanupDaemonThreads=false
```

Para evitar este trabalho, pode criar alias para os comandos mais usados.

Por exemplo, pode colocar num ficheiro `aliases.sh` as linhas seguintes:

```
alias testj="mvn exec:java -Dexec.mainClass=dbutils.TestJogadorRDGW -Dexec.cleanupDaemonThreads=false
alias createdb="mvn exec:java -Dexec.mainClass=dbutils.CreateDatabase -Dexec.cleanupDaemonThreads=false
alias initdb="mvn exec:java -Dexec.mainClass=dbutils.InitDatabase -Dexec.cleanupDaemonThreads=false
alias resetdb="mvn exec:java -Dexec.mainClass=dbutils.ResetTables -Dexec.cleanupDaemonThreads=false
alias deletedb="mvn exec:java -Dexec.mainClass=dbutils.DeleteAllTables -Dexec.cleanupDaemonThreads=false
```

Para tornar efetivo essas definições, basta correr o comando

```
source aliases.sh
```

no terminal.

## O eclipse parece loco

O eclipse não está a perceber os seus imports assinala a vermelho classes que estão a ser importadas. Para resolver o problema pode :

- Clicar com o botão direito no projeto, escolher a opção **Maven > Update project >**.
- Se a sugestão anterior não resultou, faça *Delete* do projeto e volte a imporarar (como projeto Maven)

# 5.11 Guião JPA

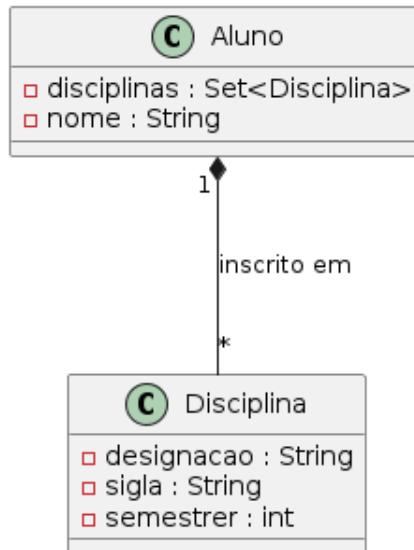
Este guião tem como objetivo a realização de um primeiro projeto usando a tecnologia JPA.

## 5.11.1 Criação do repositório

Criar um repositório vazio chamado **GuiaoJPA** no site `git.alunos.di.fc.ul.pt`. Clonar este repositório no seu computador. Criar na pasta do repositório um ficheiro `README.md` onde colocará uma descrição do trabalho realizado. Nesta fase pode colocar o seu nome e número de aluno. Efetue o commit inicial e partilhe o seu repositório com o prof. (user : `css-lti-000`).

### 5.11.2 Criação do projeto de tipo Maven

Na pasta do repositório criar um projeto de tipo Maven. Adicionar os pacotes `domain`, `main` e `utils`. No pacote `domain` defina as classes `Aluno` e `Disciplina` conforme o diagrama seguinte:



### 5.11.3 Definição do ficheiro pom.xml

Neste projeto o ficheiro pom deve conter várias configurações para atingir vários objetivos.

- Para especificar a versão do Java que deve ser usada e o tipo de codificação dos caracteres, é necessário adicionar as propriedades seguintes:

```

<properties>
 <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
 <maven.compiler.source>1.8</maven.compiler.source>
 <maven.compiler.target>1.8</maven.compiler.target>
</properties>

```

**copy**

Nota: se tiver uma versão mais recente do Java, indique a sua versão.

- Para usar uma base de dados de tipo Derby temos que acrescentar a dependência:

```

<dependency>
 <groupId>org.apache.derby</groupId>
 <artifactId>derby</artifactId>
 <version>10.12.1.1</version>
</dependency>

```

**copy**

- Para usar MySQL:

```

<dependency>
 <groupId>mysql</groupId>
 <artifactId>mysql-connector-java</artifactId>
 <version>8.0.19</version>
</dependency>

```

**copy**

- Eclipselink é o persistence provider que vamos usar para a ligação à base de dados:

```

<dependency>
 <groupId>org.eclipse.persistence</groupId>
 <artifactId>eclipselink</artifactId>
 <version>2.5.2</version>
</dependency>
<dependency>
 <groupId>org.eclipse.persistence</groupId>
 <artifactId>org.eclipse.persistence.jpa.modelgen.processor</artifactId>
 <version>2.5.2</version>
 <scope>provided</scope>
</dependency>
copy

```

- Para as anotações JPA:

```

<dependency>
 <groupId>javax.annotation</groupId>
 <artifactId>javax.annotation-api</artifactId>
 <version>1.3.2</version>
</dependency>
copy

```

#### 5.11.4 Transformar as classes em Entidades

- Quais são modificações que devem ser feitas nas classes **Aluno** e **Disciplina** para que passam a ser entidades reconhecidas pelo JPA?
- Como definir a chave primária das tabelas correspondentes ?
- Qual é a relação entre as duas entidades ? Como se traduz na anotações a introduzir no código ?

#### 5.11.5 Criação do ficheiro persistence.xml

Se não existir, criar a pasta **src/main/resources/META-INF/**. Nesta pasta criar o ficheiro **persistence.xml** com o seguinte conteúdo :

```

<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1">
 xmlns="http://xmlns.jcp.org/xml/ns/persistence"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence http://xmlns.jcp.org/
<persistence-unit name="JPATestDerby" transaction-type="RESOURCE_LOCAL">
 <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
 <class>domain.Aluno</class>
 <class>domain.Disciplina</class>
 <properties>
 <property name="javax.persistence.jdbc.url" value="jdbc:derby:data/derby/JPATestDerby;"/>
 <property name="javax.persistence.jdbc.driver" value="org.apache.derby.jdbc.EmbeddedDriver"/>
 <property name="javax.persistence.jdbc.user" value="" />
 <property name="javax.persistence.jdbc.password" value="" />
 <property name="javax.persistence.schema-generation.database.action" value="create" />
 </properties>
</persistence-unit>
</persistence>
copy

```

- Qual é o nome da unidade de persistência ?
- Quais são as entidades ?
- Que tipo de base de dados é usado ?
- Porque é o *username* e a *password* não têm valores associados ?

### 5.11.6 Testar o programa

A classe principal está no pacote `main`. Aqui está um esboço da classe:

```
package main;

import java.io.IOException;
import java.util.Scanner;

import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

public class Main {
 static EntityManagerFactory emf =
 // Como criar o EntityManagerFactory ????

 public static void main(String[] args) {
 Scanner in = new Scanner(System.in);
 Menu.mainMenu(in);
 System.out.println("done.");
 }
}
```

`copy`

A classe `Menu` é fornecida mas é incompleta. Comente o código que impede a compilação e acrescente o código em falta para fazer funcionar as operações “Adicionar aluno” e “Lista de alunos”. Uma vez que verificou que essas duas operações funcionam corretamente, passa para implementação das operações seguintes.

```
package main;

import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;
import java.util.Set;
import java.util.stream.Collectors;

import javax.persistence.EntityManager;
import javax.persistence.TypedQuery;

import domain.Aluno;
import domain.Disciplina;

/**
 * This class deals with the interactions with the user.
 *
 * @author Thibault Langlois
 */
public class Menu {
 /**
 * The main menu of the application.
 *
 * @param in a Scanner instance that correspond to the input of the program.
 }
```

```
* @return
* @throws IOException
*/
static void mainMenu(Scanner in) throws IOException {
 boolean end = false;
 do {
 System.out.println("Escolhe uma opção: ");
 System.out.println("Lista de alunos1");
 System.out.println("Lista de disciplinas.....2");
 System.out.println("Adicionar aluno.....3");
 System.out.println("Adicionar disciplina.....4");
 System.out.println("Procurar aluno pelo nome.....5");
 System.out.println("Inscriver aluno.....6");
 System.out.println("Procurar disciplina pela sigla....7");
 System.out.println("Terminar.....8");
 System.out.println("> ");
 switch (nextInt(in)) {
 case 1:
 listaDosAlunos();
 break;
 case 2:
 listaDasDisciplinas();
 break;
 case 3:
 adicionarAluno(in);
 break;
 case 4:
 adicionarDisciplina(in);
 break;
 case 5:
 procurarAlunoPeloNome(in);
 break;
 case 6:
 inscreverAluno(in);
 break;
 case 7:
 procurarDisciplinaPelaSigla(in);
 break;
 case 8:
 end = true;
 }
 } while (!end);
}
private static void adicionarAluno(Scanner in) {
 EntityManager em = null;
 System.out.println("Nome: ");
 String nome = nextLine(in);
 Aluno a = new Aluno(nome);
 try {
 em = // TODO criar o entity manager
 em. // TODO iniciar a transação
 em. // TODO persistir a entidade
 em. // TODO terminar a transação
 System.out.println("Commit done.");
 } finally {
 if (em != null)
 em.close();
 }
}
```

```

private static void listaDosAlunos() {
 List<Aluno> result = null;
 EntityManager em = null;
 try {
 em = Main.emf.createEntityManager();
 TypedQuery<Aluno> query = em.createQuery("SELECT a FROM ALUNO a", Aluno.class);
 result = query.getResultList();
 } catch (Exception e) {
 System.out.println(e.getMessage());
 e.printStackTrace();
 } finally {
 em.close();
 }
 printListaDeAlunos(result);
}

private static void adicionarDisciplina(Scanner in) {
 System.out.print("Designação da disciplina: ");
 String designacao = nextLine(in);
 System.out.print("Sigla da disciplina: ");
 String sigla = nextLine(in);
 System.out.print("Semestre da disciplina (1 ou 2): ");
 int semestre = nextInt(in);
 EntityManager em = null;
 Disciplina d = new Disciplina(designacao, sigla, semestre);
 try {
 em = // TODO criar o entity manager
 em. // TODO iniciar a transação
 em. // TODO persistir a entidade
 em. // TODO terminar a transação
 System.out.println("Commit done.");
 } finally {
 if (em != null)
 em.close();
 }
}
private static void listaDasDisciplinas() {
 List<Disciplina> result = null;
 EntityManager em = null;
 try {
 em = // TOTO criar o entity manager
 TypedQuery<Disciplina> query = em.createQuery(/* que parâmetros ??? */);
 result = query.getResultList();
 } finally {
 em.close();
 }
 // para apresentação do resultados:
 List<List<String>> printableResult = new ArrayList<>();
 for (Disciplina d : result) {
 List<String> l = new ArrayList<>();
 l.add(Integer.toString(d.getCodigoDeDisciplina()));
 l.add(d.getSigla());
 l.add(Integer.toString(d.getSemestre()));
 l.add(d.getDesignacao());
 printableResult.add(l);
 }
 // A classe utils.Table é fornecida.
 System.out.println(utils.Table.tableToString(printableResult));
}
private static void procurarDisciplinaPelaSigla(Scanner in) {

```

```
System.out.print("Sigla da disciplina: ");
String sigla = nextLine(in);
Disciplina result = null;
EntityManager em = null;
try {
 em = // TODO criar o entity manager
 TypedQuery<Disciplina> query =
 em.createQuery("SELECT a FROM DISCIPLINA a where a.sigla = ?1",
 Disciplina.class);
 query.setParameter(1, sigla);
 result = query.getSingleResult();
} finally {
 em.close();
}
System.out.println(result);
}

private static void inscreverAluno(Scanner in) {
 EntityManager em = null;
 System.out.print("Número do aluno: ");
 int n = nextInt(in);
 System.out.print("Código da disciplina: ");
 int codigo = nextInt(in);
 try {
 em = // TODO criar o entity manager
 em. // TODO iniciar a transação
 Aluno oAluno = em.find(/* que parâmetros ??? */);
 Disciplina aDisciplina = em.find(/* que parâmetros ??? */);
 if (oAluno != null && aDisciplina != null) {
 oAluno.adicionaDisciplina(aDisciplina);
 } else if (oAluno == null) {
 System.out.println("O aluno não foi encontrado.");
 } else {
 System.out.println("A disciplina não foi encontrada.");
 }
 em. // TODO: terminar a transação
 } finally {
 if (em != null)
 em.close();
 }
}
private static void procurarAlunoPeloNome(Scanner in) {
 System.out.println("Nome: ");
 String nome = nextLine(in);
 List<Aluno> result = null;
 EntityManager em = null;
 try {
 em = Main.emf.createEntityManager();
 TypedQuery<Aluno> query =
 em.createQuery(/* que parâmetros ??? */);
 query.// TODO inicializar os parâmetros da query
 result = query. // Obter a lista de resultados da query
 } finally {
 em.close();
 }
 printListaDeAlunos(result);
}
private static void printListaDeAlunos(List<Aluno> result) {
 if (result.size() > 0) {
 List<List<String>> printableResult = new ArrayList<>();
 }
}
```

```

 for (Aluno a : result) {
 List<String> l = new ArrayList<>();
 l.add(Integer.toString(a.getNumeroDoAluno()));
 l.add(a.getNomeDoAluno());
 Set<Disciplina> ds = a.getListadeDisciplinas();
 if (ds.size() > 0) {
 l.add(ds.stream().map((Disciplina d) -> d.getSigla()).collect(Collectors.toList()));
 } else {
 l.add("N/A");
 }
 printableResult.add(l);
 }
 System.out.println(utils.Table.tableToString(printableResult));
 }
}

/*
 * The following methods read several kinds of values from a Scanner. The
 * Scanner may correspond to System.in or to an input file. This allows
 * automatic testing of the application through "use cases" that are tested
 * using the executeUseCase method in the App class. The reason for using these
 * methods instead of Scanner's nextXXX() methods is they allow comments in the
 * use case files. Comments are begin with # and end at the end of the line.
 *
 */
private static int nextInt(Scanner in) {
 String s = in.nextLine();
 while (s.startsWith("#")) {
 s = in.nextLine();
 }
 if (s.contains("#")) {
 try (Scanner sc = new Scanner(s)) {
 s = sc.nextInt();
 }
 }
 return Integer.parseInt(s);
}
private static String nextLine(Scanner in) {
 String s = in.nextLine();
 while (s.startsWith("#")) {
 s = in.nextLine();
 }
 if (s.contains("#")) {
 int p = s.indexOf("#");
 s = s.substring(0, p).trim();
 }
 return s;
}

```

copy

A classe `utils.Table` está definida da seguinte forma:

```

package utils;

import java.util.List;

/**
 *
 * @author tl
 */

```

```

public class Table {

 public static String tableToString(List<List<String>> table) {
 int[] columnsWidth = new int[table.get(0).size()];
 for (int j = 0; j < columnsWidth.length; j++) {
 int max = 0;
 for (List<String> row : table) {
 int w = row.get(j).length();
 if (w > max) {
 max = w;
 }
 }
 columnsWidth[j] = max;
 }
 //---
 StringBuilder sb = new StringBuilder();
 for (int r = 0; r < table.size(); r++) {
 sb.append("+");
 for (int j = 0; j < columnsWidth.length; j++) {
 for (int i = 0; i < columnsWidth[j]; i++) {
 sb.append("-");
 }
 sb.append("+");
 }
 sb.append("\n");
 for (int j = 0; j < columnsWidth.length; j++) {
 sb.append("|");
 String thing = table.get(r).get(j);
 sb.append(thing);
 for (int k = thing.length(); k < columnsWidth[j]; k++) {
 sb.append(" ");
 }
 }
 sb.append("|\n");
 }
 sb.append("+");
 for (int j = 0; j < columnsWidth.length; j++) {
 for (int i = 0; i < columnsWidth[j]; i++) {
 sb.append("-");
 }
 sb.append("+");
 }
 sb.append("\n");
 return sb.toString();
 }
}

copy

```

### 5.11.7 Fim da primeira parte

Neste passo, fazer um commit com a mensagem seguinte:

**GuiaoJPA fim da primeira parte.**

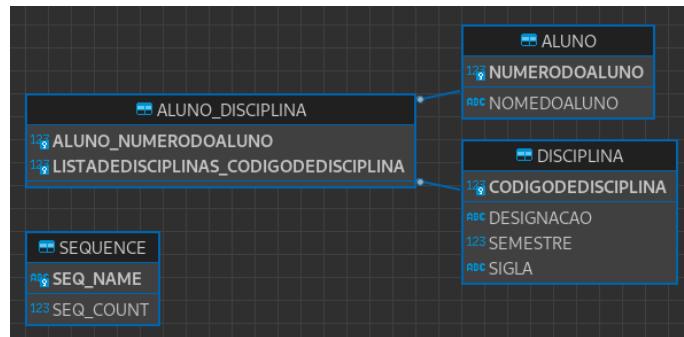
se conseguiu terminar o guião durante a semana e

**GuiaoJPA trabalho feito na primeira semana.**

caso **não** conseguiu terminar.

### 5.11.8 Join Table, Join column

O uso da anotação `@OneToMany` na entidade `Aluno` para representar a relação entre as duas tabelas leve a criação de uma *Join Table* como se vê no diagrama seguinte :



É possível indicar que a relação deve ser representada com uma chave estrangeira na tabela da entidade acrescentando a anotação `@JoinColumn` ao atributo `listaDeDisciplinas` na classe `Aluno`:

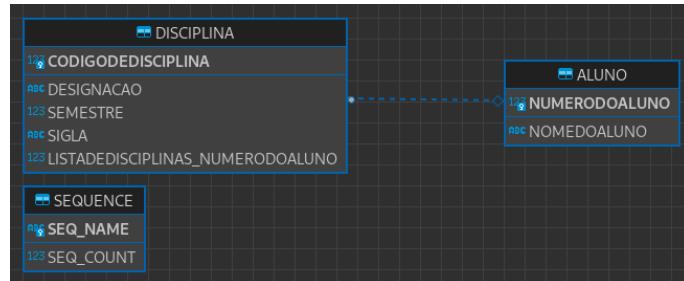
```

