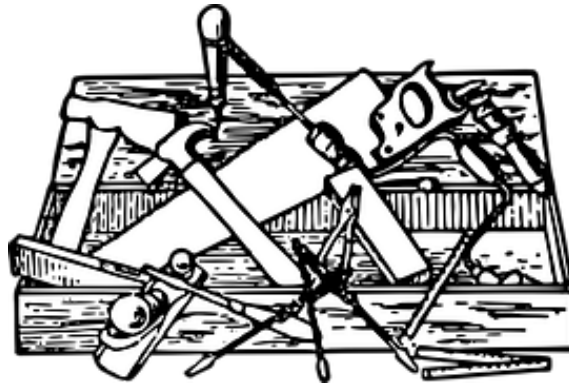


ENGENHARIA DE SOFTWARE

Guia de Laboratório

Pedro Reis dos Santos

7 de Fevereiro de 2016



Conteúdo

1	Introdução	2
2	Gestão de configurações: maven	5
3	Gestão de versões: git	10
4	Persistência e transações: fénix framework	19
5	Software web hosting: github	26
6	Gestão do projetos ágil: Scrum	33
7	Aplicação por camadas: serviços e apresentação	37
8	Testes de software: junit e jmockit	41

1 Introdução

O projeto da disciplina funciona em equipas de 6 alunos, todos inscritos no mesmo turno de laboratório. As inscrições estão abertas até 19 de fevereiro de 2016.

Cada aluno deve criar uma conta no *github*, se ainda não possuir uma, e submeter o seu número mecanográfico e o seu nome no *github* ao projeto '*github*' já aberto no fénix. O prazo termina a 19 de fevereiro de 2016.

1.1 Projeto

O projeto participa com 30% da nota final e tem uma nota mínima de 9,0 valores (em 20 valores). Os alunos que ainda não obtiveram aprovação à disciplina são obrigados a repetir toda a avaliação, incluindo o projeto. O projeto deve ser entendido como um processo de aprendizagem, em que o aluno se familiariza com os conceitos e as técnicas que serão avaliadas nos testes teóricos.

Projeto decomposto em três entregas, de cerca de quatro semanas cada: realização do domínio; camada de serviços e seus testes; camada de apresentação e testes (*mock-ups*, integração e sistema). As três entregas realizam-se no **github.com**, respetivamente nos dias 14 março [30%], 15 abril [35%] e 13 maio [35%] até às 20h00. A discussão final do projeto realiza-se entre 16 e 25 de maio de 2016.

Cada novo enunciado acrescenta funcionalidades e pode alterar funcionalidades anteriores. O código apresentado deve ser à prova de programador e de utilizador, ou seja, impede erros não deliberados ou casuais (distração ou desconhecimento). Não apresentar código duplicado! Refatorizar mesmo que envolva vários membros da equipa. O novo código e as alterações/correções devem ficar disponíveis para todos os membros da equipa o mais cedo possível: *continuous integration*.

1.2 Equipa

O projeto é em equipa, não é em grupo. Cada elemento da equipa tem um trabalho específico individual a realizar, determinado e avaliado semanalmente. Equipa deve ser homogénea, sob risco de ficar nivelada pelo elemento mais fraco. Equipa divide as tarefas pelos elementos, sob a supervisão (e avaliação) semanal do docente de laboratório. A equipa deve garantir que em cada entrega o projeto funciona como um todo. Trabalho não realizado nas datas das entregas do projeto (P1, P2, P3) vale 0 (zero) e pode ter de vir a ser realizado posteriormente sem valorização.

Todos os membros da equipa tem de perceber o código escrito pela restante equipa, têm a obrigação de verificar o código dos restantes membros, sugerir melhorias e, eventualmente alterar o código de outros membros da equipa. Trabalhar em nome de outros implica que o trabalho é creditado aos outros.

1.3 Software

O software necessário para a realização do projeto da disciplina pode ser instalado em qualquer sistema operativo, Windows, Linux e Mac OS X. No entanto, na documentação e nos laboratórios assume-se Linux. O comando `sudo apt-get install maven git mysql-server` deve instalar o software necessário, embora talvez não inclua as versões mais recentes. Assume-se que já existe uma instalação recente de Java `jdk` de disciplinas anteriores.

Windows Em Windows, lembra-se que as linhas dos ficheiros terminam em `cr-lf` (`\r\n`), que os separadores de diretorias são `'\'` e não `'/'` e que os caminhos são separados por `';` e não `':'` (`\;` em algumas emulações de **unix**). É responsabilidade do aluno fazer as adaptações necessárias e garantir que o projeto funciona corretamente em **linux**.

O *git for windows* oferece uma plataforma de trabalho semelhante ao **unix** com suporte para **git**. Na instalação deve garantir que os ficheiros são guardados em formato **unix**. O *maven* é escrito em Java, logo independente da arquitetura.

java Aconselha-se a instalação do **jdk-1.8** da **oracle**, embora a versão **1.7** seja utilizável. As variáveis de ambiente `JAVA_HOME`, `PATH` e `CLASSPATH` devem ser atualizadas para as novas localizações do software.

maven A versão da ferramenta deve ser pelo menos **3.2.5** para não haver problemas no processamento de alguns `pom.xml` mais complexos. Atualizar as variáveis de ambiente `M2_HOME` e `PATH` em função da instalação.

git A ferramenta **git** não tem restrições, embora se aconselhe a versão 2 ou mais recente.

github Os projetos a realizar na disciplina deverão residir no *github*. Como as conta de utilizador apenas permitem projetos públicos, a disciplina irá fornecer projetos privados. Para ter acesso ao projeto privado do seu grupo, o aluno deve indicar a associação entre o seu número de aluno e nome da sua conta de *github*. Para tal, deverá foi criado na página da disciplina um projeto individual chamado *github*. Cada aluno deve submeter um ficheiro, contendo apenas o número do aluno (5 dígitos) e o nome *github*, separados por um espaço. Se ainda não tem conta no *github*, deve criar uma.

mysql Não mudar o nome do PC depois de instalar o servidor de `mysql`! O servidor de *mysql* deve ser instalado com a palavra passe *rootroot* para o administrador *root*, pois alguns programas de exemplo assumem esse valor. Sugere-se a instalação com *auto-start* para evitar esquecimentos. Atualizar as variáveis de ambiente `MYSQL_HOME` e `PATH` para a localização da instalação.

Para a realização do projeto deve ser criado um utilizador `mydrive` com palavra passe `mydriv3` e uma base de dados `drivedb`, que serão utilizados pela aplicação e na avaliação de uma forma automática.

```
$ mysql -u root -p
mysql> GRANT ALL PRIVILEGES ON *.* TO 'myDrive'@'localhost'
IDENTIFIED BY 'myDriv3' WITH GRANT OPTION;
mysql> CREATE DATABASE drivedb;
```

Todas as aplicações que utilizem a **fénix-framework** (ver seção4) necessitam que a respetiva base de dados seja criada antes de iniciar a aplicação. Verificar as necessidades específicas de cada aplicação em

```
src/main/resources/fenix-framework-jvstm-objb.properties
```

1.4 eclipse

O aluno deve saber utilizar as ferramentas acima a partir da linha de comandos. No entanto, a utilização do ambiente de desenvolvimento integrado *Eclipse IDE for Java EE (Neon)* pode ser vantajosa no desenvolvimento do projeto. Esta ferramenta não é necessária ao desenvolvimento do projeto, mas pode agilizar o seu desenvolvimento através da integração do **git** e do **maven**. O utilizador deve atualizar a variável de ambiente `ECLIPSE_HOME` e configurar o Eclipse para utilizar a versão do **jdk** que tem instalada. Sugere-se, igualmente, a instalação dos conectores `m2e:jaxws-maven-plugin`, `maven-dependency-plugin`, `e-git` e `org.codehaus.mojo:jaxb2-maven-plugin`, através dos menus: Help → Install New Software → Work with

Integração com maven Antes de integrar um projeto **maven** existente no eclipse, é necessário gerar os ficheiros auxiliares do **eclipse** com o comando `mvn eclipse:eclipse`. Depois o projeto pode ser importado com os menus:

```
File → Import → Maven → Existing Maven Project
```

Para criar um projeto **maven** a partir do **eclipse**, introduzir os valores necessários (ver seção 2) no menu `package explorer[view] → new → other → new maven project → (use default location) → [add archetype]`

Integração com git Para a utilização da ferramenta **git** no **eclipse** é necessário definir no **eclipse**, além do já efetuado para a linha de comandos (ver seção 3.3), o `user.name` e `user.mail`.

Os comandos **git** estão disponíveis na opção Team do projeto em causa, na janela do Package Explorer, depois de criar ou importar o projeto. Para criar *shortcuts* na barra do eclipse para as operações mais comuns de **git** deve ativar a opção **git** em Window → Customise perspective. A informação do **git** pode ser permanentemente visível numa janela, ativada no menu Window → Show View → Git.

2 Gestão de configurações: maven

A gestão de configurações pretende automatizar a produção de *software* a partir dos ficheiros que têm de ser manipulados pelo programador. O objectivo é gerar automaticamente todos os produtos cuja produção possa ser automatizada. Assim, evita-se a realização de tarefas repetitivas, constituídas por diversos passos, e que são desnecessariamente sujeitas a erros quando realizadas manualmente.

História A ferramenta **make** baseia-se na comparação das datas de modificação do ficheiro objetivo com as datas dos seus dependentes. Se a data de, pelo menos, um dos dependentes é mais atual que o objetivo, uma sequência de comandos é executada com o intuito de atualizar o objetivo. A escrita do ficheiro *Makefile* que serve de especificação ao **make**, torna-se trabalhosa em projetos de maior dimensão. Assim, surgiram ferramentas que ajudam na construção de *Makefiles*: *imake*, *makedepend*, *configure*, ...

No ambiente **Java** apareceram mais adaptadas como o **ant**, embora a ferramenta **make** também possa ser utilizada. Contudo, o **ant** também é baseado em comandos e, tal como o **make**, requer que todas as dependências estejam disponíveis.

Maven A ferramenta **maven** (<https://maven.apache.org/>) baseia-se numa especificação declarativa (*pom.xml*) e responsabiliza-se por obter, a partir de repositórios predefinidos, as dependências e as dependências das dependências. Para gerir projetos cada vez mais complexos, o **maven** gere a estrutura do projeto (especificada por um arquétipo) através de um ciclo de produção (especificado por um *lifecycle*). Adicionalmente, um conjunto de *plugins*, permitem controlar cada fase do ciclo de produção dependendo do projeto. Sempre que um novo arquétipo, dependência ou *plugin* são necessários, o **maven** requer ligação à *net*, guardando a informação recolhida num repositório local (*\$HOME/.m2*).

2.1 Estrutura do projeto: arquétipo

Um arquétipo é uma estrutura de diretorias com alguns ficheiros de exemplo e de configuração (um ou mais *pom.xml*). Na realidade, toda esta informação é compactada num ficheiro *.jar* e é distribuída através de um servidor *web*, onde o **maven** vai buscar e expandir o arquétipo.

O comando `mvn archetype:generate` permite obter interativamente um dos 1470 arquétipos disponíveis em `org.apache.maven.archetypes` (2015). Durante a execução do comando, o utilizador especifica os parâmetros necessários. Por omissão, o arquétipo utilizado é um simples *hello-world* `maven-archetype-quickstart`. Como os arquétipos vão sofrendo evoluções ao longo dos anos, é necessário indicar a versão, em geral a última. O **groupId** é um *java fully qualified name*, ou seja, o nome da equipa com todo o contexto, por exemplo `pt.tecnico.leic.es`. Todos os pacotes desenvolvidos pela equipa devem ser sub-pacotes deste. O **artifactId** é o nome do ficheiro

.jar a produzir, sem incluir a versão. A versão é um número, eventualmente composto (2.3.1), um nome ou etiqueta, frequentemente com o sufixo **SNAPSHOT** quando se tratam de versões diárias. Finalmente, o pacote (*package*) ou coincide com o **groupId** ou é um sub-pacote deste, por exemplo `pt.tecnico.leic.es.ist99999`.

A versão não interativa do comando gera o arquétipo, sem perguntas, desde que sejam fornecidos todos os parâmetros. O comando é introduzido todo na mesma linha ou, em UNIX, pode ser dividido em várias linhas com o carácter de continuação no fim de cada uma, exceto a última:

```
mvn archetype:generate \
  -DgroupId=pt.tecnico.agenda \
  -DartifactId=agenda \
  -DarchetypeArtifactId=maven-archetype-quickstart \
  -DinteractiveMode=false
```

Qualquer servidor de **http** pode ser utilizado para disponibilizar arquétipos. Neste caso, a informação do servidor necessita ser fornecida. No caso da **fénix framework** (ver seção 4) a aplicação simples pode ser construída com:

```
mvn archetype:generate \
  -DarchetypeGroupId=pt.ist
  -DarchetypeArtifactId=fenix-framework-application-archetype-clean \
  -DarchetypeVersion=2.0 \
  -DarchetypeRepository=https://fenix-ashes.ist.utl.pt/maven-public \
  -DgroupId=example \
  -DartifactId=helloworld \
  -Dversion=1.0-SNAPSHOT \
  -DinteractiveMode=false
```

2.2 Ciclo de produção: lifecycle

O **maven** baseia-se na execução de ciclos de produção ou *lifecycles*. Um ciclo de produção descreve num ou mais passos a forma de obter determinado resultado. Existem três ciclos de produção pré-instalados no **maven**: *default*, *clean* e *site*. Outros ciclos podem ser definidos e integrados. Cada ciclo é constituído por uma ou mais fases executadas sequencialmente. O utilizador indica a fase até onde pretende desenvolver o projeto. Se uma fase não conseguir ser concluída, as seguintes não serão executadas. Associado a cada fase podem existir um ou mais *plugins* que especificam a forma como a fase é executada em cada projeto.

O ciclo *default* é o ciclo de produção por omissão e tem por missão a construção e instalação da aplicação. O *default* é constituído por 14 fases das quais salientamos:

compile para compilar o código fonte,

test para testar a funcionalidade do código compilado,

package para construir o pacote a instalar (por exemplo, **.jar**),

integration-test integra o pacote num ambiente para executar os testes de integração,

install copia o pacote para o repositório local do **maven**, para poder ser utilizado por outras aplicações locais,

deploy copia o pacote para um repositório remoto para partilha com outros utilizadores.

O ciclo *clean* é constituído por três fases (*pre-clean*, *clean* e *post-clean*) apaga a informação gerada de forma automática. As fases inicial e final permitem suspender e reativar serviços como bases de dados ou servidores que possam utilizar a informação gerada.

O ciclo *site* é constituído por duas fases (*site* e *site-deploy*), constrói e instala a documentação do projeto.

Outros ciclos podem ser integrados no maven. Um ciclo bastante útil é o *exec-maven-plugin* que permite executar programas a partir do **maven** sem ter de definir o *classpath* para todos os pacotes de que o projeto depende. Este ciclo permite a execução da aplicação na máquina virtual **java** onde corre o **maven** (**exec:java**) ou como um processo exterior (**exec:exec**). As opções, disponíveis em (<http://www.mojohaus.org/exec-maven-plugin/>), incluem as propriedades `exec.mainClass` e `exec.args` para definir a classe principal e os seus argumentos. Estas propriedades podem ser indicadas na linha de comandos, por exemplo:

```
mvn exec:java -Dexec.mainClass=com.example.Main -Dexec.args="a b c"
```

Alternativamente, as propriedades podem ser definidas no `pom.xml` do projeto em configuration do plugin (`exec-maven-plugin`), por exemplo a `mainClass`:

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>exec-maven-plugin</artifactId>
  <version>1.3.2</version>
  <executions> <execution> <goals>
    <goal>java</goal>
  </goals> </execution> </executions>
  <configuration>
    <mainClass>example.Main</mainClass>
  </configuration>
</plugin>
```

ficando o comando reduzido a:

```
mvn exec:java -Dexec.args="a b c"
```

Como é possível indicar os vários ciclos, e as respetivas fases, no comando **maven** pode-se num só comando limpar os ficheiros gerados automaticamente, produzir o pacote do projeto e executar a aplicação resultante:

```
mvn clean package exec:java -Dexec.args="a b c"
```

2.3 Documentação

A documentação técnica do projeto pode ser gerada com auxílio do **maven**. É recolhida diversa informação do projeto, como dependências e membros da equipa e são geradas páginas **.html** com a documentação.

Para identificar cada membro da equipa é necessário indicar um identificador (**id**), um nome, email e organização. Nesta disciplina pretende-se que o identificador seja o número mecanográfico, o nome seja o *username* no **github** e a organização seja o grupo:

```

<developers>
  <developer>
    <id>ist99999</id>
    <name>github username</name>
    <email>ist99999@tecnico.ulisboa.pt</email>
    <organization>es16 g99al</organization>
    <roles>
      <role>Architect</role>
      <role>Developer</role>
      <role>Tester</role>
      <role>Integrator</role>
    </roles>
    <properties>
      <picUrl>https://fenix.tecnico.ulisboa.pt/user/photo/ist199999</picUrl>
    </properties>
  </developer>
</developers>

```

Para gerar a documentação total do projeto deve ser executado o ciclo *site* (`mvn site`), ficando a informação da equipa em `target/site/team-list.html`.

2.4 Dependências do projeto

Só projetos muito simples não dependem de código de outros. Nos casos mais simples as dependências restringem-se às bibliotecas do **java**. Contudo, à medida que o projeto se torna mais complexo, é natural que se utilizem bibliotecas externas. O **maven** permite automatizar o processo de obter bibliotecas remotas e de as incluir nos ciclos de construção e execução da aplicação em desenvolvimento. No entanto, é necessário indicar ao **maven** a localização das referidas bibliotecas.

O **maven** possui um repositório de bibliotecas (<http://search.maven.org> ou <http://mvnrepository.com/>) onde podem ser obtidas as referências para as bibliotecas de maior uso em **java**. Indicando o nome do pacote pretendido na caixa de pesquisa, por exemplo `joda time`, e escolhendo a versão pretendida, em geral a mais recente, obtém-se a descrição da biblioteca. Selecionando a informação da dependência, esta deve ser incluída na secção `dependencies` do `pom.xml`:

```

<dependency>
  <groupId>joda-time</groupId>
  <artifactId>joda-time</artifactId>
  <version>2.9.1</version>
</dependency>

```

Quando o **maven** executar, a biblioteca será descarregada (é necessário dispor de uma ligação de rede) e incluída no ciclo de construção da aplicação.

2.5 Testes do projeto

Os testes permitem garantir, e não apenas assumir, que o código efetivamente faz o que se pretende, mesmo depois de efetuar alterações aparentemente inócuas. Assim, é

extremamente importante que os testes sejam desenvolvidos em conjunto com as funcionalidades que testam, e que estes testes sejam corridos sempre o projeto é compilado, como forma de garantir a respetiva funcionalidade ainda se encontra operacional. Por esta razão, o **maven** executa sempre os testes como fase do processo de construção da ferramenta. Sempre que os testes falham, as fases seguintes do processo de construção não são executadas. Para se conseguir construir o pacote, por exemplo, sem correr os testes (a evitar) é necessário indicá-lo explicitamente:

```
mvn package -DskipTests=true
```

Para compilar e correr apenas os testes, sem gerar o pacote:

```
mvn test
```

Os testes encontram-se organizados em classes (ver seção 8.1) que podem ser selecionadas individualmente através de uma lista de nomes de classes de teste (separadas por vírgulas):

```
mvn -Dtest=TestSquare,TestCircle test
```

Cada classe de teste inclui, em geral, diversos testes da classe que está a testar. É possível selecionar quais os testes dessa classe é que se pretende que sejam executados (separadas por sinais de '+' após o nome da classe e o carácter '#'):

```
mvn -Dtest=TestCircle#testOne+testTwo test
```

2.5.1 Testes de cobertura

O objetivo da análise de cobertura é verificar quais são as partes do código que estão efetivamente a ser testadas pelos testes executados. O **cobertura** é uma ferramenta de análise de cobertura para **java** que é executado com `mvn cobertura:cobertura` produzindo os resultados da análise em `target/site/cobertura/index.html`.

2.6 Plugins

O **maven** apenas gere as fases do processo de desenvolvimento da aplicação, são os *plugins* cujos objetivos associados a essas fases fazem efetivamente o trabalho. Os *plugins* permitem a reutilização da lógica de construção das aplicações em múltiplos projetos. Os *plugins* são parametrizados de acordo com o projeto e ativados a partir de um, ou mais, objetivos ou *mojos*. O *mojo* especifica os metadados do objetivo: o seu nome, a fase do ciclo de produção em que se insere e os parâmetros necessários.

Os *plugins* podem ser descarregados automaticamente pelo **maven** ou manual com **plugin:download**, indicando `groupId`, `artifactId` e `version`.

O **maven** possui um repositório de *plugins* vulgarmente utilizados pesquisável em <http://search.maven.org>. Alternativamente, podem ser definidos repositórios alternativos:

```
<pluginRepositories>
  <pluginRepository>
    <id>fenix-ashes-maven-repository</id>
    <url>https://fenix-ashes.ist.utl.pt/maven-public</url>
  </pluginRepository>
</pluginRepositories>
```

3 Gestão de versões: git

Perder o trabalho já realizado é um problema que tem preocupado os programadores. Em especial quando a versão de ontem, ou de há uma semana, já tinha isto ou aquilo a funcionar. Já para não falar quando, por uma razão ou por outro, o trabalho é apagado e a última cópia que existe é tão antiga que nem vale a pena usar, mais vale começar do início.

História Uma primeira abordagem consiste em fazer regularmente copias em bloco do trabalho, etiquetando-as com a data atual. Se o processo não for simples acaba por ser sistematicamente ignorado e esquecido. O sistema **vax-vms** criava uma nova versão de cada vez que o ficheiro era escrito (por exemplo, `Main.java;1`), sendo usada a última versão sempre que esta era omitida. Um ficheiro **zip** gerado de uma forma automática, todos os dias, de hora a hora, em cada compilação com êxito. Nestes casos, o principal problema é a proliferação descontrolada de versões. O problema era suficientemente grave para o **vax-vms** ter criado o comando **purge** que eliminava todas as versões, excepto a última, retirando a vantagem das versões com a utilização frequente deste comando.

As ferramentas **sccs** (Source Code Control System, 1972) e **rcs** (Revision Control System, 1982) permitem guardar as diferentes alterações de um ficheiro num só ficheiro de alterações com um comando simples. O **sccs** guarda a primeira versão e as diferenças para a segunda (*delta*), da segunda para a terceira, *etc.* enquanto o **rcs** guarda a última e as alterações (*reverse delta*) para a penúltima *etc.* privilegiando a recuperação das versões mais atuais. O **cvs** (Concurrent Versions System, 1990) estende o conceito do **rcs** a uma hierarquia de ficheiros, bastando um só comando para guardar todas as alterações de uma hiararquia de ficheiros modificados num repositório centralizado. Correntemente, este repositório pode ser acedido remotamente e ser partilhado por diversos utilizadores. O **subversion** (Apache Subversion, 2000) é uma evolução do **cvs**, mas tal como o **cvs** quando o repositório é remoto, requer uma ligação ao servidor para guardar as alterações.

Sistemas mais recentes de controlo de versões foram desenhados para ser distribuídos, como o **darcs** (2003), **mercury**, **bazaar**, **git** (todos de 2005), permitem que cada repositório possa ser também um servidor. Assim, as alterações são guardadas localmente e, de tempos a tempos, sincronizadas com o servidor. O conceito evolui para que o servidor possa estar localizado na nuvem, aumentando a sua disponibilidade, surgindo serviços especializados como o **sourceforge** (1999), **launchpad** (2004), **github** (2008), **bitbucket** (2008), **gitlab** (2011), **stash** (2012), **bamboo** (2007), *etc.*. Em termos de popularidade o **github** (ver seção 5) surge em primeiro lugar com 10 milhões de utilizadores e 26 milhões de projetos, seguido do **SourceForge** (3.4Mu, 324Kp), **Bitbucket** (2.5Mu, 93Kp), **Launchpad** (2.1Mu, 32Kp) ou **GitLab** (20Ku, 100Kp), entre outros. A maioria destes serviços pode ser acedido por diversos sistemas de controlo de versões, desde o **cvs** ao **git**. Por exemplo, o **SourceForge** permite utilizar **cvs**, **git**, **svn** ou **mercurial**, enquanto o **github** e o **gitlab** apenas permitem **git** (<https://git-scm.com/>).

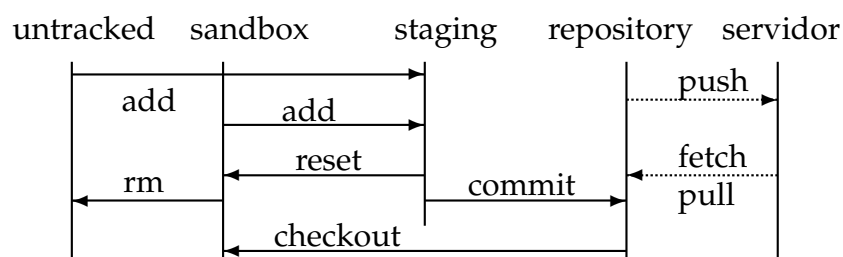
3.1 Áreas

O **git** gere uma diretoria base, e respetivas subdiretorias, a partir de uma diretoria **.git** nela criada com `git init`. Inicialmente nenhum ficheiro da diretoria base, ou das respetivas subdiretorias, é gerido pelo **git**, sendo designados por *untracked*. Para colocar um ficheiro sob a gestão do **git** é necessário adicioná-lo com `git add`, ficando numa área de espera. Caso se indique uma diretoria, todos os ficheiros dessa diretoria, incluindo as subdiretorias, ficam sob a gestão do **git**.

O **git** baseia-se na gestão de um conjunto de áreas, sendo os ficheiros movidos para áreas globalmente mais acessíveis à medida que se tornam mais definitivos. Assim, os ficheiros são editados pelo programador na área de trabalho (*sandbox*). Esta área não é mais que os ficheiros da diretoria base e respetivas subdiretorias.

Quando o ficheiro atinge determinados objetivos necessita de ser copiado (`git add`) para uma área de espera (*staging*). Os vários ficheiros envolvidos em determinada alteração vão sendo copiados, um de cada vez ou por atacado, para a área de espera. Se forem necessárias mais alterações a um ficheiro em espera basta copiar novamente o ficheiro, ou ficheiros, novamente alterados. Caso se pretenda, pode-se remover o ficheiro da área de espera ou substituí-lo por uma versão mais antiga existente no repositório (`git reset`). Contudo, deve-se ter especial cuidado ao copiar um ficheiro da área de espera para a área de trabalho (**`git reset --hard`**), pois o trabalho local será perdido se não houver uma salvaguarda prévia (**`commit`**).

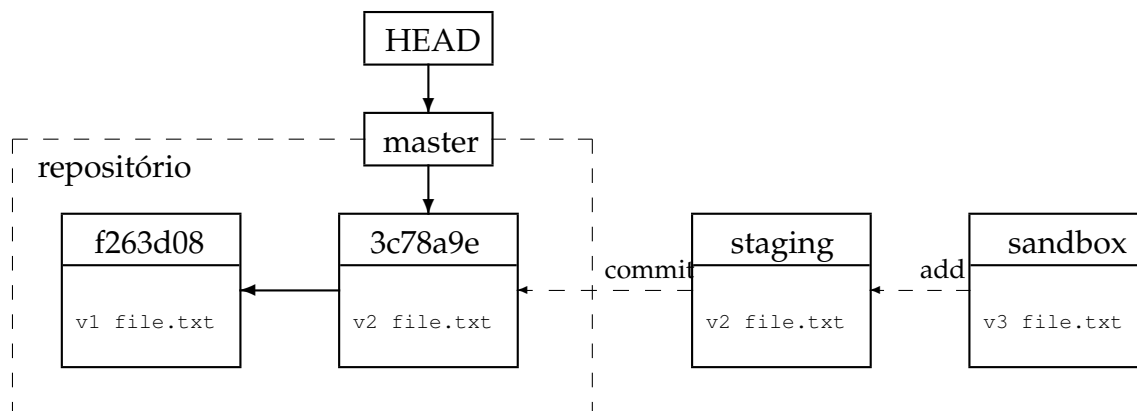
Quando os ficheiros na área de espera concluem determinada alteração, a alteração pode ser salvaguardada e registada com uma mensagem no repositório (`git commit`). As versões do repositório podem, posteriormente ser recuperadas por data, etiqueta (`git tag`) ou mensagem. O repositório pode ser partilhado com outros utilizadores, diretamente ou através de serviços, como o **github.com**.



3.2 Estrutura

De um ponto de vista da estrutura interna, o **git** pode ser visto como um gestor de árvores de ficheiros. A árvore de trabalho, ou *sandbox*, consiste na diretoria base e nas suas subdiretorias, diretamente manipuláveis pelo programador. As restantes árvores são manipuladas com comandos **git**. A área de espera, ou *staging*, não é mais que uma árvore onde os ficheiros aguardam, uns pelo outros, para serem enviados para o repositório (**`commit`**). O repositório guarda cada uma das árvores salvaguardadas (**`commit`**) num grafo, ou árvore nos casos mais simples. Em cada momento existe um ponteiro,

para um ponteiro, para a árvore corrente no repositório designada por HEAD. A movimentação dos ficheiros pode ser compreendida como a troca de ficheiros entre as três árvores de ficheiros: *sandbox*, *staging* e HEAD.



Na figura acima verificamos que as salvaguardas (**commit**) no repositório estão etiquetadas com um número hexadecimal. Internamente o **git** armazena informação como um sistema de ficheiros endereçados por conteúdo. O conteúdo é comprimido, em detrimento dos deltas, pois o espaço em disco é barato. Quando lhe é fornecido um conteúdo para armazenar, o **git** devolve uma chave. De momento é usada uma chave baseada numa função de dispersão criptográfica SHA-1 (*Secure Hash Algorithm*, 1995) com 160 bits (20 bytes ou 40 dígitos hexadecimais), sendo usada em protocolos como o TLS, SSL ou PGP. No caso do **git**, o SHA-1 é utilizado para garantir a integridade dos dados contra corrupção accidental. A maioria dos comandos **git** que requerem uma chave SHA-1 podem ser invocados apenas com os primeiros 7 dígitos hexadecimais da chave, desde que não haja conflitos com outros objetos com os mesmos 7 dígitos iniciais.

O **git** armazena quatro tipos de objetos: *blob* (*Binary Large Object*) para ficheiros, *tree* para diretorias, *commit* para salvaguardas e *tag* para etiquetas. Como curiosidade, pode calcular o valor da chave para um ficheiro com o comando `git hash-object -w file.txt`. Depois de salvarguardar o ficheiro, pode recuperar o seu conteúdo (armazenado em `.git/objects/`) com o comando `git cat-file -p fc6.....` (a opção `-t`, em vez `-p`, imprime o tipo de objeto, enquanto a opção `-s` imprime a dimensão). Os objetos são armazenados em `.git/objects/` comprimidos com a biblioteca **zlib** e podem ser descomprimidos utilizando a classe `java.util.zip.Deflater` do **java** ou a própria biblioteca `libz.a` do **C** (`zlib.h`).

3.3 Configuração

A configuração do **git** baseia-se em pares `nome-valor`, onde o nome é representado por uma ou mais componentes separadas por pontos (`.`).

O **git** utiliza três níveis de configuração:

git config --system aplica-se a todos os utilizadores do computador e localiza-se em `/etc/gitconfig` (em UNIX).

git config --global aplica-se ao utilizador individual e localiza-se em `~/.gitconfig` ou `~/.config/git/config` (onde `'~'` representa a diretoria principal do utilizador ou `$HOME`).

git config aplica-se ao repositório específico e localiza-se em `.git/config` a partir da diretoria base.

estes três níveis são processados sequencialmente pela ordem acima, sendo utilizado o último valor associado a cada nome.

Antes de se iniciar a utilização do **git** é necessário definir, para cada utilizador, o nome com que cada salvaguarda ficará registado e respetivo endereço de correio eletrónico:

```
git config --global user.name "Rui Silva"
git config --global user.email ist99999@tecnico.ulisboa.pt
```

Notar que são estes valores que serão considerados quando o repositório é envidado para o servidor *web* (por exemplo, **github.com**).

É ainda aconselhado definir o editor de texto usado para a mensagem nas salvaguardadas, pois caso a opção `bf -m` seja omitida, o editor é automaticamente invocado:

```
git config --global core.editor /usr/bin/nano
```

3.4 Exemplos de aplicação

Os comandos principais de manipulação de projetos em **git** podem ser subdivididos em comandos de construção (`init`, `add`, `commit` e `tag`), de informação (`status`, `log` e `show`) e de manipulação (`checkout`, `rebase`, `merge`, `branch` e `reset`):

init cria um novo projeto local.

add copia o(s) ficheiro(s) para a área de espera.

commit salvaguarda o projeto atual no repositório.

status mostra as diferenças entre as três árvores (ver seção 3.2).

log mostra o registo das salvaguardas.

show mostra objetos (*blobs*, *trees*, *tags* e *commits*).

tag atribui uma etiqueta a uma versão salvaguardada.

checkout move a cabeça (HEAD) entre ramificações e recupera ficheiros.

branch enumera, cria ou apaga ramificações do projeto.

merge une duas ou mais ramificações.

rebase coloca alterações salvaguardadas no fim da hierarquia.

reset gere a área de espera, movendo e apagando ficheiros.

A maioria destes comandos dispõe de inúmeras opções para especializar o seu comportamento.

Para exemplificar a sua utilização coordenada apresentam-se alguns exemplos de complexidade crescente.

3.4.1 Criação de três versões

Este exemplo cria um projeto desde a origem e produz três versões de um só ficheiro. Os comandos **status**, **log** e **show** são apresentados para se compreender a evolução do projeto e não são necessários para a criação das três versões do ficheiro. Após o segundo **commit** o projeto terá um aspeto semelhante ao das três árvores, apresentado acima. Para referência futura vamos considerar que as chaves das três versões são f263d08, 3c78a9e e c5a5028, respetivamente, embora cada execução do exemplo produza chaves distintas.

```
mkdir gittest/  
cd gittest/  
git init  
echo "V0 initial" >> file.txt  
git status  
git add file.txt  
git status  
git commit -m initial  
git status  
git log  
echo "V1 first change" >> file.txt  
git status  
git add file.txt  
git commit -m "first change"  
git log  
echo "V2 2nd change to the file" >> file.txt  
git add file.txt  
git commit -m 2nd_change
```

3.4.2 Etiquetar versões

A introdução de etiquetas permite referir uma salvaguarda com um nome simples em vez de uma chave SHA-1 ou uma data (um *post-it*). Em **git** existem etiquetas leves (*lighweight*) e anotadas (**-a**), elas próprias identificadas com uma chave SHA-1 e representadas como um ficheiro em `.git/refs/tags`. Nas etiquetas leves a chave aponta diretamente para o **commit** e são mais apropriadas para uso individual do programador

Uma etiqueta anotada é um objeto do tipo **tag** e inclui uma mensagem (**-m**), que pode ser impressa com o comando **show**, e contém um criador e uma data que pode ser distinta da do **commit** que refere. São as etiquetas anotadas que devem ser partilhadas com os restantes membros da equipa, podendo ser assinadas. Ao assinar uma

etiqueta anotada é garantida a identidade do signatário com uma chave PGP (opções **-s** ou **-u key-id**), o que permite detetar distribuições fraudulentas quando o repositório é copiado.

```
git log
git tag -a v1.2 -m prs-1.4 # tag this version
git show v1.2
git tag
git log
git status
git log --pretty=oneline
git tag -a v1.0 f263d08 -m original # tag a previous version
```

Para mover uma etiqueta para uma salvaguarda mais recente basta acrescentar a opção **-f** (*force*) às anteriores. Alternativamente, a etiqueta pode ser removida com a opção **-d** e depois inserida com o comando acima. No entanto, se a etiqueta já tiver sido empurrada (**push**) para um servidor remoto, deve ser primeiro removida remotamente (`git push origin :refs/tags/<tagname>`) e, uma vez movida localmente, novamente empurrada a etiqueta para o servidor (`git push origin master -tags`).

3.4.3 Recuperar versões anteriores

Uma das vantagens de um sistema de controlo de versões consiste na possibilidade de recuperar versões anteriores dos ficheiros salvaguardados. O comando **checkout** quando inclui uma versão e uma lista de ficheiros, por nome, copia essas versões para a área de trabalho, perdendo-se quaisquer alterações locais não salvaguardadas (**commit**) entretanto efetuadas. Por exemplo, um ficheiro apagado por engano (`rm file.txt`) pode ser recuperado com (`git checkout HEAD file.txt`). Caso se trate da versão corrente (HEAD) esta pode ser omitida (`git checkout file.txt`).

O comando **reset** permite copiar um ou mais ficheiros de uma versão para a área de espera e, com a opção **-hard**, da versão para a área de espera e para a área de trabalho. Neste último caso (**-hard**) perdem-se todas as alterações locais, caso não tenham sido salvaguardadas previamente. Por exemplo, `git reset HEAD~2 file.txt` coloca na área de espera o ficheiro de há duas versões atrás.

O comando **reset** pode ainda ser invocado sem a indicação de qualquer ficheiro, por exemplo `git reset HEAD~2`, mas neste caso recua-se duas versões (HEAD e **master**). Um posterior **commit** empurra os dois **commits** recuados para um ramo auxiliar que pode ser posteriormente eliminado. A utilização de **revert** em vez de **reset**, caso em que é criado um novo **commit** depois do atual com o mesmo conteúdo do indicado HEAD~2, mantém os **commits** recuados na sua posição inicial.

3.4.4 Desenvolvimento paralelo

De nada serve guardar as diversas versões de um projeto se não for possível inspecioná-las. O comando **checkout** permite saltar entre as diversas versões do projeto. O ar-

gumento é uma chave de salvaguarda (SHA-1) ou uma etiqueta. Para recuperar por data é necessário obter a chave correspondente, por exemplo `git rev-list -n 1 -before="2016-02-16 10:01"master`. As etiquetas móveis HEAD e **master** podem ser utilizadas, bem como referências relativas HEAD~2 (duas salvaguardas acima da HEAD).

Notar que HEAD e **master** não são a mesma coisa, embora HEAD possa apontar para **master** em certas alturas. A HEAD é a minha posição atual, enquanto o **master** representa a linha principal de desenvolvimento, aquela onde os restantes utilizadores estão ligados. Se devido a alterações de outros utilizadores, ou por distração, o **master** ficar perdido no meio do grafo, pode ser puxado com `git checkout master` seguido de `merge` para a versão pretendida `git merge versão`. Se o **master** estiver na mesma linha de desenvolvimento do local de destino, não são necessárias alterações, logo não é necessário nenhum **commit**, e designa-se por *fast-forward merge*.

O comando **branch** permite criar um ramo para desenvolvimento paralelo. Este ramo permite fazer experiências que podem, ou não, vir a ser úteis, independentemente do número de salvaguardas efetuadas no ramo. O comando **branch** e o respetivo **checkout** podem ser condensados, e a versão de partida omitida for HEAD, em `git checkout -b first_fork`.