@OneToOne
@JoinColumn
private Set<Disciplina> listaDeDisciplinas;

```

`copy`

Neste caso a base de dados fica assim:



- Indique o conteúdo das tabelas a seguir às operações seguintes:
  1. inserir aluno João (id = 1)
  2. inserir aluno Mário (id = 2)
  3. inserir disciplina PCO (id = 51)
  4. inserir disciplina CSS (id = 52)
  5. inscrever João em PCO
  6. inscrever João em CSS
  7. inscrever Mário em PCO
- Qual é o problema encontrado ?
- Imagine o domínio constituído por uma entidade `BlogPost` à qual podem ser associados um número arbitrário de `BlogPostComment`. Será que neste caso o problema é o mesmo ? Porquê ?

### 5.11.9 Relação bi-direcional

Como vimos no passo anterior, o domínio não é muito adaptado ao uso de uma relação *one-to-many*. O objetivo deste passo é modificar o programa de forma a ter uma relação *many-to-many* bidirecional entre as entidades `Aluno` e `Disciplina`.

Tem dúvidas relativamente às diferenças entre as relações entre entidades ? Se for o caso, vê o exemplo seguinte.

Seja um domínio composto por uma entidade `BlogPost` e uma entidade `BlogPostComment`. A cada `BlogPost` pode-se associar vários comentários (`BlogPostComment`). É sinal de uma relação `@*ToMany` do lado do `BlogPost`. Agora tem que decidir se:

- cada `BlogPostComment` pode pertencer a apenas um `BlogPost` (isso resultará em uma restrição única (*unique constraint*), o que significa que nenhum outro `BlogPost` pode ter o mesmo `BlogPostComment`). Neste caso será uma relação `@OneToMany`.

#### `@OneToMany` (`BlogPost -> BlogPostComment`)

Esta relação pode ser representada por uma tabela de junção (*Join Table*) apenas

- se definir explicitamente assim usando a anotação `@JoinTable` ou
- quando for uma relação unidirecional em que o lado proprietário é o lado 'One' (isso significa que na entidade `BlogPost` tem uma coleção de `BlogPostComment`, mas nas `BlogPostComment` não tem nenhuma referência à `BlogPost`).

Pensando no assunto, parece bastante razoável que a tabela de junção seja usada. Não há outra maneira de o SGBD armazenar uma relação entre uma linha na tabela `BlogPost` com várias linhas na tabela `BlogPostComment`.

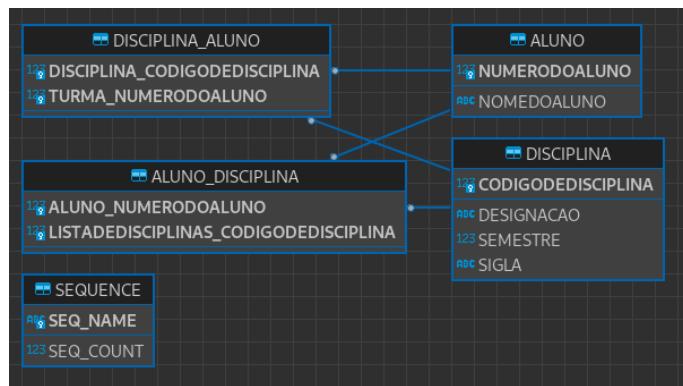
No entanto, se desejar modelar uma relação bidirecional, precisa especificar que a `BlogPostComment` (lado 'Muitos') é o lado proprietário da relação. Neste caso, o SGBD pode criar uma coluna de junção com chave estrangeira na tabela `BlogPostComment` porque cada linha de `BlogPostComment` pode ser conectada com apenas uma `BlogPost`. Desta forma, não tem nenhuma tabela de junção, mas sim chaves estrangeiras simples (ainda, como apontado no início, pode forçar a criação da tabela de junção usando `@JoinTable`).

- cada `BlogPostComment` pode pertencer a vários `BlogPost` e não haverá nenhuma restrição única (*unique constraint*) na tabela `BlogPostComment`) - corresponde a uma relação `@ManyToMany`.

#### `@ManyToMany` (`BlogPost <-> BlogPostComment`)

Esta relação deve ser representada como uma tabela de junção. Basicamente, funciona de maneira muito semelhante ao relacionamento `@OneToMany` unidirecional, mas, neste caso, pode ter várias linhas de `BlogPost` unidas a várias linhas de `BlogPostComment`.

Ao transformar o programa para suportar uma relação bidirecional, deve ter em atenção manter a consistência da informação. Se o aluno A estiver inscrito na disciplina D então A deve pertencer ao conjunto de alunos de D. Pode ser conveniente adicionar alguns métodos para facilitar a tarefa. No final pode visualizar a estrutura da base de dados com o **DBeaver**.



### 5.11.10 TypedQueries vs NamedQueries

Na secção anterior as queries são de tipo `TypedQuery`. A string que define a query está na classe `Menu` o que não faz muito sentido. Neste passo, o objetivo é substituir as queries definidas por uma string JPQL por *named queries* (queries referenciadas pelo nome) cuja definição fica na classe correspondente. Na classe da entidade define-se a query antes da classe:

```
@Entity(name="ALUNO")
@NamedQuery(name="colocar-aqui-o-nome-da-query",
 query="colocar-aqui-a-query-jpql")
public class Aluno implements Serializable {
// ...
copy
```

Se houver várias queries, a sintaxe é:

```
@Entity(name="ALUNO")
@NamedQueries({
 @NamedQuery(name="nome-da-query-1",
 query="query-jpql-1"),
 @NamedQuery(name="nome-da-query-2",
 query="query-jpql-2") })
```

copy

Para usar uma query pelo nome, deve usar o seguinte esquema:

```
TypedQuery<Aluno> query = em.createNamedQuery("nome-da-query", classe-da-entidade);
```

copy

### 5.11.11 Base de dados remota

O objetivo deste passo é passar a usar uma base de dados remota. Quais são as modificações que deve efetuar ao seu programa ?

O servidor de base de dados do DI é `appserver-01.alunos.di.fc.ul.pt` o URL que deve usar para aceder é :

```
jdbc:mysql://appserver-01.alunos.di.fc.ul.pt:3306/csslti00
```

Deve substituir "00" pelo número da conta que lhe foi atribuída (cf mensagem no forum). Deve igualmente alterar o driver de base de dados para `com.mysql.jdbc.Driver`. Fora da FCUL, o servidor está acessível apenas via VPN.

Verifique que o seu programa funciona com a base de dados remota.

### 5.11.12 Fim do guião

Uma vez que terminou este guião, faça um último commit com a mensagem `Guião JPA terminado !`. Não se esqueça de empurrar para o repositório remoto.

# Capítulo 6

## Camada de apresentação (WEB)

Introdução p.[155](#) • Scripts p.[155](#) • Páginas de servidor p.[155](#) • Model View Controller (MVC) p.[156](#) • Client Side vs Server Side p.[157](#) • Aplicação WEB em Java p.[158](#) • Servlet p.[158](#) • Java Server Pages (JSP) p.[161](#) • Guião Tomcat p.[169](#)

### 6.1 Introdução

O aparecimento dos web-browsers trouxe um novo ambiente com muitas vantagens:

- Não é preciso instalar um cliente,
- Uma abordagem UI (user interface) comum,
- Acesso universal a partir de qualquer computador com internet.

Isto tornou mais fácil a construção de aplicações com UI. Uma aplicação WEB começa com o *web-server*. A finalidade do *web-server* é disponibilizar conteúdo através da internet. Interpreta o **URL** (Unified Resource Location) de um pedido para transferir a informação para a respetiva aplicação web. Existem duas formas de estruturar aplicações web: scripts e páginas de servidor.

- Um script é um programa com funções para lidar com pedidos HTTP e gerar páginas web dinamicamente.

### 6.2 Scripts

Um script é um programa com funções para lidar com pedidos HTTP e gerar páginas web dinamicamente. Como exemplos desta abordagem temos scripts CGI (Common Gateway Interface) e Java Servlets. Usam-se expressões regulares para analisar os pedidos ou uma API especializada. Esta análise tem como objetivo dirigir o processamento para elemento de software capaz de responder ao pedido.

#### Pedidos

O pedidos oriundos do cliente são processados pelo servidor para produzir código HTML que constitui a **resposta** ao pedido.

### 6.3 Páginas de servidor

As páginas de servidor são páginas HTML “pre-fabricadas” onde elementos vão sendo preenchidos para construir uma resposta a um pedido. Em determinados pontos da página são incluídos placeholders ou algoritmos para criar um conteúdo dinâmico. Exemplos desta abordagem são PHP, ASP, JSP

etc... Este método funciona bem se houver pouco processamento mas pode rapidamente tornar-se confuso pelo uso de uma mistura de várias linguagens no mesmo ficheiro (HTML, PHP, JSP, Java etc...). Considera-se que os scripts são melhores para interpretar pedidos, enquanto as páginas de servidor são melhores para formatar as respostas. Cada uma destas atividades requer diferentes tipos de ações, é uma das motivações para o padrão *Model View Controller*.

## 6.4 Model View Controller (MVC)

O objetivo do padrão MVC é a separação em partes do processo de interação com o utilizador. Existem três papéis :

- **Model** corresponde aos **objetos** que representam informação sobre o **domínio**.
- **View** é composta pela **representação do modelo na UI**. A View tem apenas a preocupação de mostrar informação e nada mais.
- **Controller** é a parte de **recebe informação do utilizador**, manipula o Model para refletir atualizações e faz o refresh da View de forma adequada.

A separação mais importante é entre a **apresentação** e o **modelo**. Os conceitos que preocupam estas duas partes são muito distintos :

- Na **apresentação** estamos preocupados com os elementos da **interface com o utilizador** (elementos gráficos, linha de comando...) e com a melhor forma de mostrar e pedir informação relevante.
- No **modelo** a nossa preocupação é com a **camada de negócio**, quais são as políticas a implementar, os conceitos relevantes, etc...

Na apresentação podem conviver diversos UI, cada um deles a aceder ao modelo de forma distinta. No modelo não deve existir qualquer preocupação sobre a UI usada.

### Dependência

Existe uma dependência da apresentação para o modelo, mas não ao contrário !

Um problema comum, numa UI gráfica, é a coexistência de várias apresentações do domínio no mesmo ecrã :

- Uma mudança numa vista tem de afetar as restantes.
- Para evitar dependências entre o modelo e a apresentação, Fowler sugere o uso de **Observadores / Listeners**. A apresentação passa a ser um observador do modelo. Cada vez que ocorre uma alteração do modelo, este envia um evento que pode ser ouvido pela apresentação para atualizar as suas diferentes vistas.

Outra separação, não tão crítica, é entre *View* e *Controller*. Esta separação passa muitas vezes despercebida dado que as GUI costumam disponibilizar ambas sem sublinhar a diferença entre os dois tipos de componentes.

A *View* é dependente do *Model*: a informação do *Model* serve para construir a vista apropriada.

O *Controller* está dependente da *View* e do *Model*:

- conhece a *View* para interceptar as operações que ocorrem nesta componente
- conhece o *Model* porque faz repercutir neste as ações que ocorreram na *View*.

O *Model* não depende das outras camadas : as atualizações são feitas de modo indireto via notificação dos observadores interessados (e.g., **padrão Observer**).

**Componentes MVC vs Camadas.** O MVC define a forma de organizar a interação com o utilizador :

- Acontece dentro da camada de apresentação.
- A camada d'apresentação contém “todo” o MVC.

Exemplo: sistema distribuído em que a camada de negócio nem está na mesma máquina onde é feita a interação com o utilizador.

Como aplicar o MVC numa aplicação web ? Numa aplicação web a View é claramente apresentada num browser HTML + CSS. A página que contém a resposta a um pedido pode ser construída no servidor ou no browser. A primeira opção é usada em aplicações **Server Side** enquanto a segunda é usada em aplicações **Client Side**.

Numa aplicação a apresentação depende dos dados e do comportamento da aplicação, logo a mesma página pode ser apresentada de forma diferente dependendo da informação e regras aplicadas. Uma forma de implementar este comportamento é através de um template:

- A parte fixa é escrita em HTML + CSS.
- A parte dinâmica é programada (em Java, no nosso caso).

## 6.5 Client Side vs Server Side

Como aplicar o MVC numa aplicação web ? Numa aplicação web a View é claramente apresentada num browser HTML + CSS. A página que contém a resposta a um pedido pode ser construída no servidor ou no browser. A primeira opção é usada em aplicações **Server Side** enquanto a segunda é usada em aplicações **Client Side**.

Numa aplicação a apresentação depende dos dados e do comportamento da aplicação, logo a mesma página pode ser apresentada de forma diferente dependendo da informação e regras aplicadas. Uma forma de implementar este comportamento é através de um template:

- A parte fixa é escrita em HTML + CSS.
- A parte dinâmica é programada (em Java, no nosso caso).

A parte dinâmica da apresentação pode ser executada :

- No servidor e temos um Server-side template (PHP, JSP etc..) ou
- no browser e neste caso temos um Client-side template (Javascript).

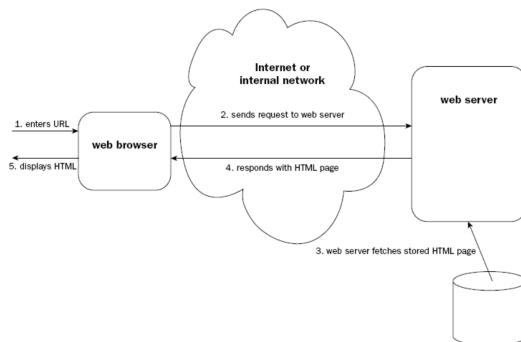
Em ambos os casos existe um servidor web a servir as páginas.

No caso Client-side os eventos são apanhados pelo código que corre no cliente (Javascript). A informação é processada e o acesso à camada de negócio é feita via serviços (Ajax). Neste caso o MVC todo está a correr no cliente (browser).

No caso Server-side, haverá um pedido HTTP enviado para o servidor, o controler está do lado do servidor recebe o pedido e propaga-o para a camada de negócio. O controler está do lado do servidor. Existem duas opções relativamente à implementação do controler. Pode haver um controler por página (page controler) ou um controler para toda a aplicação (front controler).

Como funciona em Java ? Para uma aplicação escrita em Java existem várias opções :

- **Web server**: Tomcat, Wildfly, Glassfish ...
- **Controler**: API `HTTPServlet`.
- **Template**: JSP, JSF etc...



Comunicação browser/web server.

## 6.6 Aplicação WEB em Java

En Java, uma aplicação WEB é constituída por :

- Páginas WEB,
- Classes Java,
- Bibliotecas (formato jar).

A aplicação corre num servidor. Tem de ser instalada (**deployed**) no servidor que trate de a executar. A informação necessária acerca da aplicação está fornecida num ficheiro de configuração chamado **web.xml**. O ficheiro que contém a aplicação pronta para ser instalada tem a extensão **war**.

## 6.7 Servlet

Um Java Servlet executa serviços do lado do servidor. Os servlets são objetos de classes Java que respeitam o **Java Servlet API**, um protocolo que determina como responder a pedidos. Estes recebem pedidos e geram respostas em conformidade. Um servlet gera HTML como resposta ao pedido.

É necessário haver um **web container** (i.e., **servlet container**), uma aplicação capaz de interagir com os servlets e de gerir o seu ciclo de vida (como o **Tomcat** ou o **Wildfly**). A classe **HTTPServlet** serve de base para criar os servlets de uma aplicação.

### 6.7.1 WEB container

Um **Web Container**, também conhecido como *Servlet Container*, é um componente essencial na arquitetura Java EE (Enterprise Edition) que gere a execução de aplicações WEB baseadas em tecnologias Java, como *Servlets* e *JSPs (JavaServer Pages)*. O *Web Container* é uma parte fundamental do servidor de aplicações Java EE e desempenha um papel central no processamento de pedidos e respostas em aplicações da WEB.

Aqui estão algumas das principais responsabilidades do *Web Container* :

- **Gestão do ciclo de vida:** O *Web Container* é responsável por criar, inicializar e gerir os ciclos de vida dos *Servlets*. Cria as instâncias de *Servlets* quando necessário e as destrói quando não são mais necessários.
- **Processamento dos pedidos:** O *Web Container* recebe pedidos HTTP de clientes (geralmente navegadores da web), analisa os pedidos e encaminha-os para os *Servlets* ou *JSPs* apropriados com base nas configurações de mapeamento.
- **Processamento de Respostas:** Após a execução dos *Servlets* ou *JSPs*, o *Web Container* recebe as respostas geradas por esses componentes e as envia de volta ao cliente. Lida com a geração de cabeçalhos HTTP, cookies e conteúdo da resposta.
- **Gestão de Threads:** O *Web Container* gera *pools* de *threads* para lidar com vários pedidos simultaneamente. Aloca uma *thread* para cada pedido e garante que o pedido seja processado de forma concorrente e segura.
- **Segurança:** O *Web Container* também é responsável pela implementação de recursos de segurança, como autenticação e autorização. Pode controlar o acesso a recursos da web com base nas configurações de segurança.

O *WEB container* efetua as etapas seguintes no tratamento de um pedido :

1. Recebe o pedido HTTP,
2. Cria um objeto Java **HttpRequest** e preenche-o com a informação do pedido HTTP,  
⇒ Existe um novo objeto **HttpRequest** por cada pedido,
3. A partir do URL determina a *Servlet* que responde ao pedido,
4. Cria uma nova *thread* para executar a chamada ao método *service* desta *Servlet*,
5. Chama o método **service()** e passa-lhe o objeto **HttpRequest**.

### 6.7.2 Exemplo de Servlet

Aqui está um exemplo de *Servlet*:

```
package main;
public class HelloWorldExample extends HttpServlet {

 public void doGet(HttpServletRequest request,
 HttpServletResponse response)
 throws IOException, ServletException {

 String msg = "Hello, world!";
 response.setContentType("text/html");
 PrintWriter out = response.getWriter();
 out.println("<html>");
 out.println(" <head>");
 out.println(" <title>Hello World</title>");
 out.println(" </head>");
 out.println(" <body>");
 out.println(msg);
 out.println(" </body>");
 out.println("</html>");
 }
 // ...
}
```

**copy**

### 6.7.3 Funcionamento e definição de um Servlet

A classe `HTTPServlet` possui vários métodos usados pelo *WEB container* para gerir o **ciclo de vida** do *Servlet* e lidar com os pedidos :

- `init()`: executado quando o objeto é carregado pela primeira vez.
- `service(...)`: invocado por cada pedido. É usado para reenviar o pedido para os vários `doXXX()` entre outras coisas. Não deve ser reescrito (*overridden*)!
- `doGet(...)`, `doPost(...)`: lidam com os pedidos, deve-se reescrevê-los para se obter o comportamento devido. É aqui que (quase) todo o trabalho é feito.
- `destroy()`: invocado quando o servidor vai apagar definitivamente o objeto *servlet*.

Tanto `init()` como `destroy()` só são invocados pelo *WEB container* uma vez, e não de cada vez que ocorre um pedido. Para definir um *servlet* para a sua aplicação basta criar uma **sub-classe** de `HttpServlet` e re-definir os métodos `doGet` e/ou `doPost`.

Os métodos `doGet()` e `doPost()` recebem dois objetos:

```
public void doGet(HttpServletRequest request,
 HttpServletResponse response) {
 ...
}
```

**copy**

O parâmetro `request` contém informação sobre a transação originada no browser, é do tipo `javax.servlet.http.HttpServletRequest`. Por exemplo pode obter o número IP do cliente com o método `getRemoteHost` :