```
cat file.txt
git checkout 3c78a9e
cat file.txt
git branch fork_first 3c78a9e
git checkout first_fork
echo "V1.1 from forked first" >> file.txt
git add file.txt
git commit -m first_fork # SHA-1 = 4ffd0ea
git log --graph --oneline --decorate --all
```

Com a introdução do desenvolvimento paralelo, a opção `-graph` do comando **log** permite ter uma representação rudimentar do grafo de versões no repositório. Para simplificar a invocação deste comando para **git graph**, incluir-se na secção `[alias]` do ficheiro de configuração `~/.gitconfig` a definição

```
graph = !"git log --graph --oneline --decorate --all"
```

Caso o trabalho desenvolvido no ramo seja positivo deve ser integrado na linha principal de desenvolvimento (**master**). Os ramos que não desenvolveram trabalho útil ficam com as pontas soltas, embora possam ser recuperados em qualquer momento. Para remover definitivamente o ramo, usa-se a opção **-d** (`git branch -d name`). O comando **merge** permite integrar as alterações do ramo corrente (HEAD) no ramo pretendido, em geral **master** (`git merge master`). Se não surgirem conflitos, alterações diferentes para uma mesma linha de código, o comando conclui com sucesso. Um conflito surge quando a mesma linha de um mesmo ficheiro tem dois conteúdos distintos. Neste caso, a terceira linha do ficheiro contém um texto diferente entre a terceira versão (c5a5028 ou **master**) e a versão do `first_fork` (4ffd0ea ou HEAD). Ao executar

o comando **git merge master** é detetado o conflito e invocado o editor anteriormente configurado. Caso este ficheiro seja escrito tal qual é fornecido (sem alterações), os conflitos são assinalados nos respetivos ficheiros com as marcas <<<<<<, ===== e >>>>>>. Compete ao programador decidir qual o aspeto final do ficheiro, ou seja qual das duas modificações (ou composição destas) sobrevive. Cuidado, pois se os ficheiros que incluem conflitos não forem todos corrigidos, a versão final irá incluir as marcas acima como definitivas, o que não é uma sequência válida em qualquer linguagem. Um **commit** permite concluir o **merge**, tornando definitivas as alterações. A opção **-a** efetua implicitamente o **git add** dos ficheiros modificados e o **git rm** dos ficheiros removidos antes do **commit**.

```
git merge master
git status # show conflicts
edit file.txt
git commit -am "conclude merge" # SHA-1 = 9b3ce96
git log --graph --oneline --decorate --all
git checkout master
git merge fork_first # move master to fork_first
```

O desenvolvimento paralelo com **branch** e **merge** é simples e não destrutivo, no sentido em que todas as alterações efetuadas são mantidas imutáveis. Este facto é importante quando for necessário descobrir a origem de um problema passado. Contudo, numa equipa grande, com muitos ramos simultaneamente ativos, o grafo pode tornar-se complexo e incompreensível.

3.4.5 Rebase

O comando **rebase** permite realizar a mesma operação que o comando **merge** mas linearizando as alterações do ramo a integrar na linha de desenvolvimento principal. O grafo torna-se mais simples, no limite é uma linha contínua de desenvolvimento. Contudo, ao integrar os **commits** do ramo na linha principal, estes são modificados por forma a refletir o facto de agora terem como base o **master**. Na realidade, é como se as alterações efetuadas no ramo fossem introduzidas a partir do **master** e não a partir do local de onde foi efetuado o **branch**.

No modo interativo, opção **-i**, o programador tem a opção de decidir quais são os **commits** realizados no ramo que ficarão na versão final em caso de conflito. O editor é invocado automaticamente e o programador decide quais os **commits** que são aproveitados (**picked**). Caso contrário, sem conflitos, o **rebase** conclui com sucesso. O comando `git rebase -abort` desiste do **rebase**, enquanto o comando `git rebase -continue` aceita as alterações efetuadas. Ficando o processo concluído com a colocação da etiqueta **master** na versão *rebased*.

Ao criar um ramo com três alterações, o comando **rebase -i** permite escolher quais dos três **commits** ficam na versão final linearizada. Caso se opte por manter as três, escrevendo o ficheiro com os três **pick**, e corrigindo o `file.txt` por forma a manter

todas as alterações já efetuadas, o resultado do **rebase** inclui as três versões do ramo alteradas (ver HEAD~1 e HEAD~2).

```
git branch rebase_change 3c78a9e #first_change
git checkout rebase_change
cat file.txt
echo "branch rebase 1" >> file.txt
git commit -am "rebase V1.1.1" # SHA-1 = d9a034f
echo "branch rebase 2" >> file.txt
git commit -am "rebase V1.1.2" # SHA-1 = 97bf09e
echo "branch rebase 3" >> file.txt
git commit -am "rebase V1.1.3" # SHA-1 = 57c164f
git rebase -i master
git status
edit file.txt
git add file.txt
git rebase --continue # SHA-1 = 31b7abe
git checkout master
git merge rebase_change # move master to rebase_change
git log --graph --oneline --decorate --all
cat file.txt
git checkout HEAD~2
cat file.txt
```

Contudo, o comando **rebase** permite eliminar por completo, sem possibilidade de recuperação, os **commits** intermédios do ramo, criando a ilusão que todo o trabalho foi efetuado num só passo após o **master**. Claro que uma das grandes vantagens de um sistema de controlo de versões foi perdida pois existe trabalho intermédio que ficou irremediavelmente perdido. Na realidade, trata-se de uma espécie de **purge** do **vax-vms** falado no início desta secção. Por outro lado, a história do projeto, embora adulterada, fica mais limpa e linear.

A regra de ouro do **rebase** consiste em nunca fazer **rebase** de ramos públicos. Uma vez que outro utilizador pode estar a trabalhar no **master**, por exemplo, não só se perde a linha de desenvolvimento principal como, potencialmente, todos os **commits** efetuados desde o **branch** em questão. Ao tentar efetuar um **push** para o servidor (ver seção 5) ocorre um erro, que pode ser contornado com a opção **-force** que reposiciona o **master** para toda a equipa. Assim, o ramo pode ser *rebased* para o **master** mas nunca o contrário.

4 Persistência e transações: fénix framework

A **fénix framework** (ver <https://fenix-framework.github.io/>) é um ambiente persistente e transacional de suporte ao desenvolvimento de aplicações em Java. A persistência permite salvar e recuperar objectos a partir de um armazenamento persistente, no caso uma base de dados relacional **mysql**. O ambiente transacional permite que vários objectos sejam alterados simultaneamente, ou ignorar um conjunto de alterações (*rollback*). A **fénix framework** oferece o mapeamento para a base de dados de uma forma automática, libertando o programador dessa tarefa, embora se perca o acesso direto a mecanismos específicos da base de dados.

O modelo persistente e transacional é especificado através de uma linguagem (*Domain Modeling Language* ou **dml**). As classes e suas relações especificadas no ficheiro **dml** são tornadas persistentes, quando as suas alterações são efetuadas numa transação. Uma transação não é mais que um método Java com a anotação `@Atomic`, que ao terminar realiza o correspondente **commit**. Notar que todas as classes e relações (referências, listas, ...) não incluídos na especificação **dml** não são salvaguardadas na base de dados.

Resumidamente, a utilização da **fénix framework** presuppõe a especificação num ficheiro `.dml` do domínio da aplicação, ou seja um conjunto de classes e suas relações. Este ficheiro é compilado, pelo compilador da linguagem **dml**, produzindo para cada classe `X` especificada em **dml**, duas classes Java `X_Base` e `X`. A classe `X_Base` contém os atributos e as relações especificados em **dml** e encontra-se sob o controlo da **fénix framework**, não devendo ser modificada. A classe `X`, que é gerada apenas se não existir já (ou seja, apenas da primeira vez), encontra-se sob o controlo do programador. É nesta classe que deve ser colocada a lógica de negócio, ou seja, os métodos que manipulam os objectos e as suas relações. Notar que qualquer atributo colocado na classe `X` não será tornado persistente, consequentemente não é posteriormente recuperável. Uma vez construída a lógica de negócio, os ficheiros Java podem ser compilados e a aplicação executada.

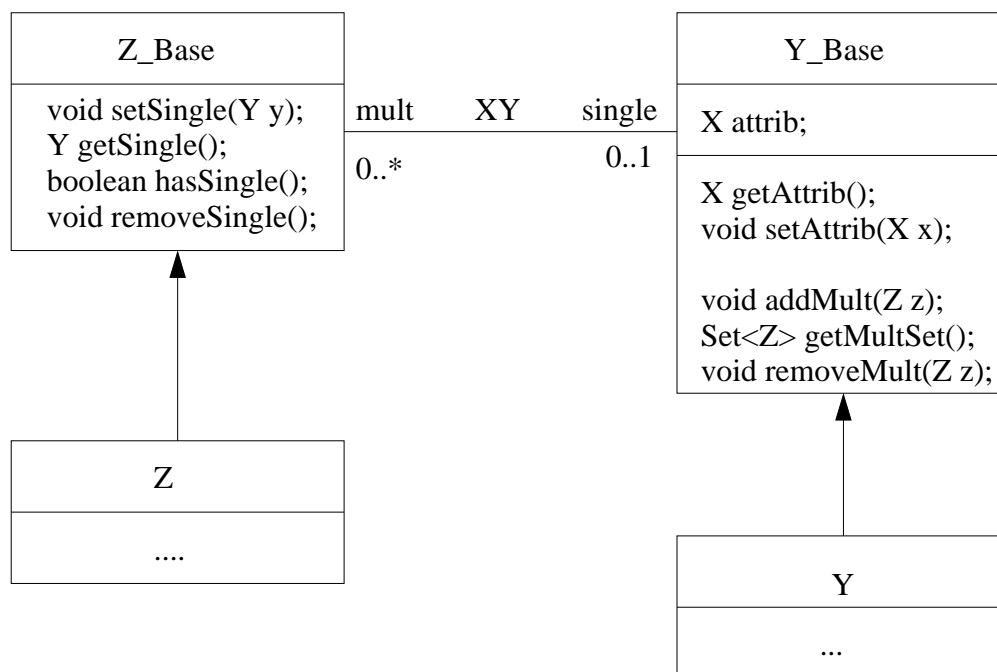
4.1 Domain Modeling Language

A especificação do domínio persistente da aplicação consiste num ficheiro `.dml` com declarações de classes e suas relações. Em **dml** as classes contêm apenas atributos, ou seja o conteúdo de cada objeto da classe a salvar. Estes atributos podem ser:

- os tipos primitivos de Java (`boolean`, `byte`, `char`, `short`, `int`, `long`, `float`, `double`) e respetivas classes (`Boolean`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float`, `Double`).
- as classes de agregados `String` e `bytearray` (correspondente a `byte[]` em Java).
- as classes de manipulação de datas (`DateTime`, `LocalDate`, `LocalTime`, `Partial`) da biblioteca Joda Time (<http://www.joda.org/joda-time/>).

As relações entre as classes são bidirecionais e especificadas com a palavra reservada `relation` e o respetivo nome, bem como as classes envolvidas na relação. Para cada classe envolvida na relação é indicado um nome pelo qual a classe é referida na relação e uma multiplicidade. A multiplicidade representa o número de ocorrências da classe na relação, podendo ser um número não negativo, um intervalo (por exemplo, `2..5`) ou `*` para representar *zero ou mais*. Se a multiplicidade for omitida assume-se `0..1`.

```
class Y { X attrib; }
class Z,
relation YZ { Y playsRole single; Z playsRole mult { multiplicity *; } }
```



A classe `.pt.ist.fenixframework.DomainRoot` representa a raiz da base de dados. Só os objectos direta ou indiretamente ligados à raiz é que são salvaguardados. A declaração da `package`, que tal como em Java, a existir, deve ser a primeira declaração da especificação, indica a sequência de diretórios onde serão colocadas as classes sob a responsabilidade do programador.

Associados a cada atributo declarado em **dml** são gerados dois métodos públicos `getX()` e `setX(X x)`, que permitem obter e alterar o valor do atributo `X`. A invocação do método permite, ler da base de dados, ou marcar o atributo em questão para posterior salvaguarda (*commit*). Notar que a primeira letra do atributo é tornada maiúscula no nome do método, caso tenha sido declarada minúscula.

No caso de relações em que o papel (`playRole`) da classe `Y` tem cardinalidade `0..1`, são criados três métodos públicos `getY():Y`, `setY(Y y)` e `removeY()` na outra classe da relação. O valor nulo (`null`) é devolvido pelo método `getY():Y` se não existir um objeto associado.

No caso de relações com mais de um elemento, o método `add` passa a designar-se por `addZ(Z z)`, o método `get` passa a devolver um objeto do tipo `Set`, sendo desig-

nado por `getZSet() : Set<Z>`. Neste caso, o método `remove` passa a receber o objeto a remover `removeZ(Z z)`.

4.2 Construção do domínio

A construção da aplicação inicia-se tendo por base o arquétipo (ver seção 2) base da **fénix framework**, onde o `groupId` representa a package da aplicação a criar e o `artifactId` é o nome do diretório de desenvolvimento:

```
mvn archetype:generate
  -DarchetypeGroupId=pt.ist
  -DarchetypeArtifactId=fenix-framework-application-archetype-clean
  -DarchetypeVersion=2.0
  -DarchetypeRepository=https://fenix-ashes.ist.utl.pt/maven-public
  -DgroupId=example
  -DartifactId=hellofenix
  -Dversion=1.0-SNAPSHOT
  -DinteractiveMode=false
```

A indicação da localização do arquétipo é necessária por não se tratar de um arquétipo da biblioteca da ferramenta **maven**.

O primeiro passo na especificação do projeto consiste em definir o estado persistente da aplicação no ficheiro `.dml` em `src/main/dml/hellofenix.dml` (`hellofenix` é o nome do artefato indicado no arquétipo). Por exemplo, para guardar uma lista de nomes de pessoas, a especificação **dml** pode ser:

```
package example;
class Person {
    String name;
}
relation KnownPeople {
    .pt.ist.fenixframework.DomainRoot playsRole root;
    Person playsRole people { multiplicity *; }
}
```

Para produzir uma versão inicial mínima das classes indicadas na especificação **dml** basta compilar o projecto (`mvn compile`). Alternativamente, pode-se construir as classes de raiz, tendo em atenção que estas devem sempre herdar da respetiva classe `_Base`:

```
class X extends X_Base
```

A construção da classe `Person` deve ligar o objeto à raiz, para que não existam objetos soltos na aplicação. Adicionalmente, realiza-se um método para imprimir o nome atribuído.

```
package example;
import pt.ist.fenixframework.DomainRoot;
public class Person extends Person_Base {
    public Person(String name, DomainRoot root) {
        setName(name);
        setRoot(root);
    }
    public void sayHello() {
```

```

        System.out.println("Hello, I'm " + getName() + "!");
    }
}

```

A classe principal é responsável pela gestão das transações (@Atomic). O método `addNewPeople` acrescenta novas pessoas à base de dados, enquanto o método apenas de leitura `greetAll` imprime o nome de todas as pessoas registadas.

```

package example;
import pt.ist.fenixframework.*; // for Atomic, DomainRoot, FenixFramework
public class Main {
    public static void main(String [] args) {
        addNewPeople(args);
        greetAll();
    }
    @Atomic
    public static void addNewPeople(String[] args) {
        DomainRoot root = FenixFramework.getDomainRoot();
        for (String name : args)
            new Person(name, root);
    }
    @Atomic
    public static void greetAll() {
        DomainRoot root = FenixFramework.getDomainRoot();
        for (Person p : root.getPeopleSet())
            p.sayHello();
    }
}

```

Agora é necessário compilar o código e correr a aplicação resultante:

```
mvn clean package exec:java -Dexec.mainClass=example.Main -Dexec.args="Rui"
```

A aplicação, depois de compilar e construir o pacote, executa e imprime `Hello, I'm Rui!`.

Executando novamente, agora sem compilar, mas indicando `Maria` como argumento:

```
mvn exec:java -Dexec.mainClass=example.Main -Dexec.args="Maria"
```

A aplicação só imprime `Hello, I'm Maria!` pois estamos a usar memória como backend, logo não existe persistência.

4.3 Utilização da base de dados

Para que os objetos sejam tornados persistentes na base de dados, é necessário configurar o acesso à base de dados. Para tal é necessário acrescentar o ficheiro de recursos `src/main/resources/fenix-framework-jvstm-obj.properties`:

```

appName=INFER_APP_NAME
dbAlias=//localhost:3306/DATABASE?useUnicode=true
&characterEncoding=UTF-8
&clobCharacterEncoding=UTF-8

```

```

&zeroDateTimeBehavior=convertToNull
dbUsername=USERNAME
dbPassword=PASSWORD

```

onde DATABASE é o nome da base de dados a utilizar (por exemplo, `hello` para esta experiência), USERNAME é o nome do utilizador da base de dados em nome de quem a aplicação vai efetuar o acesso (por exemplo, `root`) e PASSWORD é a palavra passe do utilizador (`rootroot`, de acordo com as instruções de instalação da disciplina).

Como a aplicação apenas utiliza a base de dados indicada é necessário criar a base de dados antes da primeira execução:

```
mysql -uUSERNAME -pPASSWORD -e "create database DATABASE;"
```

Sempre que o ficheiro de especificação é alterado, a estrutura da base de dados tem de ser alterada em conformidade. Para reiniciar a aplicação com uma base de dados limpa, pode destruir a base de dados (`drop database DATABASE;`) e voltar a criar com o comando acima. Em alternativa, pode ser utilizado o `dbclean-maven-plugin` (ver [github](#) da disciplina) que cria ou limpa a base de dados com a informação do ficheiro de recursos, com o comando **`mvn dbclean:dbclean`**.

Finalmente, é necessário configurar o **maven** para juntar a aplicação e a base de dados num processo de compilação e execução integrado. Infelizmente não existe um arquétipo pré-definido para este caso na **fénix framework**. Como a complexidade da `pom.xml` (*Project Object Model*) sai fora do âmbito deste documento, o programador deve utilizar um modelo. Sugere-se procurar em *Gist* do [github.com](#) por `fenix-framework-jvstm-obj.properties` ou Seguidamente é necessário editar o ficheiro e alterar os `FIXME`, o `groupId`, o `artifactId` e as referências à `mainClass` para a package escolhida para desenvolver o projeto. Alternativamente, pode adaptar o exemplo **phonebook** disponibilizado.

Depois de montado o projeto é necessário compilar e executar:

```
mvn clean package exec:java -Dexec.args="Paulo Rui"
```

Desta vez, ao executar uma segunda vez a aplicação, os nomes da execução anterior são recuperados da base de dados e impressos juntamente com os novos:

```
mvn exec:java -Dexec.args="Ana Joana Beatriz"
```

Executando a aplicação sem argumentos é possível observar os nomes guardados:

```
mvn exec:java
```

4.4 Herança e eliminação de objetos

A geração, pela **fénix framework**, de uma classe `_Base` por classe especificada, significa que em caso de herança existe uma classe `_Base` que se interpõe entre as duas. Por exemplo, se na especificação **dml** `X` herda de `Y`, então no código gerado `X` herda de `X_Base` e `X_Base` herda de `Y` (e `Y` herda de `Y_Base`). Como os construtores do código gerado não têm argumentos, torna-se impossível invocar o construtor da classe `Y` a partir do construtor da classe `X`. Notar que o construtor da classe `Y` com argumentos pode

ainda ser útil se esta classe for diretamente instanciada. Para tal efeito sugere-se a utilização de um método auxiliar (por exemplo, `init`) a ser invocado dentro do construtor de `X` e definido em `Y`. A declaração do método como `protected` obriga à invocação de métodos da hierarquia da classe, como os construtores da classe, para realizar a sua inicialização. Claro que os nomes escolhidos não devem coincidir com os nomes utilizados pela **fénix framework** na geração das classes base. O mesmo processo pode ser utilizado para quaisquer outros métodos além dos construtores.

Eliminação de objetos Para remover entidades é necessário quebrar a ligação:

```
public static void removePeople(String name) {
    DomainRoot root = FenixFramework.getDomainRoot();
    for (Person p : root.getPeopleSet())
        if (name.equals(p.getName())) {
            p.remove();
            return;
        }
    System.out.println(name + ": not in set (can not remove)");
}
```

A relação pode ser quebrada com `root.getPeopleSet().remove(p);` ou com `p.setRoot(null);`. Como a relação é bidirecional, ao eliminar a ligação a partir de um dos lados, a ligação fica quebrada.

Contudo, uma inspeção da respetiva tabela na base de dados permite verificar que o objeto ainda está armazenado, apenas não está acessível pois não existem ligações até ele a partir da raiz. Para eliminar o objeto da base de dados é necessário efetuar um `this.deleteDomainObject()` depois de terem sido quebradas todas as ligações do objeto. Como o método `deleteDomainObject()` só pode ser acedido dentro do próprio objeto (`protected`) opta-se por ser o próprio objeto a eliminar as suas ligações e, finalmente, remover-se da base de dados. Não existindo mais ligações com o objeto, este será eliminado da memória pelos mecanismos habituais do ambiente de execução da linguagem Java (*garbage collection*).

```
public void remove() {
    setRoot(null);
    deleteDomainObject();
}
```

Uma solução simples consiste em eliminar os nomes iniciados pelo carácter `' - '` (por exemplo, `-Rui`) pelo que basta alterar o método

```
@Atomic
public static void addNewPeople(String[] args) {
    DomainRoot root = FenixFramework.getDomainRoot();
    for (String name : args)
        if (name.charAt(0) == ' - ')
            removePeople(name.substring(1));
        else
            new Person(name, root);
}
```


4.5 Gestão de transações

A `fénix framework` não tem suporte para transações aninhadas, transações dentro de transações ou *nested transactions*. Assim, métodos com a anotação `@Atomic` não devem chamar outros com a mesma anotação. O programador deve organizar o código de forma que este tipo de situações não se veirifique.

Por omissão, a `fénix framework` inicia cada transação como uma transação de leitura. Quando verifica que são pedidas alterações, a transação aborta e é reiniciada como uma transação de escrita. Por exemplo, se no início do método `addNewPeople` for impressa uma mensagem no terminal, verifica-se que sempre que se adiciona uma nova pessoa a mensagem é impressa duas vezes. No entanto, dada a estrutura da aplicação, se não for adicionada ou removida qualquer pessoa, o método é invocado e a mensagem é impressa uma só vez no terminal (o corpo do ciclo `for` não executa nenhuma vez). O programador deve ter em atenção quaisquer efeitos secundários que a aplicação possa exibir sempre que uma transação de leitura é reiniciada como a de escrita.

Em especial para efeitos de teste, pode ser útil impedir que as transações terminem com a alteração da base de dados. Na execução de uma bateria de testes é desejável que as condições sejam idênticas para todos os testes de uma situação para que não haja dependência entre os testes. Para tal, em vez da utilização da anotação `@Atomic` pode-se controlar mais finamente o funcionamento das transações. Por exemplo, ao iniciar uma transação com `FenixFramework.getTransactionManager().begin(false)` pode-se controlar a linha do código onde a transação se inicia, sem que esta tenha de coincidir com início de um método. Assim, uma transação pode ser iniciada num método e terminada noutro ou permitir duas ou mais transações dentro de um mesmo método. Adicionalmente, com o argumento `true`, a transação é iniciada em modo leitura e a exceção `WriteOnReadError` será lançada se houver uma tentativa de escrita. Substituindo o método `begin` por `.commit()`, `.rollback()`, `.suspend()` ou `resume()`, conclui-se, aborta-se, suspende-se ou continua-se uma transação.

5 Software web hosting: github

O **github.com** é um serviço de *Web Hosting Compartilhado* (*collaborative software development*) para projetos que usam o **git** como ferramenta de controlo de versões distribuídas (*distributed revision control*), permitindo a gestão do código (*source code management* ou SCM) através de um navegador *web* (*web browser*). No **github.com** os repositórios públicos (onde qualquer utilizador tem acesso) são gratuitos, enquanto o repositórios privados são pagos. Outros serviços, como **bitbucket.org** ou o **gitlab.com**, permitem repositórios públicos e privados gratuitos. Estes serviços oferecem funcionalidades de uma rede social adaptada ao desenvolvimento de código (*social coding*), como *wiki*, seguidores, grupos (equipas), *issues*, *clipboard* (Gists ou snippets) com uma interface gráfica em rede.

A ferramenta **git** (ver seção 3) inclui comandos para interagir com o servidor de **git**, de **github.com** ou outro. A salvaguarda das alterações (*commits*) para o servidor (*push*) permite que estas fiquem visíveis para os outros utilizadores. Notar que todas as alterações locais são enviadas e não apenas a última. Inversamente, podem-se obter (*fetch*) todas as alterações registadas no servidor por forma a posteriormente efetuar a fusão (*merge*) da versão de trabalho com uma das versões registadas, em geral a última.

A abordagem de desenvolvimento colaborativo depende da publicação das salvaguardas locais do projeto (**commit**) num repositório remoto, para poder ser obtido ou atualizado por outros. Repositórios remotos, geridos por servidores, são versões do projeto. Cada projeto pode ter vários repositórios remotos. Os repositórios são identificados e acedidos por URL, podendo ser acedidos por **https** ou **ssh**, por exemplo:

https `https://github.com/user/repo.git`

ssh `git@github.com:user/repo.git`

O **git** associa ao URL remoto um nome simbólico, mais simples que um endereço completo, frequentemente designado por **origin** (ver `git remote`).

Os principais comandos **git** de manipulação remota são:

clone cria uma cópia local de um projeto remoto.

push envia para o servidor as alterações locais.

remote associa um nome local a um projeto remoto.

fetch copia as alterações do servidor para o projeto local.

pull copia (*fetch*) e junta (*merge*) as alterações do servidor.

5.1 Os primeiros passos

Primeiro deve criar uma conta no **github.com**, devendo indicar o endereço de *email* que indicou no `git config` (ver seção 3.3). É quanto basta para aceder ao servidor via `http` (e `https`).

Para aceder aos servidores por **ssh** é necessário enviar previamente uma chave SSH para o servidor. Se já criou anteriormente uma chave SSH, da qual conhece a palavra passe, basta adicionar o conteúdo do ficheiro que contém a parte pública, por exemplo `~/.ssh/id_rsa.pub`, nas seções das configurações da sua conta **github**. Para

criar uma chave SSH deve executar o comando `SSH-KEYGEN`, indicando uma palavra passe que deve memorizar, sendo criados dois ficheiros contendo a parte privada e a pública da chave SSH, em geral `~/.ssh/id_rsa.pub` e `~/.ssh/id_rsa` respetivamente.

O projeto pode ser criado através da interface gráfica, podendo depois associar os ficheiros do projeto a desenvolver de duas formas. Pode optar por clonar o repositório e copiar para ele os ficheiros, que já possui ou que vai criando:

```
$ git clone https://github.com/user/proj.git
$ cp myfiles proj
$ cd proj
$ git add . # mark all files to commit
$ git commit -m "message" # commit to local repository
$ git push # send to github
```

Alternativamente, pode associar um repositório local **git** com o projeto **github** que acabou de criar:

```
$ cd /path/to/my/repo
$ git remote add origin https://github.com/user/proj.git
$ git push -u origin master
```

Se o repositório já contiver etiquetas e referências, estas devem ser enviadas separadamente:

```
$ git push -u origin --all # pushes repo and refs (1st time)
$ git push -u origin --tags # pushes any tags
```

Caso se pretenda efetuar o acesso com **ssh** em vez de **https**, os endereços devem ser substituídos por `git@github.com:user/repo.git`, embora necessite da chave SSH em ambos os casos.

Para não necessitar de introduzir a palavra chave em cada acesso, esta pode ser armazenada em `/.netrc`:

```
machine github.com
  login <user>
  password <password>
```

Outros serviços semelhantes funcionam da mesma forma, com algumas diferenças no formato dos endereços dos projetos, por exemplo `git@gitlab.com:user/proj.git` em **gitlab.com** ou `https://user@bitbucket.org/user/proj.git` em **bitbucket.org**.

5.2 Rebase remoto

O desenvolvimento de software colaborativo (*collaborative software development*) implica que as alterações efetuadas por uns utilizadores vão influenciar o comportamento do código desenvolvidos por outros. Assim, é de extrema importância executar testes ao software antes de enviar alterações para o servidor (**github** ou outro).

Se as alterações efetuadas por dois, ou mais, utilizadores não afetarem a mesma zona do ficheiro, em geral as mesmas linhas, a ferramenta **git** consegue fundir as alterações.

Isto não significa que ambas as alterações funcionem corretamente em conjunto, mesmo que não apresentem problemas cada uma por si só.

Consideremos que dois utilizadores (**a** e **B**) obtêm cópias de um mesmo repositório:

```
a> git clone http://github.com/user/hello.git
B> git clone http://github.com/user/hello.git
```

Para simplificar, este projeto tem um único ficheiro, tipo `hello world`, que imprime a mensagem `Olá pessoal`.

O utilizador **a** resolve alterar a mensagem para `Olé pessoal`:

```
a> cd hello/
a> edit src/hello/hello.java
a> git status
a> git commit -am "Olé pessoal"
a> git push
```

e submete as alterações ao servidor sem problemas.

Enquanto isso, o utilizador **B** altera a mensagem para `Olá touro`:

```
B> cd hello/
B> edit src/hello/hello.java
B> git commit -am "Olá touro"
B> git push
```

que ao submeter as alterações ao servidor recebe um erro, pois o ficheiro já não está igual ao que ele foi buscar devido às alterações de **a**. Consequentemente é necessário ir buscar as alterações no servidor e unir (*merge*) com as alterações locais:

```
B> git pull --rebase
```

Se houver alterações inconsistentes na mesma zona de código a união não é realizada (error: Failed to merge in the changes.) e os ficheiros com inconsistências são assinalados (Merge conflict in `hello.java`). Assim, ele deverá editar o ficheiro, ou ficheiros, com conflitos e optar por uma solução.

```
package hello;
public class hello {
    public static void main(String[] args) {
<<<<<<< HEAD
        System.out.println("Olé pessoal!");
=====
        System.out.println("Olá touro!");
>>>>>>> touro
    }
}
```

Notar que esta solução pode ser a que ele escreveu, a que o utilizador **a** escreveu, ou uma solução alternativa que incorpore ambas as alterações.

```
B> git pull --rebase
B> edit src/hello/hello.java # Olé touro
B> git add src/hello/hello.java
B> git rebase --continue
B> git push
```

Caso as alterações sejam noutros ficheiros ou zonas de código, o comando `git pull -rebase` já consegue fazer a união (*merge*) das duas versões sem conflitos, pelo que basta enviar o código resultante para o servidor: `git push`. Notar que antes de fazer qualquer `git push` deve testar o código a enviar, por exemplo `mvn test` (ver seção 2).

5.3 Ramos remotos

Em **git** os ramos (*branches*) são pouco pesados, pois apenas é armazenada a referência para a salvaguarda (*commit*). Quando é efetuada a salvaguarda, esta fica associada ao ramo corrente.

O comando `git branch` permite saber qual o ramo atual. Para criar um novo ramo basta indicar o nome pretendido: `git branch ramo`. O comando `git checkout ramo` permite tornar este ramo corrente.

Depois de efetuadas alterações no ramo, caso se pretenda unir as alterações ao ramo principal (*master*), deve-se ir para *master* (`git checkout master`) e incorporar as alterações do ramo em *master* (`git merge ramo`).

Para integrar com o código do servidor é necessário ir buscá-lo com `git pull`, agora sem a opção `-rebase`. Caso surjam conflitos, é necessário resolvê-los e depois enviar as alterações para o servidor: `git push -u origin master`

Se as alterações introduzidas forem problemáticas então pode-se efetuar na interface gráfica *web* do servidor um *pull request* (em linguagem **github**), ou *merge request* (em linguagem **gitlab**), para envolver os restantes membros da equipa na aceitação das soluções encontradas para os conflitos.

5.4 Fluxos de trabalho: *workflows*

Os fluxos de trabalho são definidos pela organização, ou pela equipa, como forma de coordenação do trabalho realizado pelos membros da equipa envolvida no desenvolvimento. Os fluxos de trabalho pretendem evitar a proliferação descontrolada de ramos de trabalho, introduzindo alguma estrutura na evolução temporal das diferentes versões do projeto.

5.4.1 Central workflow

A forma mais simples de fluxo de trabalho consiste em considerar um único ramo de desenvolvimento: o *master*. Notar que, nos repositórios locais, os diferentes utilizadores vão mantendo versões desatualizadas e com alterações locais do *master* comum remoto, em geral designado de *origin*.

Neste fluxo de trabalho a integração do código desenvolvido é sempre efetuada no *master*, logo tem de ser realizada à custa de sucessivos *rebase* (ver seção 5.2).

Esta solução simples torna-se particularmente atrativa se nenhum dos membros da equipa nunca estiver muito atrasado face ao *master*. Isto obriga a que assim que uma

alteração está utilizável, ou pelo menos não perturba o trabalho dos restantes, deve ser imediatamente integrada no `master`. Este processo de desenvolvimento, designado por integração contínua (*continuous integration*), caracteriza-se por sucessivas atualizações do servidor, frequentemente várias vezes num só dia. Alguns servidores, como o **github** ou **gitlab**, permitem que a própria compilação e teste do projeto seja efetuada no próprio servidor mediante a ativação de um *runner* (**gitlab**).

5.4.2 Feature branching

Neste fluxo de trabalho cada funcionalidade é desenvolvida num ramo independente. Esta solução permite que diversos membros da equipa trabalhem em funcionalidades específicas, sem perturbar o trabalho dos restantes. Quando prontas, as funcionalidades são fundidas (*merged*) no código principal, pelo que o `master` nunca contém código não operacional (*broken code*). Esta abordagem facilita a integração contínua (*continuous integration*), em especial quando uma funcionalidade é desenvolvida por mais de um membro da equipa e partilham o código de desenvolvimento no servidor comum ao projeto.

Nesta solução, os *pull requests*, ou *merge requests*, podem ser efetuados do ramo específico. A integração no `master` só se realiza quando é obtido um acordo.

5.4.3 Strict branching

Para projetos de maior dimensão, o *strict branching* oferece uma estrutura robusta, onde existem diversos ramos com funções específicas:

- `master` é o ramo principal que contém o código instalado ou pronto para ser instalado no cliente.
- `release` é o ramo onde o código é testado (testes de sistema) e preparado para ser distribuído e onde é atualizada a documentação. O ramo pode ser apagado quando é integrado no `master`.
- `develop` é o ramo de desenvolvimento onde as funcionalidades completas são integradas testadas em conjunto (testes de integração).
- ramos para cada uma das funcionalidades em desenvolvimento como no *feature branching*.

As novas funcionalidades, bem como as alterações a outras já existentes, vão subindo na hierarquia até chegarem ao `master` e poderem ser instaladas no cliente.

Notar que nenhum ramo deriva do `master`. Os diversos ramos, `release` ou funcionalidades, derivam do `develop`, sendo o `release` integrado no `master`. Apenas o ramo `develop` foi o único que derivou do `master`, no início do projeto.

5.4.4 Hotfix branching

Este fluxo de trabalho melhora a solução anterior com a adição de mais um ramo, em geral designado por `maintenance` ou `hotfix`. Este ramo é utilizado para introdu-

zir correções urgentes, que não podem esperar pela distribuição seguinte (*release*). Apenas este ramo pode derivar diretamente do `master`. Assim, que a correção está completa, o ramo é integrado quer no `master` quer no `develop`, por forma a fazer parte das distribuições seguintes. O ramo `master` deve, também, ser etiquetado com um número de versão atualizado.

5.4.5 Fork workflow

Este fluxo de trabalho utiliza diversos repositórios em vez de um só para as várias equipas envolvidas no desenvolvimento. Nesta abordagem, cada membro, ou equipa, possui o seu repositório. Assim, cada membro possui dois repositórios: um repositório local privado e um repositório público.

A principal vantagem reside no fato de o processo de integração poder ser realizado por uma equipa de integração que recolhe as contribuições dos repositórios públicos e as coloca num repositório de integração. Desta forma, não são os membros das equipas que têm de, sincronizadamente, enviar as alterações para o servidor. O seu repositório público contém sempre a versão pública e completamente funcional mais recente.

A função do integrador, ou equipa de integração, também fica mais clara, pois apenas ele tem acesso ao código de distribuição. O desenvolvimento pode, assim, incluir equipas de terceiros às quais não são concedidos privilégios especiais. Esta abordagem adapta-se igualmente a projetos de código aberto (*open source*).

Para criar as cópias de trabalho iniciais de cada equipa, é efetuado no servidor *web* um **fork** (botão específico na interface *web* do servidor oficial do projeto), ficando cada um com uma cópia independente no seu repositório a qual podem clonar para efetuar as alterações. Para efetuar a posterior integração das alterações, o integrador junta (**merge**) as alterações das várias equipas para o seu `master` local, antes de o enviar (**push**) para o `master` de distribuição oficial do projeto.

5.5 Configuração avançada

Para realizar fluxos de trabalho mais complexos pode ser útil realizar operações mais especializadas.

5.5.1 Multiple remotes

O **git** permite que o mesmo projeto possa estar replicado em dois ou mais servidores. Assim, o repositório local é configurado para que uma operação de envio para o servidor possa ser efetuada para um ou mais servidores. O programador pode decidir só enviar para um deles ou para ambos (ou mais).

Usando o comand `git remote add` adicionam-se dois servidores, indicando o nome simbólico escolhido e o URL do repositório remoto, por exemplo:

```
$ git remote add um http://github.com/user/proj.git
$ git remote add outro http://user@bitbucket.com/user/proj.git
```

A partir deste momento é possível enviar o código para um dos servidores ou para o outro, separadamente. Para enviar para ambos os servidores com a mesma instrução é necessário alterar a configuração do projeto. O comando `git config -e` permite editar o ficheiro `.git/config` que contém as definições locais do projeto. Neste ficheiro adiciona-se, por exemplo no fim, uma nova entrada com ambos os URL:

```
[remote "both"]
  url = http://github.com/user/proj.git
  url = http://user@bitbucket.com/user/proj.git
```

Agora, o comando `git push both` permite enviar de uma só vez o projeto para ambos os servidores.

Notar que a obtenção do código alterado nos servidores tem sempre de ser efetuada em separado, pois não é possível resolver os possíveis conflitos entre os três repositórios, o local e os dois remotos, simultaneamente. Assim, o processo terá de ser realizado em dois passos, primeiro com um dos servidores e depois com o outro.

5.5.2 Configuração de um servidor

Nesta seção não vamos tratar de instalar um serviço tipo **github** num computador pessoal. Embora exista uma distribuição de **gitlab** que pode ser instalada localmente, com o mesmo aspeto do servidor **gitlab.com** mas onde a gestão e o disco são locais.

Na realidade um computador com serviço de **ssh** (**sshd**) pode funcionar como servidor. Neste caso, vamos assumir uma distribuição *linux* tipo *ubuntu*. Para que outros utilizadores possam aceder à conta do utilizador (*user*) da máquina (*exemplo.pt*), que vai conter o repositório é, necessário adicionar as chaves SSH públicas dos utilizadores (ver secção 5.1) ao ficheiro `~/.ssh/authorized_keys` (a diretoria `~/.ssh` deve ser criada se não existir). Notar que quem tiver a chave privada da chave pública adicionada tem acesso à máquina por **ssh** (ver abaixo).

Para criar o repositório no servidor utilizar a opção `-bare`:

```
$ mkdir ~/proj.git
$ cd ~/proj.git
$ git init --bare
```

Os utilizadores remotos devem criar o projeto com pelo menos um *commit*, ou utilizar um existente, e aceder ao servidor a partir da sua máquina

```
$ git remote add origin user@exemplo.pt:proj.git
$ git push -u origin -all
```

Os restantes utilizadores podem obter cópias do projeto pela forma habitual