```
out.println(request.getRemoteHost());
```

**copy**

O parâmetro `response` contém um objeto de tipo `HttpServletResponse` que permite associar conteúdo à resposta que será enviada para o browser. Por exemplo a chamada `response.setContentType("txt/html")` indica que a resposta vai conter código HTML. A chamada `response.sendRedirect(anotherUrl)` pode ser usada para redirigir o browser para outra página.

Os métodos `doGet()` e `doPost()` recebem pedidos submetidos via formulários (HTML) `GET` e `POST`. Na página HTML temos um formulário :

```
<FORM METHOD=POST ACTION="address">
 <INPUT TYPE="text" NAME="param1"> ...
</FORM>
```

**copy**

Quando o utilizador submete o formulário (via um botão *Submit*, por exemplo), o pedido é recebido pelo WEB container é encaminhado para o servlet previsto para o seu tratamento :

```
@WebServlet ("/address")
public class Test extends HttpServlet { ...
 public void doPOST(HttpServletRequest request,
 HttpServletResponse response) { ...
 String s = request.getParameter("param1")
 // ...
 // processamento da resposta
 }
}
```

**copy**

**Get vs. Post** : Deve-se usar **GET** para operações que **não alterem o estado dos objetos** em questão. Ou seja, não faz diferença se o comando for repetido várias vezes (o que resulta numa operação idempotente). Estas operações são tipicamente de consulta e visualização. Quando o método GET é usado :

- Os comandos podem ser guardados como *bookmark* para serem repetidos no futuro,
- Vê-se os parâmetros do pedido no url (maior risco de segurança)
- Limites na dimensão da string que codifica os pedidos.

Deve-se usar **POST** para operações de criação, atualização e destruição de dados. Não se vê os parâmetros no *URL* e não existe limite para a dimensão do pedido enviado (os nomes/valores dos vários parâmetros). Não se pode fazer *bookmark*.

A resposta enviada por um Servlet não é limitada a um página HTML, pode enviar qualquer tipo de dados. No exemplo seguinte a resposta é uma folha XL:

```
@WebServlet ("/ApplesAndOranges")
public class ApplesAndOranges extends HttpServlet {
 @Override
 public void doGet(HttpServletRequest request,
 HttpServletResponse response)
 throws ServletException, IOException {

 response.setContentType("application/vnd.ms-excel");
 PrintWriter out = response.getWriter();
 out.println("tQ1tQ2tQ3tQ4tTotal");
 out.println("Applest78t87t92t29t=SUM(B2:E2)");
 out.println("Orangest77t86t93t30t=SUM(B3:E3)");
 }
}
```

**copy**

Este Servlet pode ser ativado pelo seguinte formulário :

```
<html> ... <body>
 <FORM METHOD=GET ACTION="ApplesAndOranges">
 <input type="submit" value="Submit">
 </FORM>
</body></html>
```

**copy**

Ou diretamente no browser usando o endereço `localhost:8080/testeServlet/ApplesAndOranges`.

## 6.8 Java Server Pages (JSP)

A abordagem com servlets tem várias desvantagens:

- Tem tendência a provocar código repetitivo.
- Qualquer alteração necessita de ser recompilada e ser feito o *redeploy* no *WEB container*.
- Mistura o desenho das páginas HTML com o código Java, ficando ambos à responsabilidade do programador.

Em resposta a essas desvantagens foi criada a tecnologia *Java Server Pages* (JSP). A tecnologia JSP não necessita que o código Java seja (re)compilado. Muitas vezes nem é preciso escrever código Java. O *WEB container* trata de recompilar o JSP quando é modificações são efetuadas no ficheiro JSP.

O JSP é uma tecnologia que permite a geração de páginas WEB em formato HTML ou XML. A abordagem é semelhante a do PHP. Um ficheiro JSP vai incluir código Java e código HTML. O *WEB container* é encarregado de gerir o JSP. Quando recebe o primeiro pedido que deve ser tratado por um JSP específico, este ficheiro JSP é convertido em código Java (que corresponde a um Servlet) e compilado (e executado). O JSP não acrescenta funcionalidades relativamente a aos Servlets, apenas facilita a produção e manutenção de páginas HTML.

Princípio funcionamento p.161 • Conteúdo dinâmico p.162 • Sessões JSP p.164 • Interação com classes p.164 • Expression Language p.166 • Servlet vs. JSPs p.169 • MVC revisitado p.169

### 6.8.1 Princípio funcionamento

Segue um exemplo de ficheiro JSP:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
 pageEncoding="UTF-8"%>
<%@ page import="java.util.Date" %>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>This is a title</title>
</head>
<body>
<h1>This is a section</h1>
<h2>Today: <%=new Date() %> </h2>
 <% int banana = 555; %>
<h3>this is five five five: <%=banana %></h3>
</body>
</html>
```

[copy](#)

Um ficheiro JSP é basicamente um ficheiro HTML com etiquetas (*tags*) especiais. Quando o JSP é invocado por um pedido, o servidor preenche dinamicamente as tags com o conteúdo apropriado. Há uma separação: por um lado o *web designer* altera o JSP para trabalhar a parte visual; por outro lado o programador lida com as classes Java que manipulam a informação.

[Como funciona ?](#) : O *WEB container* usa a informação contida no JSP para criar um servlet, o código Java é gerado automaticamente... Numa pasta muito remota (do tipo /MyWorkspace/.metadata/.plugins/org.eclipse.wst.server.core/tmp0/work/Catalina/localhost/Hell... um ficheiro Java é criado.

[copy](#)

## 6.8.2 Conteúdo dinâmico

No contexto de uma aplicação WEB, cada vez que a página é carregada, o resultado pode ser diferente. O JSP permite incluir um conteúdo dinâmico. Numa página JSP são usados tags para introduzir os elementos dinâmicos :

```
<HTML> <BODY>
 The time is now <%= new java.util.Date() %>
</BODY> </HTML>
```

**copy**

ou

```
<%@ page import="java.util.*" %>
<HTML> <BODY>
<% Date date = new Date(); %>
 The time is now <%= date %>
</BODY> </HTML>
```

**copy**

A página JSP é transformada num Servlet e os tags permitem a inclusão de código Java em vários pontos da classe Servlet :

- **<%@ page attribute="value"...%>** : Define diretivas para a página JSP, como a importação de classes Java, especificações da página e configurações do JSP. Exemplos:

```
<%@ page language="java" contentType="text/html; charset=UTF-8" %>
```

**copy**

Os atributos que podem ser usados são:

- **language**: Define a linguagem de programação usada na página (geralmente “java”).
- **contentType**: Define o tipo de conteúdo da página (por exemplo, “text/html” para HTML).
- **pageEncoding**: Define o encoding de caracteres para a página.
- **import**: Importa classes Java para uso na página.
- **session**: Controla o uso de sessões HTTP na página.
- **isErrorHandler**: Indica se a página JSP é uma página de tratamento de erros.
- **errorPage**: Define a página de tratamento de erros para a página atual.

- **<%...%>** : é usado para incluir uma **instrução** Java. Por exemplo: `<% Date date = new Date(); %>`.
- **<%=...%>** : é usado para incluir uma **expressão** Java. Por exemplo: `<%= new java.util.Date() %>`.
- **<%!...%>** : Permite a declaração de variáveis e métodos que podem ser usados em toda a página JSP. As declarações são colocadas fora dos métodos e têm escopo de classe. Por exemplo:

```
<%
private int contador = 0;

public int getContador() {
 return contador;
}
%>
```

**copy**

- **<%---...--%>**: Permite incluir comentários numa página JSP:

```
<%-- Página realizada por João Santos --%>
copy
```

No contexto de uma página JSP, algumas variáveis são definidas. Em particular, a variável `out` contém um `Writer` que permite escrever para a página HTML produzida para a resposta:

```
<HTML> <BODY>
<% java.util.Date date = new java.util.Date(); %>
Hello! The time is now
<% out.println(String.valueOf(date)); %>
</BODY> </HTML>
```

[copy](#)

As variáveis `request` e `response` estão disponíveis:

```
<% response.setContentType("txt/html"); %>
<HTML> <BODY>
<% java.util.Date date = new java.util.Date(); %>
Hello! The time is now
<% out.println(date);
out.println("
Your machine's address is ");
out.println(request.getRemoteHost()); %>
</BODY> </HTML>
```

[copy](#)

Exemplo: **Criação de uma tabela HTML**:

```
<% int n = 5; %>
<TABLE BORDER=2>
<% for (int i = 1; i <= n; i++) { %>
<TR>
<TD>Number</TD>
<TD> <%= i %> </TD>
</TR>
<% } %>
</TABLE>
```

[copy](#)

O código JSP pode estar distribuído em vários ficheiros a inclusão é feita da seguinte forma:

```
<HTML> <BODY>
A tabela produzida é:

<jsp:include page="tabela.jsp" %>
</BODY> </HTML>
```

[copy](#)

No exemplo seguinte a variável `theDate` e o método `getDate()` estão definidos no âmbito da classe:

```
<%@ page import="java.util.*" %>
<HTML> <BODY>
<%! Date theDate = new Date();
Date getDate() {
 return theDate;
} %>
Hello! The time is now <%= getDate() %>
</BODY> </HTML>
```

[copy](#)

O valor declarado da data não muda após o *reload* da página. As declarações são executadas apenas na primeira vez que a página é enviada nesta sessão. **Cuidado ! Se houver vários usos da página,**

haverá interferências entre os acessos concorrentes à página. É melhor usar os objetos `request` ou `session` (cf. adiante).

### 6.8.3 Sessões JSP

Ao usar a aplicação o utilizador vai obter no seu browser várias páginas e efetuar interações. Uma **sessão** é um objeto associado a **cada visitante**. Pode ser usado como uma *hash table* para incluir informação. No exemplo seguinte é apresentado ao utilizador um formulário :

```
<HTML> <BODY>
<FORM METHOD=POST ACTION="SaveName.jsp"> What's your name?
<INPUT TYPE=TEXT NAME=username SIZE=20>
<INPUT TYPE=SUBMIT> </FORM>
</BODY> </HTML>
```

**copy**

O utilizador preenche o seu nome e carrega no botão “submit”. No servidor o processamento do pedido é encaminhado para a página `SaveName.jsp`:

```
<% String name = request.getParameter("username");
 session.setAttribute("theName", name); %>
<HTML> <BODY>
 Continue
</BODY> </HTML>
```

**copy**

O código da página recebe o valor do nome (usando o método `getParameter` da classe `request`) e o guarda num atributo da sessão (método `setAttribute` da classe `session`). Quando o utilizador carrega no link “Continue” o processamento é encaminhado para a página `NextPage.jsp`. Nesta página será possível obter o nome do utilizador via o objeto `session`.

#### Variável session

A variável **session** é automaticamente inicializada pelo WEB container.

Continuando com o mesmo exemplo, para aceder ao nome na página `NextPage.jsp` será usado o método `getAttribute`:

```
<HTML> <BODY>
Hello, <%= session.getAttribute("theName") %>
</BODY> </HTML>
```

**copy**

A sessão é mantida até um período de *timeout*, a partir do qual é considerado que o utilizador já não está a visitar o site e a informação é eliminada. O *timeout* é um parâmetro ajustável.

### 6.8.4 Interação com classes

Um aspeto que é preciso ter em conta é a organização do código quando a complexidade da aplicação cresce. Se temos que incluir 100 linhas de código, temos duas opções :

- Escrever as linhas no ficheiro JSP
- Escrever uma classe com esse código e incluir uma linha de código no JSP (para invocar a classe).

A segunda abordagem é preferível ! podemos usar o IDE para desenvolver, compilar e testar o nosso código antes de o usar e o código pode ser re-utilizado em várias páginas JSP.  
Seja o seguinte formulário:

```
<HTML> <BODY>
 <FORM METHOD=POST ACTION="SaveName.jsp">
 What's your name? <INPUT TYPE=TEXT NAME=username SIZE=20>

 What's your e-mail address? <INPUT TYPE=TEXT NAME=email SIZE=20>

 What's your age? <INPUT TYPE=TEXT NAME=age SIZE=4>
 <P> <INPUT TYPE=SUBMIT>
 </FORM>
</BODY> </HTML>
```

**copy**

Após introduzir os dados é enviado o pedido para carregar a página `SaveName.jsp`. Vamos usar uma classe Java para lidar com os dados.

Esta classe vai ser de um tipo especial chamado **Java Bean**. Um *Java Bean* deve ter as características seguintes:

- ter um construtor nulário (sem argumentos),
- os atributos têm de ser privados e ter *getters/setters* (usa-se `isX` para atributos booleanos),
- tem de ser serializável e
- pertencer a um pacote.

Exemplo:

```
package userPack;
public class UserData implements java.io.Serializable {
 private String username;
 private String email;
 private int age;

 public void setUsername(String value) { username = value; }
 public void setEmail(String value) { email = value; }
 public void setAge(int value) { age = value; }

 public String getUsername() { return username; }
 public String getEmail() { return email; }
 public int getAge() { return age; }
}
```

**copy**

Este objeto não é um objeto do domínio ! É uma classe que serve apenas na interação com a camada de apresentação.

A página `SaveName.jsp` vai usar esta classe Java:

```
<jsp:useBean id="user" class="userPack.UserData" scope="session"/>
<jsp:setProperty name="user" property="*" />
<HTML> <BODY>
 Continue
</BODY> </HTML>
```

**copy**

- A tag `usebean` vai procurar o objeto `userPack.userData` que pertença à sessão ou cria um se ainda não existir, i.e., similar a:

```
<% user.UserData user = new userPack.UserData(); %>
```

**copy**

- A tag `setProperty` vai buscar a informação do formulário (que foi submetida para esta página) e para aqueles atributos com nomes iguais, irá fazer a atualização automática de todos (é o efeito do “\*”).

Quando se clicar no próximo link iremos para a `NextPage.jsp`. Para recuperar a informação usa-se o objeto `user`, instância do Java Bean definido anteriormente:

```
<jsp:useBean id="user" class="userPack(userData" scope="session"/>
<HTML> <BODY>
You entered

 Name: <%= user.getUsername() %>

 Email: <%= user.getEmail() %>

 Age: <%= user.getAge() %>
</BODY> </HTML>
```

[copy](#)

A tag `usebean` é usada aqui para encontrar o objeto “`user`” e invocar os métodos necessários que devolvem a informação armazenada.

Também se pode usar:

```
<jsp:getProperty name="user" property="age" />
<jsp:setProperty name="user" property="age" value="16" />
```

[copy](#)

**Uso das variáveis pré-definidas** As variáveis pré-definidas, `request`, `response`, `out`, `session` São locais (ao JSP) e não são conhecidas pelas classes (*beans*) que fizermos. Se houver essa necessidade, a solução é passar estes objetos como argumentos de métodos:

```
public class Cl {
 public static void method(HttpSession s) { ... }
```

[copy](#)

No ficheiro JSP:

```
<% pacote.Cl.method(session); %>
```

[copy](#)

**Âmbito dos beans** Quando usamos um bean ele tem um âmbito (*scope*) associado:

```
<jsp:useBean id="user" class="user.userData" scope="session"/>
```

[copy](#)

O *scope* determina a “duração de vida” do bean:

- `scope="page"`: o bean é criado e usado na página onde é criado. Quando mudar de JSP, perde-se o objeto. É o valor por defeito.
- `scope="request"`: o bean é criado e usado durante este pedido (*request*) particular (pode incluir vários JSPs se forem feitos *forwards*)
- `scope="session"`: o bean é criado e usado durante a interação do utilizador com o *WEB server* durante vários pedidos. É destruído quando a sessão termina.
- `scope="application"`: o bean é guardado no contexto da aplicação WEB, todos os *servlets* podem aceder ao estado destes objetos.

Quanto mais “largo” é o âmbito mais recursos são necessários do lado do servidor. Aplicações com muitos utilizadores em simultâneo têm tendência a usar âmbitos mais restritos (*page* ou *session*).

## 6.8.5 Expression Language

As tags com prefixos como `jsp:` e `c:` são usadas em páginas JSP para incorporar funcionalidades adicionais e estender a capacidade do JSP. Esses prefixos indicam a utilização de bibliotecas de tags específicas que fornecem conjuntos de tags predefinidas para realizar tarefas comuns de forma mais conveniente e abstrata. Aqui está uma explicação mais detalhada:

### 1. Prefixo `jsp:` (*JavaServer Pages Standard Tag Library - JSTL*):

- O prefixo `jsp:` é usado para incorporar tags da biblioteca JSTL em páginas JSP.

- A JSTL é uma biblioteca de tags padrão que fornece tags para tarefas como iteração, condicionais, formatação, manipulação de datas e muito mais. Ela é usada para simplificar e limpar o código JSP, separando a lógica da apresentação.
- Exemplo: `<c:forEach>`, `<c:if>`, `<c:choose>`, `<fmt:formatDate>`, etc.
- Uso típico: A JSTL é frequentemente usada para substituir tags `<% ... %>` e expressões `<%= ... %>` em páginas JSP, tornando o código mais legível.

## 2. Prefixo c: (*Custom Tags ou Community Tags*):

- O prefixo `c:` pode ser usado para incorporar tags personalizadas ou tags de comunidade em páginas JSP.
- As tags personalizadas são criadas pelo programador ou fornecidas por terceiros para estender as funcionalidades do JSP. Elas podem encapsular lógica complexa ou fornecer componentes reutilizáveis.
- Exemplo: `<c:myCustomTag>`, onde `myCustomTag` é uma tag personalizada.
- Uso típico: As tags personalizadas são usadas para dividir o código em componentes reutilizáveis e promover a modularidade e a reutilização de código em páginas JSP.

## 3. Outros Prefixos:

- Além dos prefixos mencionados, existem muitos outros prefixos que podem ser usados em páginas JSP, dependendo das bibliotecas de tags que você deseja incorporar.
- Por exemplo, o prefixo `sql:` pode ser usado para incorporar tags relacionadas ao processamento de bases de dados, enquanto o prefixo `x:` pode ser usado para tags XML.

Esses prefixos são usados para diferenciar entre as várias bibliotecas de tags que podem ser usadas em uma página JSP. Cada biblioteca de tags tem seu próprio conjunto de funcionalidades e finalidades, permitindo que os desenvolvedores escolham as tags mais adequadas para suas necessidades e, assim, tornem o desenvolvimento de aplicações WEB em JSP mais eficiente, organizado e legível.

Para além das bibliotecas de tags existe mais uma maneira de incluir conteudos dinâmicos em páginas JSP: a ***Expression Language***.

A Expression Language (EL) é uma linguagem de expressão que permite a inclusão de valores dinâmicos em uma página JSP. EL foi introduzida para simplificar a inclusão e a manipulação de dados diretamente em páginas JSP, tornando o código mais limpo e legível. Aqui estão alguns aspectos importantes sobre a Expression Language:

- **Sintaxe:** A EL é incorporada numa página JSP usando a sintaxe  `${expressão}`. A expressão dentro das chaves  `${}` é avaliada e seu resultado é inserido na página quando a página é produzida no navegador.
- **Âmbito:** A EL permite aceder objetos armazenados em diferentes âmbitos, como o âmbito do `request`, o âmbito da sessão ou o âmbito da aplicação, usando os prefixos apropriados. Por exemplo,  `${requestScope.nome}` aceede a um atributo no âmbito do `request` com o nome “`nome`”.
- **Funções:** A EL inclui funções pré-definidas que podem ser usadas para realizar operações comuns, como formatação de datas, manipulação de strings e outras transformações de dados. Por exemplo,  `${fn:length(texto)}` retorna o comprimento da string “`texto`”.
- **Acesso a Objetos Java:** A EL permite o acesso direto a objetos Java, como variáveis, propriedades e métodos de objetos. Isso torna mais fácil recuperar e exibir dados de objetos Java em uma página JSP.

A EL torna mais rápida a inserção de expressões:

```
<p>Há ${16 * 51 % 12} soluções</p>
${5 > 1+2}
${empty param.action} <%-- o parâmetro action está vazio? --%>
```

**copy**

A biblioteca de tags *Custom Tags* é geralmente usada em conjunto com a EL:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<c:set var="salario" value="500" scope="session"/>
Um salario de ${salario} euros.
```

**copy**

Esta biblioteca permite “programar” na página JSP. Aqui está um exemplo com a instrução **c:if**:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<c:set var = "salario" value="500" scope="session"/>
 Um salario de ${salario} euros é
<c:if test="${salario<1200}"> baixo </c:if>
<c:if test="${salario>=1200}"> médio ou alto </c:if>
```

**copy**

e com a instrução **c:choose** (equivalente a um `switch`):

```
<c:choose>
 <c:when test="${salario<1200}"> baixo </c:when>
 <c:when test="${salario>=1200}"> médio ou alto </c:when>
 <c:otherwise>Não deveria chegar aqui!!</c:otherwise>
</c:choose>
```

**copy**

A EL permite criar ciclos:

```
<c:forEach var="i" begin="1" end="5">
 Item <c:out value="${i}" /><p>
</c:forEach>
```

**copy**

Exemplo de criação de uma tabela HTML usando a instrução **c:forEach**:

```
<table>
<c:forEach var="cor" items="${oMeuBean.asMinhasCores}">
 <tr><td> Cor: ${cor} </td></tr>
</c:forEach>
</table>
```

**copy**

Neste exemplo `asMinhasCores` é um `HashMap` devolvido pelo respetivo método.

**c:forTokens** é outra instrução de ciclo que permite iterar sobre os elementos de uma lista:

```
<c:forTokens items="A,B,C,D" delims="," var="name">
 <p> <c:out value="${name}" /> </p>
</c:forTokens>
```

**copy**

Neste caso aparecem quatro parágrafos, cada um com a sua letra.

A EL permite também **acceder aos métodos** de um bean:

A idade é de \${user.age} anos.

A idade é de \${user["age"]} anos.

**copy**

neste exemplo o getter da classe é invocado. A mesma sintaxe é usada também para aceder a estruturas de dados:

O primeiro elemento do array do bean ‘dados’ é \${dados.oArray[0]}  
O nome do departamento do cliente: \${oCliente.oDept.nome}  
A chave ‘verde’ no hashMap ‘mapa’ é: \${mapa["verde"]}

**copy**

É também possível remover uma variável de um dadi âmbito:

```
<c:set var="salary" scope="session" value="${2000*2}" />
<p>Antes da remoção: <c:out value="${salary}" /></p>
<c:remove var="salary"/>
<p>Depois da remoção: <c:out value="${salary}" /></p>
```

**copy**

A seguir à palavra “remoção:” não aparece nada, a variável `salary` foi removida do contexto.

**Objetos implicitos:** No contexto da EL várias variáveis são definidas implicitamente (é o WEB container que trata de criar as instâncias e inicializar essas variáveis):

- Os âmbitos são acessíveis via as variáveis `pageScope`, `requestScope`, `sessionScope` e `applicationScope`.
- Os parâmetros dos formulários podem ser recolhidos via `param` (dados com valores únicos) e `paramValues` (arrays). Ex.:  `${param["age"]}`
- Os valores relativos aos headers dos HTML:  `${header["host"]}`
- O contexto da página atual:  `${pageContext.servletContext.serverInfo}`

Em resumo: A EL providencia um acesso conciso a vários elementos:

- Variáveis nos âmbitos conhecidos
- Propriedades dos beans
- Elementos das coleções Java
- Elementos HTTP típicos
- Formatação de texto (tags `<fmt:xxx>`)
- Acesso a BDs (tags `<sql:xxx>`)
- Manipulação de documentos XML (tags `<x:xxx>`)
- Manipulação de strings (tags `<fn:xxx>`)

Porém, vamos seguir a convenção que:

1. Usar EL **apenas para mostrar dados** produzidos por código Java.
2. **Evitar escrever lógica do domínio** (i.e., camada de negócios) com EL

## 6.8.6 Servlet vs. JSPs

No contexto de uma aplicação WEB em Java existem duas abordagens para criar uma página WEB: Definir um *Servlet* ou uma página JSP. Como decidir entre essas duas opções ?

- Usar unicamente servlets quando:
  - a saída (resposta) é binária (eg., responder com uma imagem)
  - Não há saída (eg, fazer um *forward* para outra página)
  - Quando a paginação é muito variável (por exemplo: *web portal*, jornal)
- Usar unicamente JSPs quando:
  - A saída é texto (eg, HTML)
  - A paginação é relativamente fixa.
- Combinar os dois quando:
  - Um pedido pode resultar em diferentes resultados
  - Diversos programadores a desenvolver partes diferentes do sistema
  - Sistemas com processamento complexo de dados, mas com paginação relativamente fixa.

## 6.8.7 MVC revisitado

## 6.9 Guião Tomcat

Instalação no seu computador (Ubuntu e Mac) p.[170](#) • Instalação nos computadores dos labs p.[172](#) • WEB app HelloWorld p.[174](#) • Modificação e «redeployment» p.[175](#) • Santa Claus Servlet p.[176](#) • SantaClausJSP p.[180](#)  
 • One more thing p.[182](#)

### 6.9.1 Instalação no seu computador (Ubuntu e Mac)

Para instalar o Tomcat basta usar num terminal o comando seguinte:

```
sudo apt-get install tomcat9
```

Se esiver a usar um Mac pode usar:

```
brew install tomcat@9
```

Será necessário entrar a sua password.

Uma vez instalado o servidor deve estar a correr. Para verificar pode usar o comando seguinte:

```
sudo systemctl status tomcat9.service
```

No Mac o comando será:

```
brew services status tomcat@9
```

Se observar algo semelhante a esta imagem :

```
tl@Ubuntu20:~$ sudo systemctl status tomcat9.service
[sudo] password for tl:
● tomcat9.service - Apache Tomcat 9 Web Application Server
 Loaded: loaded (/lib/systemd/system/tomcat9.service; enabled; vendor preset: en
 Active: active (running) since Tue 2021-11-30 23:26:39 WET; 10h ago
 Docs: https://tomcat.apache.org/tomcat-9.0-doc/index.html
 Main PID: 27524 (java)
 Tasks: 29 (limit: 4651)
 Memory: 111.4M
 CPU: 27524 /usr/lib/jvm/java-11-openjdk-amd64/bin/java -Djava.util.logging

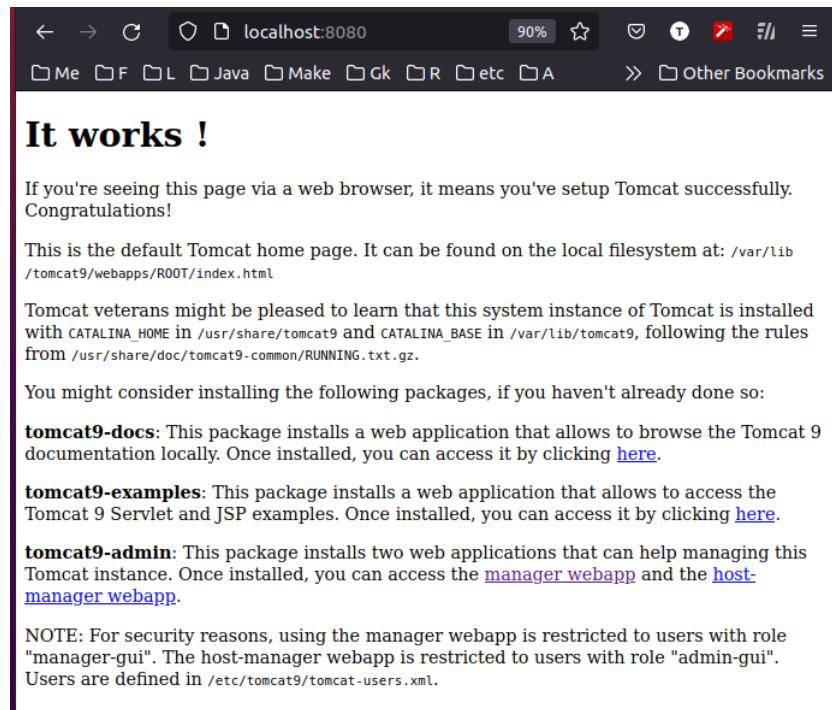
nov 30 23:26:43 Ubuntu20 tomcat9[27524]: OpenSSL successfully initialized [OpenSSL 1
nov 30 23:26:44 Ubuntu20 tomcat9[27524]: Initializing ProtocolHandler ["http-nio-808
nov 30 23:26:44 Ubuntu20 tomcat9[27524]: Server initialization in [2,599] millisecond
nov 30 23:26:44 Ubuntu20 tomcat9[27524]: Starting service [Catalina]
nov 30 23:26:44 Ubuntu20 tomcat9[27524]: Starting Servlet engine: [Apache Tomcat/9.0
nov 30 23:26:44 Ubuntu20 tomcat9[27524]: Deploying web application directory [/var/l
nov 30 23:26:47 Ubuntu20 tomcat9[27524]: At least one JAR was scanned for TLDs yet n
```

O servidor web faz parte dos serviços que o seu sistema corre. Pode usar vários comandos para controlar o estado de um serviço (no nosso caso o nome do serviço é `tomcat9.service`):

- `sudo systemctl status <nome do serviço>` informa sobre o estado do serviço, se está a correr ou não.
- `sudo systemctl start <nome do serviço>` arranca o serviço.
- `sudo systemctl stop <nome do serviço>` para o serviço.
- `sudo systemctl restart <nome do serviço>` para o serviço e arranca outra vez.
- `sudo systemctl enable <nome do serviço>` configura o serviço de forma a arrancar automaticamente cada vez que o computador arranca.
- `sudo systemctl disable <nome do serviço>` é a operação inversa.

No caso do Mac os comandos são semelhantes: `brew services status tomcat@9`, `brew services start tomcat@9`, `brew services stop tomcat@9`, `brew services restart tomcat@9`, `brew services enable tomcat@9` e `brew services disable tomcat@9`.

A seguir verifique que o seu browser acede ao servidor usando o endereço `localhost:8080`:



O Tomcat disponibiliza uma aplicação para fazer a gestão das aplicações WEB que correm no servidor. Esta aplicação (**manager**) é uma aplicação WEB que corre no próprio Tomcat. Está disponível num pacote que precisa de ser instalado. Use o comando seguinte para a instalar bem como a documentação do Tomcat:

```
sudo apt-get install tomcat9-admin tomcat9-docs
```

No caso do Mac, esses pacotes estão instalados com o pacote tomcat@9.

Deve configurar o servidor para poder usar o "manager".

Altere o conteúdo do ficheiro /etc/tomcat9/tomcat-users.xml (deve usar o comando sudo nano /etc/tomcat9/tomcat-users.xml). E incluir as linhas seguintes:

```
<role rolename="admin-gui" />
<role rolename="manager-gui" />
<user username="manuel" password="s3cret" roles="admin-gui,manager-gui" />
```

copy

Não se esqueça de substituir "manuel" e "s3cret" por valores adequados. No Mac, este ficheiro encontra-se em /etc/tomcat@9/tomcat-users.xml. A seguir arranca de novo o servidor com o comando :

```
sudo systemctl restart tomcat9.service
```

ou

```
brew service restart tomcat@9
```

no Mac. Pode aceder à aplicação via o endereço `localhost:8080/manager/`. Terá que entrar as credências indicadas no ficheiro `tomcat-users.xml`.

The screenshot shows the Tomcat Web Application Manager interface. At the top, there's a logo of a cat and the Apache Software Foundation logo. Below that is the title "Tomcat Web Application Manager". A message box says "Message: OK". A navigation bar at the top has tabs for "Manager", "List Applications", "HTML Manager Help", "Manager Help", and "Server Status". The main area is titled "Applications" and contains a table with four rows. The columns are "Path", "Version", "Display Name", "Running", "Sessions", and "Commands". The rows show the following data:

| Path          | Version        | Display Name                    | Running | Sessions | Commands                                                                 |
|---------------|----------------|---------------------------------|---------|----------|--------------------------------------------------------------------------|
| /             | None specified |                                 | true    | 0        | Start Stop Reload Undeploy<br>(Expire sessions) with idle ≥ [30] minutes |
| /docs         | None specified | Tomcat Documentation            | true    | 0        | Start Stop Reload Undeploy<br>(Expire sessions) with idle ≥ [30] minutes |
| /host-manager | None specified | Tomcat Host Manager Application | true    | 0        | Start Stop Reload Undeploy<br>(Expire sessions) with idle ≥ [30] minutes |
| /manager      | None specified | Tomcat Manager Application      | true    | 1        | Start Stop Reload Undeploy<br>(Expire sessions) with idle ≥ [30] minutes |

At the bottom, there's a "Deploy" button.

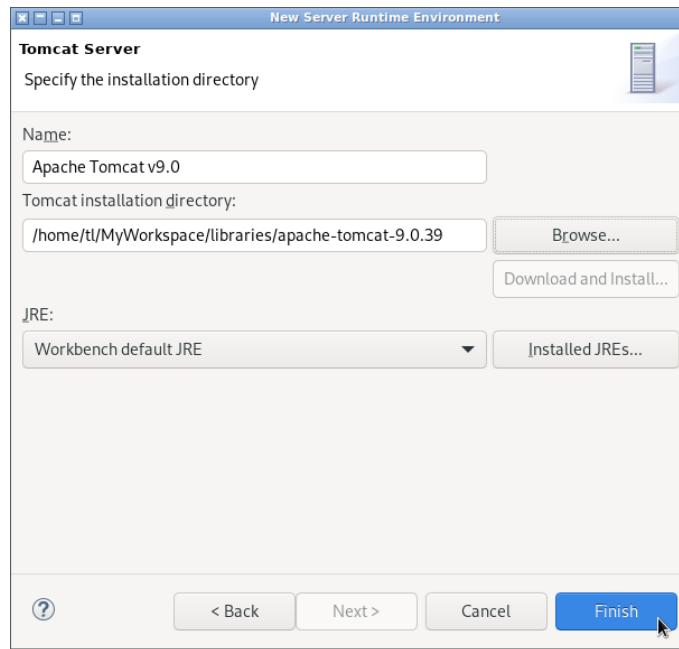
Pode ver que o “manager” é uma das aplicações WEB instaladas no servidor.

### 6.9.2 Instalação nos computadores dos labs

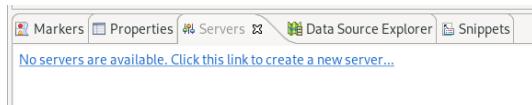
Não tem os privilégios suficientes para instalar novos pacotes nos computadores dos laboratórios. A solução consiste em instala-lo localmente e associa-lo ao eclipse. Assume-se que o workspace está na raiz da sua área: `/home/manuel/MyWorkspace`. As bibliotecas devem estar em `/home/manuel/MyWorkspace/libraries/` (se a pasta não existir, cria-la). No contexto do eclipse, o papel do workspace é agrupar configurações e projetos. No entanto, um projeto pode estar associado ao seu workspace sem estar na pasta `/home/manuel/MyWorkspace`. Na maioria dos casos é vantajoso ter o projeto fora da pasta Workspace. Na pasta `/home/manuel/MyWorkspace/libraries/` descompactar o arquivo `apache-tomcat-9.0.39.zip`. Em Windows > Preferences > Server > Runtime Environments > clicar em Add... e escolher **Apache Tomcat v9.0**.

The screenshot shows the Eclipse Preferences dialog for "Server Runtime Environments". On the left, there's a sidebar with various options like JavaScript, JSON, Language Servers, Maven, Mylyn, Oomph, Plug-in Development, Remote Systems, Run/Debug, Server, Runtime Environment, SonarLint, and Team. The "Runtime Environment" option is selected. The main pane shows a table of server runtime environments with columns "Name" and "Type". There are buttons for "Add...", "Edit...", "Remove", "Search...", and "Columns...". To the right, a modal window titled "New Server Runtime Environment" is open. It has a title "New Server Runtime Environment" and a sub-instruction "Define a new server runtime environment". It asks "Select the type of runtime environment:" and lists "Apache Tomcat v9.0" as the selected option. Other options shown are Apache Tomcat v6.0, Apache Tomcat v7.0, Apache Tomcat v8.0, Apache Tomcat v8.5, and Apache Tomcat v9.0. Below the list, it says "Apache Tomcat v9.0 supports J2EE 1.2, 1.3, 1.4, and Java EE 5, 6, 7, and 8 Web modules." There's also a checkbox for "Create a new local server". At the bottom of the modal are buttons for "?", "Cancel", "Apply and Close", "< Back", "Next >", "Cancel", and "Finish".

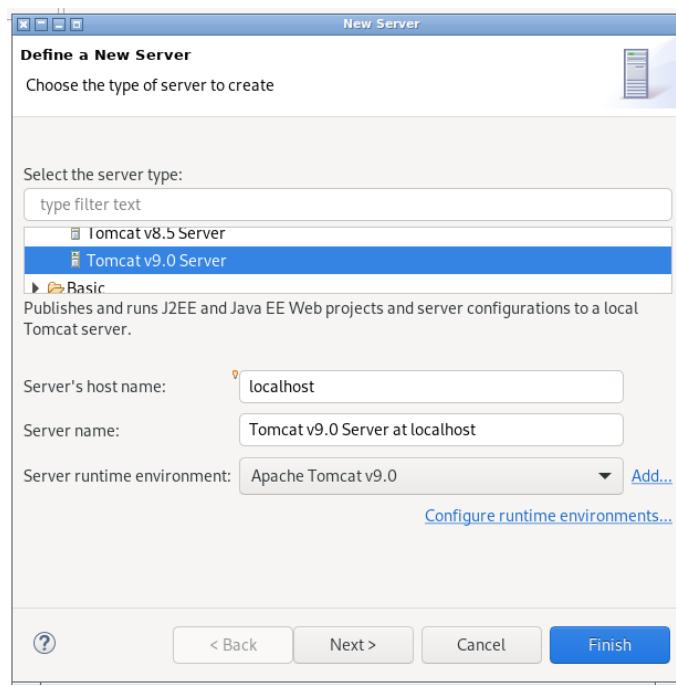
A seguir, clicar em **Next >**. Na janela que aparece, selecionar a pasta onde está a biblioteca (`/home/manuel/MyWorkspace/libraries/apache-tomcat-9.0.39`). Clicar em **finish**.



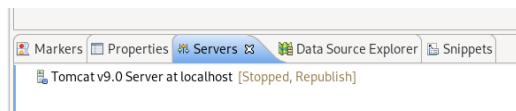
No tab **Servers** não está ainda nenhum servidor configurado (como indicado na imagem seguinte). Clique no link para criar um servidor. Se o tab Server não aparecer na sua interface, procure em **Window > Show view > Other > .**



Para definir o servidor basta escolher a versão (9.0) e clicar em **Finish**.



O servido está instalado:



Para usar este servidor, deve escolher a opção **Run As > Run on Server >** quando clique com o botão direito no projeto. O Eclipse vai pedir-lhe para escolher o servidor (deve haver apenas um disponível) e a seguir vai arrancar um browser incluído no eclipse e mostar a página da sua aplicação.

### 6.9.3 WEB app HelloWorld

O objetivo neste passo é criar uma aplicação WEB estática, sem código Java, para entender a estrutura de uma aplicação WEB e aprender a usar o Tomcat. Primeiro deve criar, como nos guiões anteriores um repositório no servidor `git.alunos.di.fc.ul.pt`. Chame o seu repositório “`guiautomcat`”.

1. Crie neste repositório um projeto de tipo Maven chamado “**HelloWorldWEB**”. Ao criar o projeto deve escolher :

- Group Id: `fcul.css`
- Artefact Id: `HelloWorldWEB`
- **Packaging: war**

2. No projeto, acrescente as pastas `src/main/webapp` e `src/main/webapp/WEB-INF`
3. Na pasta `src/main/webapp/WEB-INF`, criar um ficheiro `web.xml` com o seguinte conteúdo:

```
<?xml version="1.0" encoding="utf-8"?>
<web-app id="WebApp_ID"
 version="2.5"
 xmlns="http://java.sun.com/xml/ns/javaee"
 xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
 <display-name>Hello World Web Application</display-name>
</web-app>
```

**copy**

4. Na pasta `src/main/webapp` criar um ficheiro `index.jsp` com o seguinte conteúdo:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
 pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>
 <h2>Hello World!</h2>
</body>
</html>
```

**copy**

Acabou de escrever uma aplicação WEB estática minimalista. O passo final consiste em instalar a aplicação no servidor mas antes é necessário empacotá-la.

5. Abre um terminal e corre os comandos:

```
mvn clean
mvn package
```

Deve observar no terminal algo semelhante a:

```
[INFO] Packaging webapp
[INFO] Assembling webapp [HelloWorldWEB] in [/home/tl/tmp/TomcatTest/HelloworldWEB/target/HelloworldWEB-0.0.1-SNAPSHOT]
[INFO] Processing war project
[INFO] Copying webapp resources [/home/tl/tmp/TomcatTest/HelloworldWEB/src/main/webapp]
[INFO] Webapp assembled in [40 msecs]
[INFO] Building war: /home/tl/tmp/TomcatTest/HelloworldWEB/target/HelloworldWEB-0.0.1-SNAPSHOT.war
[INFO] WEB-INF/web.xml already added, skipping
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 1.303 s
[INFO] Finished at: 2021-12-01T14:30:00Z
[INFO] -----
```

Note que o Maven vai fazer download de bibliotecas, pode demorar um pouco. Na pasta `target` do projeto deve encontrar um ficheiro `HelloWorldWEB-0.0.1-SNAPSHOT.war` que contém a aplicação WEB. Caso estiver a usar um Mac e obtiver um erro neste passo, acrescente a seguinte plugin na secção “plugins” do seu `pom.xml`:

```
<plugins>
 <plugin>
 <groupId>org.apache.maven.plugins</groupId>
 <artifactId>maven-war-plugin</artifactId>
 <version>3.3.1</version>
 </plugin>
</plugins>
```

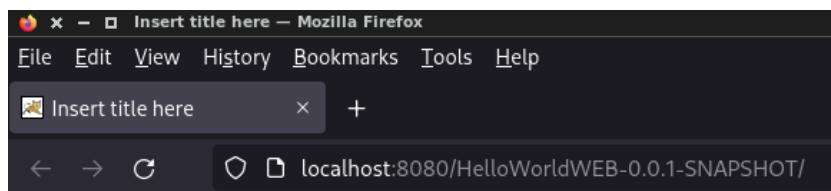
copy

- O passo seguinte consiste em instalar a aplicação no servidor. Pode ser feito de várias maneiras. Usando o manager, na secção “WAR file to deploy”, clicar em browse e selecionar o ficheiro `HelloWorldWEB-0.0.1-SNAPSHOT.war`. Clicar em “deploy”. A aplicação aparece na tabela:

|                                                     |                             |                             |      |   |                                                                                |
|-----------------------------------------------------|-----------------------------|-----------------------------|------|---|--------------------------------------------------------------------------------|
| <code>/TomcatTest1-0.0.1-SNAPSHOT</code>            | <code>None specified</code> | Hello World Web Application | true | 0 | <code>Start</code> <code>Stop</code> <code>Reload</code> <code>Undeploy</code> |
| <code>Expire sessions with idle ≥ 30 minutes</code> |                             |                             |      |   |                                                                                |

Esta operação é chamada “deployment” ou, mais simplesmente, instalação no servidor.

- Finalmente, para experimentar a aplicação, basta indicar o endereço `localhost:8080/HelloWorldWEB-0.0.1-SNAPSHOT`. O resultado é impresionante:



**Hello World!**

commit & push

#### 6.9.4 Modificação e «redeployment»

Experimentar modificar o ficheiro `index.jsp` para afixar na página “Olá CSS !” em vez de “Hello World !”. Se fizer agora reload da página não ver nenhuma alteração. Se correr novamente o comando `mvn package` uma nova versão do ficheiro `.war` será criada na pasta `target` mas ao fazer `reload` da página não verá as suas modificações. Para ver os efeitos da sua modificação deve, na página do manager, fazer `undeploy`.

Para evitar a repetição deste processo cada vez que faz uma modificação a sua aplicação, pode-se optar por efetuar o “redeployment” na linha de comando. A operação consiste em copiar o projecto empacotado (ficheiro `.war`) para a pasta “webapps” do Tomcat. Se instalou o Tomcat seguindo as instruções de instalação no Ubuntu, a pasta para as aplicações WEB é `/var/lib/tomcat9/webapps/`. É necessário pertencer ao grupo de utilizadores “tomcat” para poder escrever para esta pasta (o grupo ficou criado pelo processo de instalação do Tomcat). O comando que vai juntar a sua conta ao grupo é :

```
sudo gpasswd -a alfredo tomcat
```

... substituído alfredo pelo nome de utilizador adequado. A seguir é necessário fazer logout e entrar novamente para a modificação ter efeito.

Agora pode-se instalar uma aplicação no Tomcat copiando o ficheiro `war` do projeto para a pasta `/var/lib/tomcat9/webapps/`. Em alternativa pode copiar as linhas seguintes num ficheiro `setup.sh`:

```
#!/bin/bash
alias deploy='mvn package; cp target/*.war /var/lib/tomcat9/webapps/'
```

Este ficheiro quando executado com o comando

```
source setup.sh
```

define um comando **deploy** que recompila, empacota e instala a sua aplicação WEB no servidor Tomcat. Pode verificar que uma aplicação está instalada consultando a lista das aplicações no “manager”.

### 6.9.5 Santa Claus Servlet

O objetivo desta secção é realizar uma primeira aplicação WEB usando Servlets. A aplicação (gerida pelo Pai Natal) serve para registar uma lista de prendas desejadas. É composta de duas páginas:

- **SantaClausServlet**: É responsável por recolher os desejos de um utilizador (um de cada vez). Cada vez que um novo desejo é submetido, o sistema armazena-o numa lista e mostra novamente o formulário para recolher o desejo seguinte. Desde que a lista de desejos não está vazia, o seu conteúdo é mostrado no ecrã.
- **TooManyGifts**: É um Servlet que é usado quando a lista de desejos é demasiada longa (> 4 desejos). Neste caso o Servlet mostra uma mensagem a explicar a situação e um link que quando acionado volta à página inicial. Nest caso a lista de desejos é esvaziada.

#### Projeto SantaClausServlet

No mesmo repositório ([guiaotomcat](#)) crie um projeto Maven chamado “**SantaClausServlet**”, com group Id **fcul.css**, Artefact Id **SantClausServlet**. Não se esqueça de escolher **war** como modo de empacotamento. No projeto, acrescente as pastas **src/main/webapp** e **src/main/webapp/WEB-INF**. Na pasta **src/main/webapp/WEB-INF**, criar um ficheiro **web.xml** com o seguinte conteúdo:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app id="WebApp_ID"
 version="2.5"
 xmlns="http://java.sun.com/xml/ns/javaee"
 xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/ja
 <display-name>Santa Claus Web Application</display-name>
</web-app>
```

**copy**

No ficheiro **pom.xml** acrescente a seguinte dependência :

```
<dependency>
 <groupId>javax.servlet</groupId>
 <artifactId>javax.servlet-api</artifactId>
 <version>4.0.1</version>
 <scope>provided</scope>
</dependency>
```

**copy**

e a propriedade seguinte:

```
<properties>
 <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
 <maven.compiler.source>11</maven.compiler.source>
 <maven.compiler.target>11</maven.compiler.target>
</properties>
```

**copy**

Criar um pacote **santaclaus** e define neste pacote uma classe de tipo Servlet com nome **SantaClausServlet** (use **New > Other > Web > Servlet** para o efeito). A classe gerada vem com algum código já definido nomeadamente um esqueleto para os métodos **doGet** e **doPost**. Pode observar que o ficheiro **web.xml** foi modificado para incluir a definição do Servlet :

```
<servlet>
 <description></description>
 <display-name>SantaClausServlet</display-name>
 <servlet-name>SantaClausServlet</servlet-name>
 <servlet-class>santaclaus.SantaClausServlet</servlet-class>
</servlet>
<servlet-mapping>
 <servlet-name>SantaClausServlet</servlet-name>
 <url-pattern>/SantaClausServlet</url-pattern>
</servlet-mapping>
```

**copy**

Nesta definição, um elemento util é o **url-pattern** que define o url que permite aceder ao servlet. Por exemplo pode ecurtar o endereço substituindo a definição por :

```
<url-pattern>/SantaClaus</url-pattern>
```

**copy**

Nesta fase, o eclipse pode ainda estar a assinalar erros no seu projeto (pode ser causado pela versão do Java que está instalada no seu computador. Se for o caso, modifique a versão 11 -> 1.8 no **pom.xml**). É altura para experimentar compilar empacotar e instalar o projeto no servidor. Se seguiu as instruções na secção anterior, basta fazer:

**deploy**

A seguir pode aceder à aplicação usando no seu browser o endereço <http://localhost:8080/SantaClausServlet-0.0.1-SNAPSHOT/SantaClaus> (se optou por encurtar o url). Deve aparecer no browser:

Served at: /SantaClausServlet-0.0.1-SNAPSHOT

## Definição do primeiro Servlet

Pode começar por substituir o esqueleto da classe pelo código seguinte que tem a vantagem de tartar o pedido num só método que o método seja GET ou POST:

```
public class SantaClausServlet extends HttpServlet {

 /**
 * Processes requests for both HTTP <code>GET</code> and <code>POST</code>
 * methods.
 *
 * @param request servlet request
 * @param response servlet response
 * @throws ServletException if a servlet-specific error occurs
 * @throws IOException if an I/O error occurs
 */
 protected void processRequest(HttpServletRequest request, HttpServletResponse response)
 throws ServletException, IOException {
 response.setContentType("text/html;charset=UTF-8");
 // TODO ****
 }

 /**
 * Handles the HTTP <code>GET</code> method.
 *
```

```

 * @param request servlet request
 * @param response servlet response
 * @throws ServletException if a servlet-specific error occurs
 * @throws IOException if an I/O error occurs
 */
@Override
protected void doGet(HttpServletRequest request, HttpServletResponse response)
 throws ServletException, IOException {
 processRequest(request, response);
}
/**
 * Handles the HTTP <code>POST</code> method.
 *
 * @param request servlet request
 * @param response servlet response
 * @throws ServletException if a servlet-specific error occurs
 * @throws IOException if an I/O error occurs
 */
@Override
protected void doPost(HttpServletRequest request, HttpServletResponse response)
 throws ServletException, IOException {
 processRequest(request, response);
}
}

copy

```

O método `processRequest` deve ser completado de forma a :

1. produzir um formulário como na imagem seguinte:

**Indique aqui a prenda desejada:**

A small screenshot of a web form. It contains a single text input field with a placeholder text 'Indique aqui a prenda desejada:' and a single 'Submit' button below it.

2. associar ao formulário a acção “SantaClaus” que corresponde ao URL do próprio Servlet. A consequência é que quando o utilizador carregará no botão Submit, o pedido (com o formulário preenchido) será dirigido para o mesmo Servlet. Ao tag <input> deve associar um nome que corresponderá ao nome do parâmetro que vai conter o valor correspondente.
3. Verifique que depois de submeter o formulário, a página aparece novamente.
4. Modifique o Servlet de maneira a mostrar o desejo indicado pelo utilizador após submissão do formulário.  
Os valores dos campos de um formulário são associados aos parâmetros do objeto `request`. Encontrará como usar o método `getParameter` na documentação da classe `HttpServletRequest`.

Nesta fase, o Servlet mostra o valor que foi indicado no campo do formulário cada vez que é submetido. É preciso agora armazenar os valores numa lista.

### Âmbitos

Em qualquer aplicação WEB é um passo essencial perceber qual será o âmbito dos valores a manter em memória.

Cada vez que o utilizador carregue no botão Submit do formulário, um novo pedido (`request`) é criado. A lista dos desejos não pode ser guardada neste contexto. Uma solução consiste em associar o valor à sessão.

Para usar uma sessão, é preciso :

1. ciar a sessão,

2. armazenar o valor atribuindo um nome.

Os métodos necessários estão documentados nas classes `HttpServletRequest` e `HttpSession`. Em particular os métodos `getSession` (classe `HttpServletRequest`), `setAttribute` e `getAttribute` (classe `HttpSession`).

Uma vez que a lista de prendas desejadas está armazenada, podemos mostra-la ao utilizador. É aconselhado escrever um método separado `private String giftTable(List<String> list)` que será responsável por produzir o HTML onde a lista de prendas será mostrada numa tabela. Quando a lista estiver vazia, a aplicação deve mostrar:

**Santa Claus WEB App**

A sua lista de prendas está vazia.

**Indique aqui a prenda desejada:**

A medida que vão ser acrescentadas prendas, a lista é mostrada:

**Santa Claus WEB App**

**A sua lista de prendas:**

- (1) iphone 3
- (2) prancha de surf
- (3) fato de mergulho

**Indique aqui a prenda desejada:**

## O Servlet TooManyGifts

Em tempos de crise, o Pai Natal está desejoso de impor alguns limites. A lista de desejos não pode ser infinita. A ideia é adicionar um Servlet que mostrará um aviso indicando o excesso de prendas quando o utilizador ultrapassar quatro prendas. Vamos usar para isso o reencaminhamento entre Servlets. No que toca o Servlet `TooManyGifts` a página é composta de uma mensagem e de um link que permite regressar à página inicial:

**You have too many gifts in your list !!!**

[Back](#)

O reencaminhamento é obtido usando um `RequestDispatcher`:

```
ServletContext sc = getServletContext();
RequestDispatcher r = sc.getRequestDispatcher("/TooManyGifts");
r.forward(request, response);
copy
```

## O Pai Natal é estiloso

Pode decorar as suas páginas usando folhas de estilo CSS. Tem de incluir o elemento :

```
<link rel='stylesheet' type='text/css' href='styles.css' />
```

**copy**

Na página gerada e colocar o ficheiro `style.css` na pasta `src/main/webapp/`. Se usar o seguinte conteúdo:

```
body {
 background-color: #cc0033;
}
h1 {
 font-family: Arial, Helvetica, sans-serif;
 font-size: 25pt;
 color: #ffffff;
}
h2, a {
 font-family: Arial, Helvetica, sans-serif;
 font-size: 20pt;
 color: #ffffff;
 margin-top: 0px;
}
div {
 border-radius: 7px;
 border-color: #ffffff;
 border-style: solid;
 border-width: 1px;
 background-color: #990033;
 padding-left: 20px;
 width: 70%;
 margin-right: auto;
 margin-left: auto;
 margin-top: 0px;
 margin-bottom: 30px;
 color: #ffffff;
 font-family: Arial, Helvetica, sans-serif;
 font-size: 15pt;
}
```

**copy**

a aplicação terá este lindo aspecto:



## 6.9.6 SantaClausJSP

O objetivo desta secção é fazer novamente a mesma aplicação usando a tecnologia JSP.

1. No mesmo repositório criar um projeto Maven chamado `SantaClausJSP`.
2. Acrescentar a dependência e a propriedade que colocou no ficheiro `pom.xml` do projeto anterior.

3. Criar as pastas `src/main/webapp/` e `src/main/webapp/WEB-INF/`.
4. Na pasta `src/main/webapp/` criar um ficheiro `santaclaus.jsp` usando: **New > Other... > Web > JSP File >**
5. Na mesma pasta criar um ficheiro `toomanygifts.jsp`.

Aos dois ficheiros `.jsp` vão corresponder dois Servlets que vão fazer exatamente as mesmas operações do que no projeto anterior. A diferença é que se vai escrever menos código Java, as classes Servlets serão geradas automaticamente pelo Web Container. Pode-se ver um ficheiro `.jsp` como uma página html com um pouco de Java à mistura.

Pode começar pelo ficheiro `santaclaus.jsp`. Escreve o código HTML que corresponde à página principal da aplicação, incluído em primeiro formulário. A parâmetro “action” do tag `<form>` pode ser o nome no ficheiro `.jsp` que vai processar o pedido ou um nome escolhido por si, por exemplo `SantaClaus`. A seguir é necessário criar o ficheiro `web.xml`, na pasta `src/main/webapp/WEB-INF/`:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app id="WebApp_ID" version="2.5"
 xmlns="http://java.sun.com/xml/ns/javaee"
 xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/ja
<display-name>Santa Claus Web/JSP Application</display-name>
<servlet>
 <servlet-name>SantaClaus</servlet-name>
 <jsp-file>/santaclaus.jsp</jsp-file>
</servlet>
<servlet-mapping>
 <servlet-name>SantaClaus</servlet-name>
 <url-pattern>/SantaClaus</url-pattern>
</servlet-mapping>
</web-app>
```

**copy**

- Note a definição do Servlet que corresponde ao ficheiro `.jsp`.
- Note igualmente o carácter `/em` : `<jsp-file>/santaclaus.jsp</jsp-file>` bem como no tag `<url-pattern>`. Sem este carácter a aplicação não vai funcionar.
- Quando referenciar o Servlet pode-se usar o `url-pattern` ou o `jsp-file`.

Verificar que a página funciona i.e. quando carregar no botão Submit deve voltar a mostrar a mesma página.

O passo seguinte consiste em implementar a lista de prendas como foi feito no projeto anterior. Para isso deve incluir excertos de código Java no HTML. Pode usar as variáveis que existem em todos os servlet/JSP em particular a variável `session` que contém sessão corrente (não é preciso cria-la). Por exemplo, para adicionar o atributo que contém a lista pode-se usar:

```
<%
List<String> list = (List<String>) session.getAttribute("giftlist");
if (list == null) {
 list = new ArrayList<>();
 session.setAttribute("giftlist", list);
}
%>
```

**copy**

Quando é necessário incluir condicionalmente código HTML, pode usar o esquema seguinte:

```
<%
if (list.isEmpty()) {
%>
 <h2>A sua lista de prendas está vazia.</h2>
```

```
<%
} else {
%>
<div><h2>A sua lista de prendas:</h2>
<table>
```

**copy**

Alternando código Java e HTML pode inserir as instruções que vão gerar a tabela.

Uma vez que verificou que a lista de prendas é apresentada pelo Servlet, basta acrescentar instruções para passar o controlo para o Servlet `toomanygifts.jsp` quando a lista é demasiada longa. A escrita do segundo servlet segue os mesmos passos que o primeiro e não apresenta dificuldades.

commit & push

### 6.9.7 One more thing

As bibliotecas para realizar aplicações WEB incluem todas facilidades para gerar HTML. De um lado espectro temos os Servlets definidos como classes Java que incluem instruções para compor a página HTML a custa de concatenações de Strings. Um exemplo desta abordagem é o projeto `SantaClausServlet` neste guião. A seguir temos a tecnologia JSP onde a abordagem consistem em descrever a página HTML e embutir usando tags especias (`<%`, `<=%` etc...) exertos de código Java (é a abordagem usada no projeto `SantaClausJSP`). No sentido de eliminar cada vez mais código (Java) da definição das páginas, pode-se usar bibliotecas de tags (JSTL) que permitem fazer pseudo-programação com tags que implementam instruções tipo `if`, `switch`, `for-each` etc... (esta abordagem não foi explorada no guião).

Uma abordagem alternativa (mais dirigida para os programadores) consiste em fazer desaparecer tanto como possível o HTML do Servlet programado em Java. Em vez de usar uma instância de `PrintWriter` (ou `StringBuilders`) onde se vai acumulando strings que representam a página HTML, usa-se uma biblioteca especializada na geração do HTML. Uma dessas bibliotecas é o `J2HTML` que pode ser usada via uma dependência no ficheiro `pom.xml`:

```
<dependency>
 <groupId>com.j2html</groupId>
 <artifactId>j2html</artifactId>
 <version>1.4.0</version>
</dependency>
```

**copy**

Por exemplo no Servlet `SantaClausServlet` pode-se escrever :

```
sb.append("<!DOCTYPE html>")
.append(join(html(head(title("Servlet «Santa Claus»")),
body(h1("Santa Claus WEB App"),
giftTable(list),
addToWishListForm()))).render();
```

**copy**

Os métodos que correspondem aos tags HTML fazem parte da biblioteca (`import static j2html.TagCreator.`). Retornem objetos de tipo `ContainerTag` que podem ser agrupados numa sequência com o método `join`. O método `render` transforma um `ContainerTag` em código HTML (`String`). Os métodos `giftTable` e `addToWishListForm` são usado para gerar partes da página. Torna-se assim facil dividir o código que trata da geração do HTML em vários métodos. No contexto de uma aplicação mais sofisticada, é facil imaginar como reaproveitar (ou não repetir) o código que gera elementos das páginas (`headers`, `footers` etc...)

O método `addToWishListForm` pode ser escrito assim:

```
private ContainerTag addToWishListForm() {
 return div(form(h2("Indique aqui a prenda desejada:"),
input().withType("text").withId("gift").withName("gift"),
input().withType("submit").withValue("Submit"))
```

```
 .withAction("SantaClaus").withMethod("get"));
}
```

copy

E o método giftTable assim:

```
private ContainerTag giftTable(List<String> list) {
 return (list.isEmpty()
 ? h2("A sua lista de prendas está vazia.")
 : div(h2("A sua lista de prendas:"),

 table(each(list, g -> tr(td(g))))));
}
```

copy



# Capítulo 7

## Project Memories

Recebeu de um familiar um disco contendo milhares de fotografias. Essas fotografias representam memórias de uma vida. O objetivo deste projeto é criar uma aplicação que permite classificar fotografias digitais. O meio principal usado para classificar é o uso de etiquetas ou tags. A aplicação deve permitir atribuir um número arbitrário de tags a cada fotografia. Por omissão uma fotografia terá sempre pelo menos três tags que correspondem a data na qual foi feita a fotografia. O outro meio usado para organizar as fotografias é a criação de albums. Os tags não são apenas palavras, têm associado um conjunto de atributos.



Primeira fase: MemoriesDB p.[185](#) • Segunda fase: MemoriesJFX p.[193](#)

### 7.1 Primeira fase: MemoriesDB

Nas secções seguintes estão descritos mais em detalhes os conceitos da aplicação e as tarefas a desenvolver.

Fotografias p.[185](#) • Tags p.[186](#) • Albums p.[186](#) • Primeiro passo p.[186](#)  
• Modelo de dados p.[187](#) • Modelo de domínio p.[187](#) • Implementação p.[187](#) • Testes p.[190](#)

#### 7.1.1 Fotografias

A aplicação permite importar uma pasta que contém fotografias (formato JPEG apenas). São igualmente incluídas todas as fotografias em sub-pastas e sub-sub-pastas etc... sem limite no número de níveis. A importação consiste em copiar as fotografias para uma hierarquia de pastas definida pela

data da fotografia: YYYY/MM/DD. Por exemplo, uma fotografia tirada no dia 23 de março de 2010 será colocada na pasta 2010/03/23/. As datas são obtidas via a leitura dos metadados EXIF associados à fotografia pela máquina fotografica. Caso não existe metadata, a data atribuida será 2000/01/01 e o tag "noExif" será atribuido à foto.

Do ponto de vista da aplicação, uma fotografia será representada pelo nome do ficheiro, incluindo a pasta. Por exemplo: /2010/03/12/IMG\_827364782634.jpg. Os pixels da imagem não fazem parte da base de dados.

### 7.1.2 Tags

Um tag representa uma categoria a qual uma fotografia pode pertencer ou não. É um meio de classificação muito flexivel que permite muito esquemas de arrumação diferentes. Na altura da sua importação serão atribuidos pelo menos três tags à foto: yYYYY, mMM e dDD onde YYYY, MM e DD correspondem aos anos, meses e dia da fotografia.

O utilizador pode criar os tags que quiser, sem restrição.

Uma query consiste num conjunto de tags. Dois tipos de pedidos podem ser feitos usando os tags:

- procurar todas as fotografias que têm todos os tags da query,
- procurar todas as fotografias que têm pelo menos um tag da query.

Por exemplo, se alguém nasceu no dia 13/06/2002, será facil obter todas as fotografias tiradas nos seus dias de anos usando uma query composta dos tags "m06" e "d13".

Admitindo que as fotografias foram etiquetadas, é possível ser mais especifico usando a query composta dos tags {m06, d13, marina}, para encontrar as fotografias da Marina tiradas no seu dia de anos.

Pensando no aspeto ergonomico e estetico da aplicação, será possível atribuir uma cor (dentro de um conjunto de cores predefinido) a cada tag.

### 7.1.3 Albums

Um album não é mais do que um conjunto de fotografias ordenadas. O primeiro passo para a criação de um album é a criação de um tag específico ao album, por exemplo **Natal2014**. A seguir o utilizador atribuirá uma ordem ás fotografias selecionadas. O album tem para além do tag que o define e a sequência de fotografias, um nome (String) e uma descrição (String).

A aplicação permite visualizar um album.

### 7.1.4 Primeiro passo

O primeiro passo para a realização do projeto consiste em

1. criar um repositório chamado **MemoriesDB**. Tem de haver apenas um repositório por grupo.
2. Partilhar o repositório com o utilizador `css-lti-000`. Atribuir as permissões de "Developer". Tem de fazer este passo logo no início assim, caso seja necessário, poderei resolver dúvidas que podem surgir, com acesso ao código.
3. Ciar um ficheiro README.md onde indica o número do seu grupo e os nomes e números de aluno dos membros.
4. Fazer um primeiro commit.
5. Adicionar um ficheiro .gitignore onde impede a inclusão no repositório dos ficheiros gerados na compilação do Java e os ficheiros gerados pelo Eclipse (.metadata, .project etc...). Adicione as linhas seguintes:

```
*.jpg
*.jpeg
*.JPEG
*.JPG
*.pdf
```

para evitar que esses ficheiros estejam incluídos no repositório.

6. Fazer um segundo commit.
7. Criar 2 ou 3 ramos, um para cada membro do grupo.

### 7.1.5 Modelo de dados

Elabore um modelo de dados que permite suportar as funcionalidades da aplicação. Pode usar qualquer ferramenta para realizar o modelo (inclusive papel e caneta). O resultado deve ser um ficheiro imagem **ModeloDeDados.png** (com o formato PNG) colocado na raiz do repositório.

O ficheiro tem de estar no ramo principal (master ou main).

### 7.1.6 Modelo de domínio

Elabore um modelo de domínio para a aplicação. Pode usar qualquer ferramenta para realizar o modelo (inclusive papel e caneta). O resultado deve ser um ficheiro imagem **ModeloDominio.png** (com o formato PNG) colocado na raiz do repositório.

Uma vez que realizou os modelos de dados e de domínio, verifique que estão no repositório remoto e avise-me por email.

### 7.1.7 Implementação

Para implementar a camada de persistência deve:

1. Criar as classes do domínio conforme o seu modelo de domínio. Deve haver classes para representar as fotografias (**Picture**), tags (**Tag**) e albums (**Album**).
2. Implementar a camada de dados.

**Classes do diomínio, entidades p.187** • Camada de persistência p.188  
 • Importação de fotografias p.188 • Resumindo p.190

### Classes do diomínio, entidades

Segue os passos do guião JPA para criar as classes do domínio e transforma-las em entidades. Deve acrescentar as dependências seguintes ao ficheiro pom.xml:

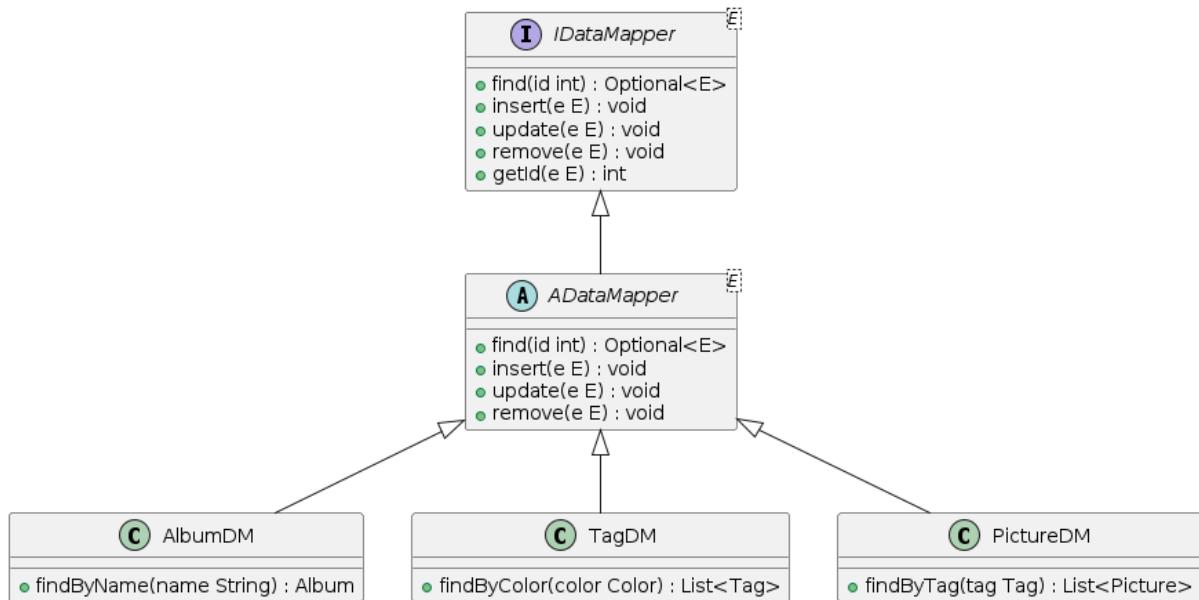
```
<dependency>
 <groupId>javax.annotation</groupId>
 <artifactId>javax.annotation-api</artifactId>
 <version>1.3.2</version>
</dependency>
<dependency>
 <groupId>org.eclipse.persistence</groupId>
 <artifactId>eclipselink</artifactId>
 <version>2.5.2</version>
</dependency>
<dependency>
 <groupId>org.eclipse.persistence</groupId>
 <artifactId>org.eclipse.persistence.jpa.modelgen.processor</artifactId>
 <version>2.5.2</version>
 <scope>provided</scope>
</dependency>
```

copy

Deve igualmente definir o acesso a base de dados no ficheiro `persistence.xml`. O nome da unidade de persistência deve ser **memoriesdb** sendo igual ao nome da base de dados. Como indicado p. 145. Note que a classe **Album** deve ter atributos para um nome e uma descrição, ambos de tipo **String**.

## Camada de persistência

A camada de persistência será implementada usando JPA e o padrão **Data Mapper**. No pacote `persist` define os seguintes elementos :



- `IDatamapper` é uma interface genérica.
- `ADatamapper` é uma classe que implementa parcialmente a interface de forma genérica.
- As classes `xxxDM` são mappers específicos para cada entidade. Será nessas classes que serão implementadas as queries específicas a cada entidade. Os métodos presentes nesta imagem são exemplos, será necessário acrescentar mais, a medida das necessidades.

O método `find` da classe `ADatamapper` vai recorrer ao método `find` do `EntityManager` (ver p. 121). Este método tem como assinatura `find(Class<T> entityClass, Object primaryKey)` portanto é necessário indicar à classe da entidade a recolher. O problema é que a classe `ADatamapper` é genérica ou seja não é conhecida a classe da entidade. Para resolver o problema vamos incluir na classe abstrata um atributo que vai conter a classe da entidade. Para o data mapper de tipo `AlbumDM` será `Album.class`, para mapper `PictureDM` será `Picture.class`.

O construtor da classe `ADatamapper` será :

```

protected ADatamapper(Class <E> theClass) {
 this.theClass = theClass;
}

```

copy

Enquanto nas subclasses o construtor será do tipo:

```

private AlbumDM() {
 super(Album.class);
}

```

copy

Além disso, as classes `xxxDM` terão que seguir o padrão **Singleton** dado que vai existir apenas um exemplar de cada data mapper.

## Importação de fotografias

É fornecido um conjunto de fotografias (`fotos.zip`). Para esta fase de desenvolvimento. Deve colocar a estrutura de pastas contida no zip numa pasta do seu computador (mas não no repositório). Esta pasta será conhecida do programa através da definição de uma constante. Todas as constantes relativas à aplicação serão definidas numa classe `ApplicationSettings` colcada no pacote `main`. Aqui está um esboço da classe:

```
public class ApplicationSettings {
 public static String importFolder = "a sua pasta com as fotos do zip";
 public static final String managedImagesFolder = "a pasta para as fotografias geridas pelo";
 public static final int ThumbnailHeight = 300;
}
```

copy

A constante `importFolder` corresponde à pasta onde estão as fotografias prontas para serem importadas e a constante `managedImagesFolder` corresponde a hierarquia de pastas onde as fotografias importadas estão armazenadas.

É necessário procurar todas as fotografias presentes na pasta de importação. Pode ser feito usando as instruções seguintes:

```
Stream<Path> pathStream = Files.find(Paths.get(ApplicationSettings.importFolder),
 999,
 (p, bfa) -> bfa.isRegularFile()
 && (p.toString().endsWith("jpg")
 || p.toString().endsWith("JPG")))
```

List<Path> pathList = pathStream.collect(Collectors.toList());

copy

Uma vez obtida a lista de fotografias, é preciso arruma-las na pasta `managedImagesFolder`. As fotografias são arrumadas numa hierarquia de pastas do tipo YYYY/MM/DD. Para obter a data onde foi tirada a fotografia, podemos ler a informação contida nos dados Exif do ficheiro. Existe uma biblioteca para fazer isso. Para usa-la no projeto basta acrescentar a seguinte dependência ao ficheiro `pom.xml` :

```
<dependency> <!-- EXIF -->
 <groupId>com.drewnoakes</groupId>
 <artifactId>metadata-extractor</artifactId>
 <version>2.18.0</version>
</dependency>
```

copy

O método seguinte lê a os metadados Exif:

```
private Metadata readExifMetadata() {
 try {
 return ImageMetadataReader.readMetadata(new File(this.originalPath));
 } catch (ImageProcessingException ex) {
 System.err.println(ex.getMessage());
 ex.printStackTrace();
 } catch (IOException ex) {
 System.err.println(ex.getMessage());
 ex.printStackTrace();
 }
 return null;
}
```

copy

Pode incluir este método na classe `Picture`. Neste exemplo, o valor de `this.originalPath` é o nome de ficheiro (e caminho) (String) de uma fotografia. Para extrair a informação desejada basta usar (e modificar) este método:

```
private void extractExifData(Metadata metadata) {
 Directory exifSubIFDDirectory = metadata.getFirstDirectoryOfType(ExifSubIFDDirectory);
 if (exifSubIFDDirectory != null) {
 this.date = exifSubIFDDirectory.getString(ExifSubIFDDirectory.TAG_DATETIME_ORIGINAL);
 System.out.println("DateTimeOriginal: " + date);
 } else {
 System.out.println("Exif data not found in the image.");
 }
}
```

```

 this.addTag("noExif");
}
}

```

**copy**

O valor do parâmetro deste método é o valor retornado pelo método `readExifMetadata`.

## Resumindo

Neste projeto devem desenvolver as classes/funcionalidades seguintes:

- classe `Picture`: construtor, setters, getters, `toString`, métodos para adicionar e remover tags. Anotações JPA.
- classe `Tag`: construtor, setters, getters, `toString`.
- classe `Album`: construtor, setters, getters, `toString`, métodos para permitir definir/alterar a ordem das fotografias.
- As classes do pacote persist. Os DM devem ser *Singletons*. Método para encontrar fotografias com um ou mais tags especificados (`PictureDM`).
- No pacote `utils` uma classe `ImportPictures` com um método (static) que tratam de importar as fotografias. Mais concretamente um método **com a seguinte assinatura: `public static void initializeDB()`** que trata de obter a lista de ficheiros de tipo imagem JPEG (contidos no ficheiro zip fornecido) e que usa em cada imagem :

```

for (Path p : jpeg) {
 Picture pic = Picture.importPicture(p.toString());
 // ...
}

```

**copy**

O método `importPicture` é um método static da classe `Picture` que trata de extrair a data, criar as pastas correspondentes (eventualmente), atribuir os tags relativos à data e copiar a imagem para esta pasta.

- No pacote `main`, uma classe `Main` que arranca aplicação (método `main`). Aqui está definido o entity manager factory (variável static).
- Testes: ver próxima secção.

### 7.1.8 Testes

O projeto **MemoriesDB** contém classes para a implementação das camadas de negócio e de persistência. Como ainda não existe a camada de apresentação, não é possível testá-la fazendo o papel do utilizador. A solução consiste em criar um conjunto de testes que vão verificar o bom funcionamento das camadas desenvolvidas.

Vamos usar uma versão mais recente do JUnit para conseguir fazer isso. A versão que deve ser usada é indicada na dependência seguinte que deve incluir no seu ficheiro `pom.xml`:

```

<dependency>
 <groupId>org.junit.jupiter</groupId>
 <artifactId>junit-jupiter-engine</artifactId>
 <version>5.8.1</version>
 <scope>test</scope>
</dependency>
<dependency>
 <groupId>org.junit.jupiter</groupId>
 <artifactId>junit-jupiter-api</artifactId>
 <version>5.8.1</version>
 <scope>test</scope>
</dependency>

```

[copy](#)

O JUnit5 introduz alguma anotações novas que são úteis para conseguir realizar o objetivo:

- **@BeforeAll** assinala um método de classe que vai ser executada antes de começar os testes da classe corrente. É útil para, por exemplo, criar o `EntityManagerFactory` e inicializar um atributo que contém uma instância de Data Mapper:

```
@BeforeAll
public static void setUpClass() {
 if (Main.emf == null || !Main.emf.isOpen()) {
 Main.emf = Persistence.createEntityManagerFactory("memoriesderbydb");
 }
 pictureDM4test.clearTable();
}
```

[copy](#)

- **@AfterAll** assinala um método de classe que será executado uma vez terminados os testes. Pode ser usado para, por exemplo, limpar a base de dados:

```
@AfterAll
static void afterAll() {
 tagDM.clearTable();
 albumDM.clearTable();
 // etc...
}
```

[copy](#)

- **@BeforeEach** assinala um método que deve ser executado antes de cada teste.
- **@AfterEach** assinala um método que deve ser executado após cada teste.
- **@Disabled** permite ignorar alguns testes. É útil se o teste não estiver ainda completamente funcional ou demorar demasiado tempo correr cada vez.
- **@Test** assinala um método de teste.
- **@Order(n)** permite atribuir uma ordem de execução aos testes. Esta opção pode ser usada se a anotação `@TestMethodOrder` estiver presente ao nível da classe:

```
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
class PictureDMTest {
 private static PictureDM pictureDM;
 private static TagDM tagDM;
 // etc...
```

[copy](#)

Os testes devem interagir com a base de dados e é útil controlar a ordem na qual são executados. Temos de saber exatamente o conteúdo da base de dados para efetuar os testes. Por exemplo, com uma base de dados inicialmente vazia, podemos verificar que não há nenhum Tag na BD com o teste seguinte:

```
/**
 * Test method for tagDM.findAllTags()
 */
@Test
@Order(2)
void testTagList1() {
 List<Tag> l = tagDM.findAllTags();
 assertEquals(l.size(), 0, "Should be 0.");
}
```

**copy**

O teste é valido apenas se as condições seguintes são verificadas:

1. a variável `tagDM` contém uma instância de `TagDM`
2. a classe `tagDM` tem um método `findAllTags` que retorna a lista de todos os tags registados
3. a tabela `Tag` está vazia.

Por essa razão é útil poder correr métodos antes e depois da sequência de testes e ordenar os testes.

## Refactoring

Para a avaliação do projeto serão usados mais testes (realizados por mim). Para que tudo funcione como esperado, é necessário haver uma uniformização das assinaturas dos métodos públicos da camada de persistência (para testar as queries) e da camada de negócio (principalmente os construtores) para poder criar objetos nos métodos de teste. Nesta secção estão descritas as assinaturas que devem constar nas classes do projeto. Têm toda a liberdade para acrescentar métodos com assinaturas diferentes mas essas deve estar porque podem fazer parte dos testes.

As classes para testar são em primeiro lugar os data mappers. Os métodos que devem existir são listados a seguir. Todos os DM devem ser singletons. Alguns melhoramentos foram feitos em relação à versão anterior. Tenha em conta a interface de cada método. Em cada classe pode acrescentar os métodos que julgar úteis para a realização do projeto.

**ADataMapper :**

- `public Optional<E> find(long id)` O tipo `Optional` é usado no retorno. A chave primária é de tipo `long`.
- `public long insert(E e)`
- `public void update(E e)`
- `public void remove(E e)`
- `public void clearTable():` opcional mas será útil.

**TagDM:**

- `public long getId(Tag t)`
- `public Tag makeTag(String name)`. Este método é mais importante do que parece. Um dos problemas encontrados quando usa-se uma base de dados é a criação de entidades duplicadas. Para evitar o problema, pode-se usar o esquema seguinte:
  - No data mapper, criar um método `makeTag` que verifica se existe uma entidade com este nome na BD. Se não existir, cria (usando um construtor sem argumentos), inicializa e persiste a entidade antes de a devolver. Se existir devolve a entidade da BD.
  - Na classe da entidade (`Tag`) tornar o construtor com argumento(s) privado. Criar um método `getTag` que chama o método do DM:

```
public static Tag getTag(String tagName) {
 return TagDM.getInstance().makeTag(tagName);
}
```

**copy**

Uma chamada a este método será usada cada vez que será preciso um `Tag`.

- `public long insert(Tag theTag)`. Antes de fazer a inserção (chamar o método da superclasse) verificar se existe um tag com este nome. Caso existir, não criar um novo.
- `public Optional<Tag> findByName(String testTag)`
- `public Set<Tag> findAllTags()`

**PictureDM:**

- `public long getId(Picture e)`
- `public Set<Picture> getRecentPictures()` este método será útil para a segunda fase. Mostra as fotografias do ano mais recente para o qual existem fotografias.

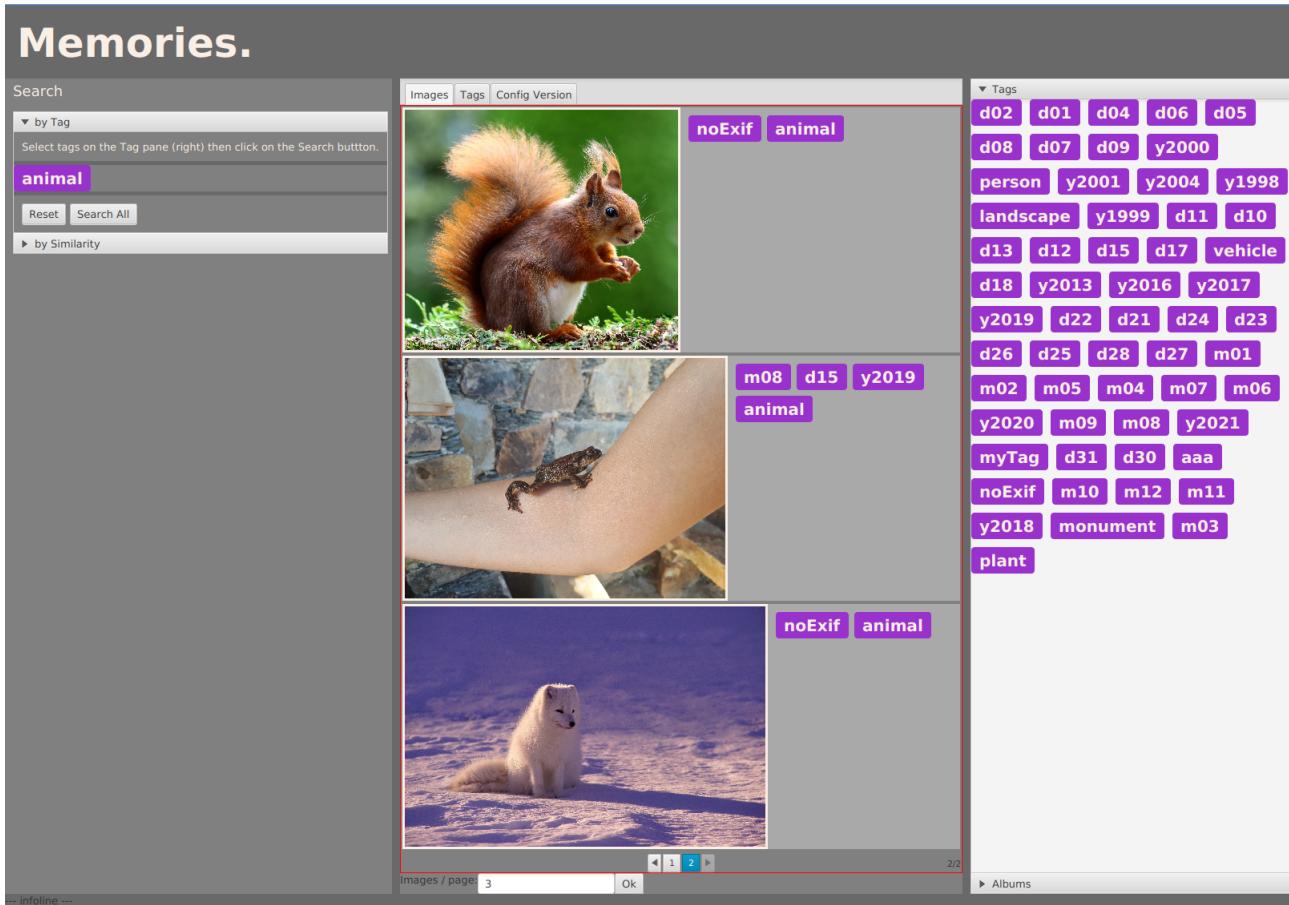
- public Set<Picture> findByTag(Tag tag) retorna as fotografias que têm o tag indicado.
- public Set<Picture> findByTags(Set<Tag> query) retorna as fotografias que têm TODOS os tags indicados. Isto pode ser feito com uma (e apenas uma) query JPQL. Pode procurar ajuda junto do ChatGPT mas deve indicar no **README.md** do seu projeto a pergunta que fez e um resumo da resposta obtida.
- public Set<Picture> findByAnyTags(Set<Tag> query) retorna as fotografias que têm pelo menos um dos tags indicados.
- public void addTag(Picture pic, Tag tag)
- public void removeTag(Picture pic, Tag tag)
- public Set<Picture> findAllPictures()

#### AlbumDM:

- public long getId(Album e)
- public Album makeAlbum(Tag tag)
- Optional<Album> findByName(String albumName)
- Optional<Album> findByTag(Tag theTag)

Uma query que é suposta retornar apenas um resultado deve ser feita usando o método **getSingleResult**. O método pode lançar duas exceções: NonUniqueResultException e NoResultException (é uma vantagem). Deve portanto tratar essas exceções. Conforme os casos pode-se tratar de uma situação anómala ou não. Por exemplo se um elemento não é encontrado pode ser judicioso retornar um `Optional.empty()`.

## 7.2 Segunda fase: MemoriesJFX





# Capítulo 8

## Guiões

Em CSS os guiões têm como objetivo esclarecer um aspeto da matéria por via de experimentações realizadas pelo aluno. A maioria das dúvidas devem estar esclarecidas após a realização dos mesmos. A realização de um guião consiste em reproduzir no seu computador uma sequência de instruções. No início do guião todos os passos estão detalhados em pormenor. A medida que avança e que partes da matéria foram esclarecidas os detalhes são mais escassos, caso contrário tornaria-se muito repetitivo. Note que a descrição dos passos a realizar é sequencial pelo que, caso não obtém o resultado esperado após ter realizado o passo N, deve procurar ajuda e resolver os eventuais problemas antes de passar ao passo N+1.

Os guiões são realizados durante as aulas de laboratório; quando o guião é comprido pode acontecer que duas aulas de laboratório consecutivas estejam dedicadas ao mesmo guião. Caso não consiga completar o guião durante a(s) aula(s), **deve completa-lo em casa**. O trabalho correspondente a um guião está colocado num repórtorio git cujo nome é indicado no enunciado. Uma vez terminado deve acrescentar um commit intitulado “Guião XXX terminado” (e não esquecer empurrar para o repositório remoto).

### 8.1 Lista dos guiões a realizar durante o semestre

- Guião git 1 [23](#): introdução ao git.
- Guião git 2 [42](#): como trabalhar em equipa usando o git.
- Guião JDBC e TransactionScript [128](#): como implementar o padrão TransactionScript e trabalhar com uma base de dados.
- Guião JPA [143](#): como usar a *Java Persistence API*.
- Guião Tomcat [169](#): como realizar uma aplicação WEB simples usando *Servlets*.



# Capítulo 9

## Revisões

Exceções p.197 • SQL p.205

### 9.1 Exceções

Não é possível detectar todos os erros em tempo de compilação. Uma exceção assinala a ocorrência de uma situação especial, normalmente uma situação de erro, num programa Java. Quando uma determinada situação é detectada é “**lançada**” uma exceção (exception) que deve ser tratada por um tratador de exceções (exception handler).

Uma execção é uma instância da classe `Exception` ou das suas subclasses. Existem numerosos tipos de exceções na API. O lançamento da exceção é feito com a instrução `throw` e o seu tratamento é feito usando um bloco `try-catch`.

#### 9.1.1 Lançar uma exceção

Quando um problema não permite a continuação de um método e não existe forma de tratar o problema no contexto desse método, uma exceção deverá ser criada. Portanto, **as exceções podem ser de diversos tipos** e o tipo da exceção criada depende da situação detectada. A criação de uma exceção é feita, tal como qualquer outro objecto, com `new`. Para lançar uma exceção utiliza-se a instrução `throw` (não confundir com `throws`). Quando uma exceção é lançada a execução do método pára e a referência da exceção é enviada para o tratador de exceções.

No exemplo seguinte, a exceção do tipo `NullPointerException` é lançada quando a variável `str` tem o valor `null`.

```
if(str == null) {
 throw new NullPointerException();
 // código a seguir ao "throw new" nunca será executado.
 // A execução do método pára quando é lançada uma exceção.
}

copy
```

#### 9.1.2 Tratamento de uma exceção

O mecanismo base para tratar exceções no Java é a instrução `try-catch`, composta por um bloco `try` e um ou mais blocos `catch`. Mais precisamente, no bloco `try` encontram-se as instruções do programa que são susceptíveis de provocar um erro. O bloco `catch` é responsável por “apanhar” um certo tipo de exceção e executar uma sequência de instruções adequadas que, conforme os casos podem simplesmente avisar o utilizador ou remédiar ao problema caso seja possível. É este processamento que designamos informalmente por “tratar a exceção”. Os dois tipos de blocos são indissociáveis, i.e. não podemos usar um bloco `catch` sem usar um bloco `try`, nem podemos usar um bloco `try` sem pelo menos um bloco `catch`.

Uma execção é uma instância da classe `Exception` ou das subclasses. Existem numerosas tipos de exceções na API.

**Exemplo:** O método `nextInt` da classe `Scanner` lança uma exceção quando o valor encontrado na entrada não corresponde a um valor inteiro. O tipo desta execção é `InputMismatchException`. Existem duas formas de tratar uma exceção:

- Usando a instrução `try-catch` que permite apanhar e tratar a exceção da forma que se ache adequada.
- “Lançando” a exceção (`throws`) para os métodos na cadeia de execução até que seja encontrado um método que saiba tratar a exceção.

```
public void m(Scanner s) {
 int n;
 try {
 System.out.println("in try block.");
 n = s.nextInt();
 // mais instruções ...
 System.out.println("out try block.");
 } catch (InputMismatchException e) {
 System.out.println("Error !");
 }
 System.out.println("after catch block.");
}
```

copy

O bloco `try` é:

```
try {
 System.out.println("in try block.");
 n = s.nextInt();
 // mais instruções ...
 System.out.println("out try block.");
}
```

copy

O bloco `catch` é:

```
catch (InputMismatchException e) {
 System.out.println("Error !");
}
```

copy

Caso a leitura de um valor inteiro falhar, a saída do programa será:

```
in try block.
Error !
after catch block.
```

Se pelo contrário não houve erro na leitura:

```
in try block.
out try block.
after catch block.
```

Em caso de erro, o controlo passa imediatamente para o bloco `catch` e as instruções seguintes do bloco `try` não são executadas.

Vários tipos de erros podem ocorrer num bloco `try`. Pode-se associar vários blocos `catch` a um bloco `try` de forma a tratar de forma diferenciada os vários tipos de erro.

```
try {
 // Código que pode gerar exceções
} catch(Tipo1 id1) {
 // Código que trata as exceções de Tipo1
} catch(Tipo2 id2) {
 // Código que trata as exceções de Tipo2
} catch(Tipo3 id3) {
 // Código que trata as exceções de Tipo3
}

copy
```

Acontece com frequência que a exceção não possa ser tratada logo no método onde ocorre. No exemplo seguinte o método abre um ficheiro para ler um valor inteiro:

```
public static int readIntFromFile (String filename) {
 Scanner s = new Scanner(new FileReader(filename));
 int i = s.nextInt();
 s.close();
 return i;
}
```

**copy**

Um dos problemas que podem ocorrer é que o ficheiro indicado não existir. O problema é que este método pode ser usado em variadíssimas situações. Não sabemos de onde vem o nome do ficheiro. Pode ter sido indicado pelo utilizador (neste caso uma opção é perguntar outra vez o nome do ficheiro) mas também pode se tratar de um ficheiro da aplicação, desconhecido pelo utilizador. Não sabemos e não há maneira de resolver a falta do ficheiro **neste método**.

É provável que haja informação relevante para remediar ao problema no método que chamou `readIntFromFile` ou num método acima na cadeia de execução. Há um mecanismo que permite propagar a exceção ao longo da cadeia de execução até encontrar um método capaz de a tratar (com um bloco `try-catch`). Esta propagação é feita com a declaração **throws**:

```
public static int readIntFromFile (String filename) throws FileNotFoundException {
 Scanner s = new Scanner(new FileReader(filename));
 int i = s.nextInt();
 return i;
}
```

**copy**

Note que a versão anterior não era compilável. É obrigatório tratar (através de um bloco `try-catch`) ou lançar a exceção `FileNotFoundException`. Ainda neste exemplo, o método `nextInt` pode lançar uma exceção de tipo `InputMismatchException`. O seu tratamento é facultativo porque é subclasse de `RuntimeException`.

*RuntimeException and its subclasses are unchecked exceptions. Unchecked exceptions do not need to be declared in a method or constructor's throws clause if they can be thrown by the execution of the method or constructor and propagate outside the method or constructor boundary.*

`FileNotFoundException` não sendo uma subclasse de `RuntimeException`, deve ser tratada. O mecanismo de exceções tem como objectivo permitir o tratamento de cada tipo de erro de forma específica. Existem portanto várias classes de exceções. A exceção mais genérica é `Exception`. A título de exemplo vamos examinar agora os problemas que podem ocorrer na abertura e leitura de um ficheiro.

```
try {
 Scanner input = new Scanner(new FileReader(nomeFicheiro));
 // mais instruções aqui
 // onde podem ocorrer outros erros.
}
catch (Exception e) {
 // tratar o erro aqui
}
```

```
}
```

**copy**

Este é um mau exemplo ! Porque o `catch` trata qualquer tipo de erro que possa ocorrer no bloco `try`. Em consequência não se pode fazer um tratamento adequado do erro. Primeira conclusão: Devemos usar classes de exceções específicas ao problema encontrado. É muito raro usar a classe `Exception`. Outro exemplo :

```
try {
 Scanner input = new Scanner(new FileReader(nomeFicheiro));
 int i = input.nextInt();
}
catch (FileNotFoundException e) {
 // tratar o erro aqui
}
```

**copy**

Neste caso, o `catch` permite tratar convenientemente o erro que ocorre se o ficheiro não existir mas, outros erros são possíveis. Se o ficheiro estiver vazio, a exceção `NoSuchElementException` será lançada; se o ficheiro não contiver um número inteiro a exceção `InputMismatchException` será lançada.

Como tratar essas várias hipóteses ?

Primeira tentativa :

```
try {
 Scanner input = new Scanner(new FileReader(nomeFicheiro));
 int i = input.nextInt();
}
catch (FileNotFoundException e) {
 System.err.println("O ficheiro não existe");
}
catch (NoSuchElementException e) {
 System.err.println("O ficheiro está vazio !!!");
}
catch (InputMismatchException e) {
 System.err.println("O ficheiro não contém um inteiro !!!")
}
```

**copy**

Esta solução tem um problema. Nas instruções `try-catch` com vários blocos `catch`, apenas um destes, no máximo, é executado: o primeiro cujo tipo de exceção declarada no início do bloco `catch` corresponde ao tipo da exceção que foi lançada no bloco `try`. As exceções são tratadas na ordem dos blocos `catch`. Se o ficheiro não contiver um inteiro, vai aparecer no ecrã :

O ficheiro está vazio !!!

Porquê ? Vejamos a [documentação da classe `InputMismatchException`](#). Logo no início da página vê-se a **hierarquia das classes**. Acima da classe `InputMismatchException` encontramos a classe `NoSuchElementException` acima da qual encontramos a classe `RuntimeException` e a seguir a classe `Exception`. Nesta lista `Exception` é a classe a mais genérica e a classe `InputMismatchException` é a mais específica.

Para que o programa funcione como esperado, os blocos `catch` devem ser ordenados do mais específico até o mais genérico. A solução correcta ao problema anterior é :

```
try {
 Scanner input = new Scanner(new FileReader(nomeFicheiro));
 int i = input.nextInt();
}
catch (FileNotFoundException e) {
 System.err.println("O ficheiro não existe");
```

```

}
catch (InputMismatchException e) {
 System.err.println("O ficheiro não contém um inteiro !!!");
}
catch (NoSuchElementException e) {
 System.err.println("O ficheiro está vazio !!!");
}
}

copy

```

Note que é igualmente possível tratar várias exceções no mesmo bloco `catch`:

```

try {
 Scanner input = new Scanner(new FileReader(nomeFicheiro));
 int i = input.nextInt();
}
catch (FileNotFoundException | InputMismatchException e) {
 System.err.println("Há um problema");
}

```

[copy](#)

### Resumindo:

- Uma exceção representa uma anomalia na execução do programa, é uma instância da classe `Exception` ou de uma das suas subclasses.
- A instrução `throw` é usada para lançar uma exceção.
- Quando ocorre uma anomalia ou um erro, este deve ser tratado, isto é, no mínimo, deve avisar-se o cliente (se não houver maneira de corrigir o erro automaticamente). Para esse efeito deve usar-se a instrução `try-catch` para capturar e tratar uma exceção. Esta instrução é composta por um bloco `try` e um ou mais blocos `catch`.
  - No bloco `try` ficam as instruções que podem causar o erro.
  - É usado um bloco `catch` para cada tipo de exceção que se quer tratar.
  - A ordem dos blocos `catch` é relevante.
- Como pode não ser conveniente tratar um erro acrescentando código junto do sítio onde o erro pode ocorrer, é possível delegar o seu tratamento usando a palavra `throws` na assinatura do método para anunciar as exceções que podem resultar da sua execução.
- Há todavia um conjunto de exceções que não são verificadas e, portanto, não requerem `throws` na assinatura dos métodos onde podem ser lançadas. É o caso, por exemplo, da `ArithmeticException`

### Métodos da classe `Exception`

A classe `Exception` oferece os seguintes construtores e principais métodos:

- `Exception(String message)`. Para além do construtor sem argumento, existe este construtor que associa uma mensagem à instância de `Exception`.
- `String getMessage()`: retorna a mensagem associada à instância.
- `void printStackTrace()` imprime no ecrã (no canal `System.err`) o caminho de execução até o erro, indicando o número das linhas das chamadas às diversos métodos. Exemplo:

```

java.lang.NullPointerException
 at MyClass.mash(MyClass.java:9)
 at MyClass.crunch(MyClass.java:6)
 at MyClass.main(MyClass.java:3)

```

### 9.1.3 Criação de novas exceções

No contexto de uma aplicação pode surrir a necessidade de criar exceções mais específicas do que aquelas definidas na API. A definição de uma exceção é feita criando uma subclasse da classe `Exception`. Na maioria dos casos não é necessário redefinir os métodos da classe `Exception`. Pode-se definir o constructor caso seja necessário haver um contrutor com argumento(s) :

```
public class SuperSpecialException extends Exception {
 public SuperSpecialException (String message) {
 super(message);
 }
}
```

[copy](#)

### 9.1.4 Asserções

Pode testar o seu programa usando asserções. Por exemplo :

```
assert hours < 24;
```

[copy](#)

se a condição indicada for falsa, uma exceção de tipo `AssertionError` é lançada. Existe outra forma onde se especifica a mensagem de erro que deve ser associada à exceção :

```
assert hours < 24 : "A hora de deve ser inferior a 24.";
```

[copy](#)

A mensagem é passada ao constructor da exceção e pode ser obtida com o método `getMessage`. Uma possível utilização das asserções é a verificação dos contratos.

O calculo da expressão associada a uma asserção pode ser complexo e penalizar a rapidez de execução da apliação. Por essa razão, é possivel quando se corre um programa, especificar se as asserções devem ser verificadas ou não. Por omissão as asserções não são verificadas !

- Para efetuar a verificação das asserções, deve-se usar a opção `--enableassertions` (ou `-ea`) e,
- para as omitir a opção `--disableassertions` (ou `-da`).
- Pode-se ainda especificar as classes onde as asserções devem ser testadas (ou omitidas) desta forma:

```
> java -ea:fcul.pco.TuaClasse -da:fcul.pco.MinhaClasse Prog
```

É igualmente possível especificar um pacote, neste caso as asserções são verificadas/ignoradas em todas as classes do pacote.

Pode indicar os argumentos da linha de comando no menu “Run configuration” do eclipse.

### 9.1.5 Finally !

“finally” é uma clausúla associada a uma instrução `try`. O código associado a este bloco é sempre executado, que tenha ocorrido uma exceção ou não. É gralmente usado para libertar um recurso que foi usado no bloco `try`. Por exemplo:

```
void m() {
 Scanner s;
 try {
 s = new Scanner(new FileReader("Fich.dat"));
 // ... more computations
 } catch (FileNotFoundException e) {
 // do something !
 } finally {
 s.close();
 }
}
```

```

 }
}

copy

```

### 9.1.6 try-with-resource

Pode-se associar um ou mais blocos catch á instrução try-with-resources :

```

public static void main(String[] args) {
 try (BufferedWriter bw = new BufferedWriter(new FileWriter("tabuada7"))) {
 for (int i = 1; i < 10; i++) {
 bw.write(i + " x 7 = " + i * 7);
 bw.newLine();
 }
 } catch (IOException ex) {
 System.err.println("Occoreu um problema na escrita do ficheiro tabuada7.");
 }
}

```

```
copy
```

### 9.1.7 Exercício

Para este exercício vai-se considerar a classe cliente RunOperacoes (fornecida ) que recebe como input um ficheiro de texto cuja primeira linha contém o número n de inteiros a ler, e a segunda linha contém n inteiros separados por um espaço. O ficheiro `inteiros.txt` é um dos ficheiros que deve usar para testar esta classe.

```

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.util.*;

public class RunOperacoes {

 /**
 * @param args
 * @throws FileNotFoundException
 */
 public static void main(String[] args) throws FileNotFoundException {

 Scanner sc = new Scanner(new BufferedReader(new FileReader(args[0])));

 int tamanho = lerTamanhoArray(sc);
 int[] inteiros = lerArrayInteiros(sc, tamanho);

 sc.close();
 }

 public static int lerTamanhoArray(Scanner sc) throws NumberFormatException {
 int n = Integer.parseInt(sc.nextLine());
 return n;
 }

 /**
 * @requires tamanho>0
 */

```

```

public static int[] lerArrayInteiros(Scanner sc, int tamanho) {
 String linha = sc.nextLine();
 String [] nrs = linha.split(" ");
 int[] inteiros = new int[tamanho];

 for (int i = 0; i < tamanho; i++){
 inteiros[i] = Integer.parseInt(nrs[i]);
 }

 return inteiros;
}
}

```

⇒ RunOperacoes.java

Neste exercício **não (!) deve assumir que o input está sempre bem construído**. Deve testar o correcto tratamento das exceções **usando diferentes ficheiros de input** de forma a que as exceções sejam forçosamente lançadas.

1. Complete o método `lerTamanhoArray` de forma a que lance uma exceção do tipo `Exception`, **mas não a trate (!)**, quando a primeira linha do ficheiro não corresponde a um inteiro positivo.
2. Complete o método `lerArrayInteiros` de forma a que: 1) lance uma exceção do tipo `Exception`, **mas não a trate (!)**, quando existe um inteiro não positivo na segunda linha do ficheiro; 2) lance uma exceção do tipo `Exception`, **mas não a trate (!)**, quando o tamanho não é igual ao número inteiros na segunda linha. Deve associar a cada uma das exceções uma mensagem adequada.
3. Complete o método `main` de forma que este trate as possíveis exceções lançadas pelos métodos `lerTamanhoArray` e `lerArrayInteiros`. Também deve tratar a exceção do tipo `NumberFormatException` que pode ser lançada pelo método `parseInt` quando a string lida não corresponde a um inteiro.
4. Crie a classe servidora `Operacoes` com três métodos:
  - `int totalMaiores(int[] inteiros, int n)` - retorna o número de todos os inteiros maiores que `n` no array inteiros.
  - `int soma(int[] inteiros, int n)` - retorna a soma de todos os inteiros maiores que `n` no array inteiros.
  - `double media(int[] inteiros, int n)` - retorna a média de todos os inteiros maiores que `n` no array de inteiros utilizando os métodos `totalMaiores` e `soma` anteriores. Deve apanhar a exceção `ArithmaticException` e tratá-la retornando 0 se esta for lançada (por exemplo, quando há uma divisão por zero).
5. Complete a classe cliente `RunOperacoes` de forma a que crie um objecto `Operacoes` e chame o método `media` tendo como argumentos o array de inteiros lido do ficheiro de input e o número 7, e imprima o seu resultado no ecrã.
6. Altere a classe `RunOperacoes` de forma a tratar a exceção `FileNotFoundException` caso o ficheiro de input não exista, associando uma mensagem adequada.

Neste exercício **não (!) deve assumir que o input está sempre bem construído**. Deve testar o correcto tratamento das exceções **usando diferentes ficheiros de input** de forma a que as exceções sejam forçosamente lançadas.

1. Complete o método `lerTamanhoArray` de forma a que lance uma exceção do tipo `Exception`, **mas não a trate (!)**, quando a primeira linha do ficheiro não corresponde a um inteiro positivo.
2. Complete o método `lerArrayInteiros` de forma a que: 1) lance uma exceção do tipo `Exception`, **mas não a trate (!)**, quando existe um inteiro não positivo na segunda linha do ficheiro; 2) lance uma exceção do tipo `Exception`, **mas não a trate (!)**, quando o tamanho não é igual ao número inteiros na segunda linha. Deve associar a cada uma das exceções uma mensagem adequada.

3. Complete o método `main` de forma que este trate as possíveis exceções lançadas pelos métodos `lerTamanhoArray` e `lerArrayInteiros`. Também deve tratar a exceção do tipo `NumberFormatException` que pode ser lançada pelo método `parseInt` quando a string lida não corresponde a um inteiro.
4. Crie a classe servidora `Operacoes` com três métodos:
  - `int totalMaiores(int[] inteiros, int n)` - retorna o número de todos os inteiros maiores que `n` no array inteiros.
  - `int soma(int[] inteiros, int n)` - retorna a soma de todos os inteiros maiores que `n` no array inteiros.
  - `double media(int[] inteiros, int n)` - retorna a média de todos os inteiros maiores que `n` no array de inteiros utilizando os métodos `totalMaiores` e `soma` anteriores. Deve apanhar a exceção `ArithmetricException` e tratá-la retornando 0 se esta for lançada (por exemplo, quando há uma divisão por zero).
5. Complete a classe cliente `RunOperacoes` de forma a que crie um objecto `Operacoes` e chame o método `media` tendo como argumentos o array de inteiros lido do ficheiro de input e o número 7, e imprima o seu resultado no ecrã.
6. Altere a classe `RunOperacoes` de forma a tratar a exceção `FileNotFoundException` caso o ficheiro de input não exista, associando uma mensagem adequada.

## 9.2 SQL

Structured Query Language (SQL) é uma linguagem que permite realizar operações numa base de dados, como criar entradas, ler conteúdo, atualizar conteúdo e remover entradas.

O SQL é suportado por quase todos as bases de dados e permite que escrever o código que interage com a base de dados independentemente da base de dados subjacente.

Este capítulo fornece uma visão geral do SQL, que é um pré-requisito para entender os conceitos JDBC. Depois de passar por este capítulo, você será capaz de criar, criar, ler, atualizar e excluir (geralmente chamadas de operações CRUD) de dados de um banco de dados.

- **Criar uma base de dados:**

```
CREATE DATABASE DATABASE_NAME;
```

- **Remover uma base de dados:**

```
DROP DATABASE DATABASE_NAME;
```

- **Criar uma tabela:**

```
CREATE TABLE table_name
(
 column_name column_data_type,
 column_name column_data_type,
 column_name column_data_type
 ...
);
```

**Exemplo:**

```
CREATE TABLE Employees
(
 id INT NOT NULL,
 age INT NOT NULL,
 first VARCHAR(255),
 last VARCHAR(255),
 PRIMARY KEY (id)
);
```

- **Remover uma tabela:**

```
DROP TABLE table_name;
```

**Exemplo:**

```
DROP TABLE Employees;
```

- **Inserir uma linha numa tabela:**

```
INSERT INTO table_name VALUES (column1, column2, ...);
```

**Exemplo:**

```
INSERT INTO Employees VALUES (100, 18, 'Maria', 'Batata');
```

- **Obter valores de uma tabela:**

```
SELECT column_name, column_name, ...
FROM table_name
WHERE conditions;
```

Por exemplo, a seguinte instrução SQL seleciona a idade, a primeira e a última colunas da tabela Employees, onde a coluna id é 100

```
SELECT first, last, age
FROM Employees
WHERE id = 100;
```

A seguinte instrução SQL seleciona a idade, a primeira e a última coluna da tabela Employees onde a primeira coluna contém Maria

```
SELECT first, last, age
FROM Employees
WHERE first LIKE '%Maria%';
```

A cláusula WHERE pode usar os operadores de comparação como `=, !=, <, >, <= >=`, bem como os operadores BETWEEN e LIKE. O operador LIKE permite fazer uma procura usando um padrão. Exemplos:

- WHERE CustomerName LIKE 'a%'  
Finds any values that start with "a"
- WHERE CustomerName LIKE '%a'  
Finds any values that end with "a"
- WHERE CustomerName LIKE '%or%'  
Finds any values that have "or"in any position
- WHERE CustomerName LIKE '\_r%'  
Finds any values that have "r"in the second position
- WHERE CustomerName LIKE 'a\_%'  
Finds any values that start with "a"and are at least 2 characters in length
- WHERE CustomerName LIKE 'a\_\_%'  
Finds any values that start with "a"and are at least 3 characters in length
- WHERE ContactName LIKE 'a%o'  
Finds any values that start with "a"and ends with "o"

- **Modificar dados:**

```
UPDATE table_name
SET column_name = value, column_name = value, ...
WHERE conditions;
```

Por exemplo, a seguinte instrução SQL UPDATE altera a coluna de idade do funcionário cujo id é 100 :

```
UPDATE Employees SET age=20 WHERE id=100;
```

- **Remover linhas da tabela:**

```
DELETE FROM table_name WHERE conditions;
```

**Exemplo :**

```
DELETE FROM Employees WHERE id=100;
```