```
$ git clone user@exemplo.pt:proj.git
```

Sugere-se criar um utilizador dedicado (`sudo adduser git`) para gerir os repositórios. Para evitar que os detentores da chave privada tenham acesso à máquina, é necessário que o *login* apenas possa ser realizado com comandos **git** e não com um interpretador de comandos genérico (*shell*). O comando `sudo chsh git` indicando a localização do comando **git-shell**, em geral `/usr/bin/git-shell`, altera o *shell* do utilizador (adicionar `/usr/bin/git-shell` a `/etc/shells`, se não existir).

6 Gestão do projetos ágil: Scrum

O Scrum é uma metodologia ágil para gestão e planeamento de projetos de software. Em Scrum, os projetos são divididos em ciclos (tipicamente mensais) chamados de Sprints. O Sprint representa um conjunto de atividades que deve ser executado. As funcionalidades a serem realizadas num projeto são mantidas numa lista que é conhecida como Product Backlog. No início de cada Sprint, faz-se um Sprint Planning Meeting, ou seja, uma reunião de planeamento na qual o Product Owner prioriza os itens do Product Backlog e a equipa seleciona as atividades que ela será capaz de realizar durante o Sprint. As tarefas associadas ao Sprint são transferidas do Product Backlog para o Sprint Backlog. A cada dia de uma Sprint, a equipa faz uma breve reunião (normalmente de manhã), chamada Daily Scrum, com o objetivo informar sobre a evolução das tarefas, identificar impedimentos e definir prioridades. Ao final de um Sprint, a equipa apresenta as funcionalidades realizadas numa Sprint Review Meeting. Finalmente, faz-se uma Sprint Retrospective e a equipa faz o planeamento do próximo Sprint.


Daily scrum Em cada dia do Sprint a equipa faz uma reunião diária, chamada Daily Scrum. Esta tem como objetivo relatar o que foi feito no dia anterior, identificar impedimentos e priorizar o trabalho a ser realizado no dia que se inicia. Os Daily Scrums normalmente são realizadas no mesmo lugar, na mesma hora do dia. Todos os membros da equipa devem participar do Daily Scrum. O Daily Scrum não deve ser usado como uma reunião para resolução de problemas. Questões levantadas devem ser levadas para fora da reunião. Durante o Daily Scrum, cada membro da equipa indentifica o que fez ontem, o que fará hoje e possíveis impedimentos.


Sprint backlog O Product Backlog é uma lista contendo todas as funcionalidades desejadas para um produto, neste caso o enunciado do projeto. O Sprint Backlog é uma lista de histórias que o Scrum Team se compromete a fazer num Sprint. As histórias do Sprint Backlog são extraídas do Product Backlog, pela equipa, com base nas prioridades definidas pelo Product Owner. A equipa identifica as histórias a realizar durante a Sprint.

A equipa quebra cada história do Product Backlog numa ou mais tarefas do Sprint Backlog. Isso ajuda a dividir o trabalho entre os membros da equipa. Podem fazer parte do Product Backlog tarefas técnicas ou atividades diretamente relacionadas às funcionalidades solicitadas.


Sprint retrospective e review meeting O Sprint Retrospective ocorre no final de um Sprint e serve para identificar o que funcionou bem, o que pode ser melhorado e que ações serão tomadas para melhorar. No final de cada Sprint é feito um Sprint Review Meeting. Durante esta reunião, a Scrum Team mostra o que foi alcançado durante o Sprint. Tipicamente, isso tem o formato de uma demo das novas funcionalidades. Durante o Sprint Review, o projeto é avaliado em relação aos objetivos do Sprint, determinados durante o Sprint Planning Meeting.

6.1 Sprint: github wiki


A gestão do projeto será realizada com recurso à *wiki*  do projeto no **github.com**. Para tal deverá ser criada uma página para cada Sprint, designada por `sprint-X` (onde X representa o número do Sprint). Esta página deve incluir a identificação dos elementos da equipa, manter uma versão atual do diagrama de classes **uml** e a lista das histórias do respetivo Sprint (backlog). A página também incluirá a evolução da realização das tarefas (daily scrum) pelos membros da equipa.

A identificação de cada elemento da equipa deve incluir um **link**  para a conta de github do membro, sendo o texto do link constituído pelo número mecanográfico e respetivo nome.

O diagrama **uml**, preferencialmente no formato **.jpeg**, pode ser gerado de forma automática a partir do ficheiro de especificação **.dml** com recurso à ferramenta **dml2yuml** e ao site **http://yuml.me**.

Cada história da lista deve ser identificada com uma frase que a descreva e conter um **link**  para o respetivo *issue*, ou problema a resolver.

6.2 Tasks: github issues

Cada história ou tarefa identificada no Product Backlog deve ser representada como um *issue*  do projeto no **github.com**.

Para caracterizar as histórias e as tarefas, devem ser criadas etiquetas (*labels*) que permitam distinguir os diferentes tipos de atividades a realizar:

story deve ser utilizado para classificar *issues* que representam uma história. Deve ser aplicado quando o *issue* descreve uma funcionalidade a desenvolver do ponto de vista do utilizador. Cada história será decomposta numa ou mais tarefas de concretização.

feature deve ser utilizado para classificar *issues* que descrevem uma tarefa de concretização em que é necessário alterar código já realizado anteriormente.

new feature deve ser utilizado para classificar *issues* que descrevem uma tarefa de concretização onde o programador vai adicionar novas funcionalidades ao código já desenvolvido.


test deverá ser utilizado para classificar *issues* que correspondem a tarefas de concretização de casos de teste.


bug deverá ser utilizado para classificar *issues* que correspondem a tarefas de resolução de erros do projecto.

Para identificar os pontos de cada história devem ser criadas as etiquetas **1 s**, **2 s**, **3 s**, **5 s**, **8 s**, **13 s** e **21 s**. Estas etiquetas vão representar a taxa de esforço da história, ou seja, a complexidade da sua realização. A escala não é linear pois quanto mais complexa e longa for a história, mais imprecisa se torna a sua estimação.

Deve ainda criar as etiquetas **1/2 h**, **1 h**, **2 h**, **3 h** e **4 h** que irão ser utilizadas para associar a estimativa do tempo de realização da tarefa ao **issue** que representa a tarefa a realizar. Notar que a estimativa de tempo aplica-se a uma tarefa, não a uma história, e deve ser determinada pelo membro da equipa a quem foi atribuída a tarefa. Assim, uma mesma tarefa pode ter estimativas de tempo significativamente distintas dependendo da experiência do membro da equipa. Da mesma forma, numa história com uma só tarefa, pode não existir uma dependência direta entre a complexidade da história e a estimativa de tempo da tarefa, em função das características do membro da equipa.

Com o objectivo de acompanhar o desenvolvimento do Sprint, em termos de *issues* concluídos, deve criar um **milestone** em cada Sprint. Seguidamente, cada um dos *issues* do Sprint deve ser associado ao respetivo **milestone**.

Histórias Para cada história, ou funcionalidade do sistema do ponto de vista do utilizador, identificada no Product Backlog deve ser criado um *issue*. A história tem um nome, uma breve descrição do problema, uma etiqueta com o tipo **story**, um ponto de história e um *milestone*. Cada história é dividida em tarefas, identificadas na descrição da história, e contendo um **link**  para o respetivo *issue* da tarefa, identificado como **#tk**, onde **tk** corresponde ao número do *issue* da tarefa.

Tarefas Cada tarefa, a realizar por um único membro da equipa, tem um nome, uma breve descrição do problema, uma etiqueta com o tipo e um *milestone*. Uma vez atribuída a um membro da equipa (*assign*) a tarefa deve incluir uma etiqueta com o tempo previsto. Cada tarefa deve incluir na sua descrição um **link**  para o *issue* da história a que pertence: **Task of #st**. Onde **st** corresponde ao número do *issue* da história a que pertence.

Realização Durante a realização da tarefa em questão, deve ser mantida uma contabilização do tempo efectivamente gasto. Este tempo efetivamente gasto deve ser reportado na janela de comentários do *issue*. Quando a tarefa se encontra terminada, o **commit** que a conclui deve ter o texto **closes #id**, ou seja **commit -m "closes #id"** (genericamente o *issue* pode ser precedido por um dos nove termos `close{, s, d}`, `fix{, es, ed}` ou `resolve{, s, d}`). Quando o **commit** for carregado para o **github.com** (**git push**), a tarefa será automaticamente fechada. Notar que o utilizador **github.com** que fecha a tarefa é aquele que realiza o **commit** e não aquele que efectua o **push**. Em alternativa, embora desaconselhado, uma tarefa pode ser fechada através da interface *web* do **github.com**, devendo ser incluído na janela de comentário do *issue* a mensagem `Implemented by commit` seguido do respetivo **hash** (7 dígitos hexadecimais do commit).

6.3 Daily Scrum: aula de laboratório

Neste caso do projeto o Daily Scrum terá uma periodicidade semanal a realizar durante a aula de laboratório do grupo. Todos os membros do grupo têm de participar no Daily

Scrum. Os trabalhadores estudantes, podendo não estar presentes, têm de fornecer a mesma informação dos restantes, incluindo as tarefas que se propõem fazer na semana seguinte.

Para cada semana (de aula de laboratório a aula de laboratório), cada grupo realiza o planeamento do trabalho a realizar durante a semana. Este planeamento será representado por uma matriz com dez (10) colunas e um número de linhas igual ao número de elementos do grupo. A primeira coluna indica o número mecanográfico do membro do grupo. A primeira actividade do planeamento semanal do grupo é indicar, nas colunas 2 a 8, o número de horas disponíveis de cada elemento do grupo para trabalhar no projecto nos vários dias da semana. Cada elemento do grupo deve indicar na coluna 9 da matriz as tarefas que estima ir realizar durante a semana, em função da disponibilidade indicada. À medida se vão realizando as tarefas atribuídas, cada elemento do grupo insere as tarefas que já terminou na última coluna da matriz. É importante actualizar esta matriz à medida que as tarefas vão sendo realizadas por forma a possibilitar que tarefas atrasadas possam ser atribuídas a outros membros da equipa.

Semanalmente, antes da aula de laboratório, cada elemento do grupo deve realizar a retrospectiva do trabalho efetivamente realizado na semana que termina. Se não existiram desvios face ao planeado, quer nas tarefas quer em tempo gasto, deve ser indicado: *de acordo com o previsto*.

Abaixo apresenta-se um formato possível para cada Sprint, onde os valores indicados entre [] representam links:

Membros: [00000 Maria], [99999 João], ...

UML: [figura]

Backlog:

História-1

Tarefa-1 [#1]

Tarefa-2 [#2]

...

História-2

Tarefa-1 [#5]

...

Planeamento: Semana XX-xxx a YY-yyy

Número	Seg	Ter	Qua	Qui	Sex	Sab	Dom	A-realizar	Realizadas
00000	0	3	0	3	0	0	2	[5] [9] [12]	[5] [9]
99999	2	0	4	2	0	0	0	[1] [2] [7]	[1] [2] [7]

.....

Retrospectiva: Semana XX-xxx a YY-yyy

00000: devido a ***** não foi possível realizar a tarefa 12.

99999:

Planeamento: Semana YY-yyy a ZZ-zzz

.....

Retrospectiva: Semana YY-yyy a ZZ-zzz

.....

.....

7 Aplicação por camadas: serviços e apresentação

Uma arquitetura de software por camadas (*multilayered software architecture*) é uma arquitetura que atribui diferentes responsabilidades a um conjunto de camadas que formam o produto. Esta separação de responsabilidades ou interesses permite desenvolver aplicações maiores, mais estruturadas e mais fáceis de manter. A associação de equipas distintas a cada camada permite a especialização e paralelização das tarefas. Para garantir a separação, cada camada só fala com a camada abaixo e não expõe à camada acima pormenores de implementação da camada abaixo.

Numa divisão lógica de um desenho orientado por objetos é comum separar os interesses nas camadas de:

dados (data access): persistência e infraestrutura;

domínio (business): restrições e lógica de negócio;

serviços (application): funcionalidade disponibilizada;

apresentação (user interface): apresentação da funcionalidade.

A camada de dados é responsável pelo armazenamento, local ou remoto, e pelo respetivo tratamento de dependências de formatos e sistemas operativos: representação endian (*little* ou *big*), carácter de mudança de linha (*cr* ou *lf*), dimensão dos inteiros (32 ou 64 *bits*), codificação dos caracteres (*latin-1* ou *utf-8*).

A camada de domínio, ou lógica de negócio, é responsável por impor as restrições ao modelo de dados, por exemplo não permitir nomes duplicados. Ao colocar as restrições no domínio evita-se a duplicação do código em camadas superiores, por exemplo liberta as operações adicionar, renomear e copiar da verificação do nome duplicado.

A camada de serviços isola as operações que se pretende que o sistema realize da forma como as operações são efetivamente levadas a cabo nas camadas de domínio dos dados. Além disso, permite que o domínio evolua sem que os serviços existentes tenham de ser modificados. Os serviços são na realidade as operações que o utilizador final está autorizado, ou contratualizado, a realizar com a aplicação, por oposição às operações internas que são necessárias para concluir com sucesso esses contratos.

A camada de apresentação oferece ao utilizador uma, ou mais, formas de interagir com a aplicação. Esta camada é responsável por tratar das especificidades de computação gráfica e interface com o utilizador que se oferece para interagir com a aplicação. A interação pode ser por terminal (texto), gráfica (janelas), navegador (páginas *web*). A camada de apresentação comunica apenas com a camada de serviços utilizando os contratos que foram definidos para a aplicação.

7.1 Camadas de dados e domínio

A fénix framework (ver seção 4) é responsável, conjuntamente com a base dados que lhe dá suporte (*mysql*), pela camada de dados e parte da camada de domínio. O modelo da

aplicação a desenvolver é definido na linguagem **dml** (ver seção 4.1) que impõe algumas restrições ao modelo de dados nas classes base (`X_Base.java`), nomeadamente no tipo dos atributos, nas relações permitidas e suas multiplicidades. É responsabilidade do programador, com recurso à linguagem Java, de adicionar as restantes restrições às classes sob o seu controlo (`X.java`).

Em Java, os métodos privados (`private`) são apenas acessíveis na classe, os métodos protegidos (`protected`) são acessíveis apenas na hierarquia da classe, a ausência de proteção implica que os métodos são acessíveis às classes da própria diretoria e os métodos públicos (`public`) são acessíveis por todos. Se o domínio incluir mais de uma diretoria, mesmo para exceções, os métodos necessitam de ser públicos. Certos pacotes ou ferramentas, como a **fénix framework**, que geram acessos às associações como métodos públicos, não podem estar dependentes da visibilidade imposta nas classes. Para restringir as funcionalidades públicas da aplicação apenas às funcionalidades requeridas (contratualizadas) pelo cliente, é criada uma camada de serviços.

7.2 Camada de serviços

Os serviços ligam o mundo exterior com o domínio. A camada de serviços define uma fronteira que disponibiliza um conjunto de operações e que coordena a resposta da aplicação a cada uma dessas operações.

Um serviço tem por objetivo realizar uma determinada funcionalidade contratualizada em nome do utilizador final. A execução de um serviço decorre em três fases: preparação, execução e recolha de resultados. Num modelo transacional, e sendo o serviço a entidade contratualizada, cada serviço funciona como uma unidade de processamento, ou seja realiza uma transação. No entanto, o serviço em si não é persistente embora atue sobre o domínio e este sobre a camada de dados.

Neste modelo, a preparação consiste na recolha dos argumentos necessários à execução do serviço. Esta preparação pode ser tão simples como a invocação do construtor da classe com os argumentos requeridos, como a posterior invocação de outros métodos, nomeadamente para a construção de listas de argumentos em número variável.

A execução, num modelo transacional, pode ser dividida na transação propriamente dita (`execute()`) e nas operações que a transação realiza (`dispatch()`). Permite-se, desta forma, que um serviço possa ser realizado pela composição de operações de outros serviços já existentes. O serviço em si deve ser tão simples quanto possível, pelo que todas as restrições e lógica devem residir no domínio.

A execução do serviço utiliza os argumentos recolhidos e armazena os dados resultantes da execução do serviço. Estes dados devem corresponder apenas à informação contratualizada e não à informação disponível no domínio. Desta forma, restringe-se a quantidade de informação a transferir entre o domínio e a apresentação. A apresentação não pode ter conhecimento das estruturas internas do domínio, pois tal comprometeria a evolução do domínio e a separação de interesses. Além disso, no contexto transacional, os objetos do domínio só existem no contexto da transação, podendo os dados ter de ser transferidos para outro processo ou computador para apresentação ao utilizador.

Assim, sempre que seja necessário transferir estruturas complexas, por oposição a estruturas e tipos de dados predefinidos na linguagem (por exemplo, `Integer`, `String` ou `List<Double>`), têm de ser definidos objetos específicos para a transferência dos dados (*data transfer objects* ou DTO). Estas estruturas, sem comportamento, são utilizadas especificamente para transferência de informação da camada de serviços para a camada de apresentação, e *vice-versa*.

Um serviço não pode encobrir ao utilizador final possíveis situações de erro que decorram das operações que este ordenou, sob pena de enganar o utilizador sobre o sucesso ou insucesso do seu pedido. Assim, um serviço não deve apanhar exceções, embora exceções do domínio possam ser convertidas noutras menos dependentes de pormenores específicos do domínio.

Os serviços de uma aplicação (App) podem derivar de uma classe que uniformiza a interface com o domínio:

```
package pt.tecnico.App.service;
import pt.ist.fenixframework.Atomic;
import pt.tecnico.App.core.*;
import pt.tecnico.App.exception.*;

public abstract class AbstractService {
    @Atomic
    public final void execute() throws AppException {
        dispatch(); // template method
    }
    protected abstract void dispatch() throws AppException;
    // add common code here
}
```

Notar que a anotação `@Atomic` não funciona em métodos abstratos, pelo que ou a anotação é colocada em todas as concretizações ou a operação é diferida para o método `dispatch`, com as vantagens atrás referidas.

A concretização de um serviço (Clean) consiste na recolha dos parâmetros (`param`) necessários à execução do serviço (`dispatch()`) e na recolha dos resultados da operação (`data`) para posterior passagem (`result()`) à camada de apresentação:

```
package pt.tecnico.app.service;
import pt.tecnico.app.domain.*;
import pt.tecnico.app.exception.*;
import pt.tecnico.app.dto.*;

public class CleanService extends AbstractService {
    private final int param;
    private CleanDto data;
    public CleanService(int param) { this.param = param; }
    @Override
    protected void dispatch() throws AppException {
        /* ... */
    }
    public CleanDto result() { return data; }
}
```

7.3 Camada de apresentação

A camada de apresentação trata dos problemas específicos da interface com utilizador e dependências do modelo de computação gráfica utilizado. Na realidade uma aplicação pode oferecer diversas interfaces e, hoje em dia, muitas aplicações oferecem mais de uma apresentação. A interação pode ser por terminal (texto), gráfica (janelas), navegador (páginas *web*). Por exemplo, o **github** (ver seção 5), ou o **gitlab**, permitem efetuar grande parte das operações a partir da linha de comandos, quer por **ssh** quer por comandos JSON. Algumas operações específicas só podem ser realizadas por algumas destas vias.

As apresentações gráficas são bastante trabalhosas, em geral, embora alguns pacotes de software procurem simplificar as operações mais repetitivas e comuns com recurso a *toolkits* e linguagens específicas.

A apresentação por interpretador de linha de comandos (*Shell*) é simultaneamente simples e flexível. O interpretador (*Shell*) agrega comandos (*Command*), constrói argumentos de cada comando e trata erros (exceções). Cada concretização do interpretador (*AppShell*) apenas tem de adicionar os comandos suportados e tratar da informação partilhada entre os diversos comandos:

```
package pt.tecnico.myDrive.presentation;

public class AppShell extends Shell {
    // data common to the commands
    public static void main(String[] args) throws Exception {
        new AppShell().execute();
    }
    public AppShell() { // add commands here
        super("AppName");
        new Op(this);
        // add other commands here
    }
    // shell common methods
```

Cada comando interage com um serviço ou mais serviços, definindo o nome pelo qual o comando é invocado (*op*) e uma mensagem explicativa da funcionalidade e sintaxe, realizando a operação pretendida (*execute*) a partir de um vetor de cadeias de caracteres:

```
package pt.tecnico.App.presentation;
import pt.tecnico.App.service.*;

public class Op extends Command {
    public Op(Shell sh) { super(sh, "op", "op help message"); }
    public void execute(String[] args) {
        /* ... */
    }
}
```

Também a camada de apresentação deve ser testada, como sequências de invocações a comandos, mas estes testes desconhecem a estrutura da aplicação, ou seja, tratam a aplicação como uma entidade opaca (*block-box*).

8 Testes de software: junit e jmockit

Os testes permitem garantir, e não apenas assumir, que o código efetivamente faz o que se pretende. Notar que os testes, em geral, são mais trabalhosos que o código que testam. No entanto, em especial em projetos de maior dimensão, ao acrescentar uma nova funcionalidade pode-se impedir de funcionar funcionalidades antigas já muito testadas. Assim, é extremamente importante que os testes sejam desenvolvidos em conjunto com as funcionalidades que testam, e que estes testes sejam executados sempre que o projeto é compilado, como forma de garantir que a respetiva funcionalidade ainda se encontra operacional. Código não testado não é de confiança e não pode ser utilizado em produção.

A execução manual dos testes deve ser evitada pois está muito sujeita a esquecimentos, a falhas na sua execução e a excesso de confiança no código desenvolvido, ou seja, conduz à omissão deliberada da execução dos testes. Assim, os testes devem ser realizados de uma forma automática e, de preferência integrados no processo de desenvolvimento.

A abordagem que pode parecer mais natural consiste em desenvolver a funcionalidade pretendida e depois testar se esta está efetivamente a funcionar como esperado. Claro que se o teste falhar, o erro pode estar na funcionalidade que foi mal codificada ou no teste que está a tirar as conclusões erradas. Acontece que ao desenvolver o teste para uma funcionalidade já existente há a tendência para adaptar o teste às fragilidades ou particularidades da funcionalidade. Uma solução consiste em atribuir a equipas distintas a construção da funcionalidade e do teste. Outra solução consiste em desenvolver primeiro o teste e só depois a funcionalidade a ser testada. Nesta abordagem de desenho orientado pelos testes (*Test driven design* ou *Test first*), quando um teste falha primeiro olha-se para a funcionalidade, pois foi a última a ser desenvolvida, e só depois para o teste. Desta forma é mais provável que se desenvolva aquilo que se pretendia originalmente.

8.1 Testes de unidade

Os testes unitário têm por objetivo verificar as funcionalidades de cada classe por sua vez. Só depois de cada classe estar a funcionar conforme esperado é que se pode ter esperança que duas ou mais classes funcionem em conjunto.

Cada classe a testar vai necessitar de diversos testes que cubram as situações de sucesso e de erro mais comuns. Idealmente todos os valores possíveis deveriam ser testados, mas o tempo necessário seria proibitivo. Assim, torna-se necessário escolher sequências de teste que exercitem o maior número de situações e que executem o maior número de linhas de código possível. Testes mais abrangentes garantem também que cada condição é testada, pelo menos uma vez para o caso verdadeiro e uma vez para o caso falso, sejam condições simples (*ifs*) ou ciclos. Condições de fronteira, junto a limites impostos pelas regras de negócio, devem também ser testadas.

Um teste só testa alguma coisa se o resultado esperado for efetivamente verificado.

O modelo de execução dos testes pode ser resumido à preparação (*Arrange*) , execução (*Act*) e avaliação dos resultados (*Assert*). Assim, qualquer sequência de teste tem de terminar com uma verificação ou asserção (*assert*). Para mais, os testes devem ser independentes um dos outros e testar uma característica específica, caso contrário, caso o teste falhe, não se sabe o que deixou de funcionar. No contexto de um modelo transaccional, a possibilidade de retrocesso (*rollback*) permite que o teste deixe o domínio no estado em que o teste se iniciou.

8.1.1 junit

A ferramenta **junit** (junit.org) é uma plataforma para execução de testes ao software de uma forma repetitiva. Na sua versão atual (4.12) basta anotar o método de teste com `@Test`:

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;
public class Teste {
    @Test
    public void sqrt() {
        assertEquals(5, Math.sqrt(25), 1e-9);
    }
}
```

compilar o código

```
javac -cp .:junit-4.12.jar Teste.java
```

e executar o **junit** sobre a classe resultante

```
java -cp .:junit-4.12.jar:hamcrest-core-1.3.jar org.junit.runner.JUnitCore Teste
```

Quando integrada dentro do ciclo de desenvolvimento da ferramenta **maven** (ver seção 2), basta incluir a dependência no ficheiro `pom.xml` (ver seção 2.4), sendo os resultados colocados em `target/surefire-reports/` após a execução da fase de testes.

Em geral, as classes de teste declaram a mesma `package` para ter acesso à classe a testar sem problemas de permissões nem necessidade de `imports`. Para evitar povoar a diretoria do código a ser testado com as classes de teste, estas últimas são colocadas em diretorias distintas. Como a estrutura das diretorias corresponde à `package` declarada, a solução utilizada recorre a prefixar a `package`: `src/package/Class.java` e `test/package/Teste.java`.

8.1.2 asserções

No modelo de execução dos testes, a fase de avaliação dos resultados é aquela que permite concluir se a funcionalidade se comportou como esperado. Tal significa que não foram encontradas falhas num teste onde era esperado o sucesso, mas também garantir que a falha foi identificada num teste onde não se esperava sucesso.

O principal mecanismo para a avaliação de resultados do teste consiste na realização de uma ou mais asserções. Quando uma asserção falha, ou seja as condições esperadas

não se verificaram, o programa ou teste devem terminar com uma mensagem de erro explicativa, por forma a permitir a sua pronta correção. Já na linguagem **C** a rotina `assert` (`assert.h`) permitia efetuar este tipo de verificações, devendo o argumento da rotina ser verdadeiro. Na ferramenta **junit** a classe `Assert` realiza a mesma operação com recurso a diversos métodos estáticos. Estes métodos permitem, por exemplo, compara dois valores em vírgula flutuante até à precisão indicada, como no exemplo acima. Para a verificação da valores booleano e resultados de comparações, as rotinas `assertTrue` e `assertFalse` asseguram que o valor lógico resultante do teste corresponde ao esperado. Sem querer ser exaustivo, é de referir os métodos de comparação de dois valores quaisquer, `assertEquals` e `assertNotEquals`, e a verificação de referências para objetos, `assertNull` e `assertNotNull`. A rotina `fail` permite terminar um teste com erro, sempre que uma validação não verificada, sem ter de recorrer condições falsas nas rotinas acima referidas. Por último, cada uma das rotinas de asserção referidas permite indicar como primeiro argumento um texto (`String`) descritivo da asserção realizada. Este pormenor é particularmente útil, pois um teste pode ser validado com uma única asserção, mas frequentemente são necessárias várias asserções para efetivamente garantir que a operação decorreu de acordo com o esperado. Por exemplo, garantir que uma lista contém dois elementos e o primeiro é menor que o segundo.

A ferramenta **junit** permite quando é esperado que uma sequência de teste termine devido ao lançamento de uma exceção, o teste reconheça essa exceção como sucesso, e apenas essa. Desta forma, evitam-se blocos `try-catch` com `fail` e `assertTrue`. Para tal, a anotação `@Test` passa a ser sufixada com a exceção esperada, por exemplo `@Test(expected=IOException.class)`. Assim, todas as exceções derivadas de `IOException` conduzem a um teste com sucesso, enquanto qualquer outra exceção, ou a ausência de uma exceção conduz à falha do teste.

Outras anotações do teste incluem `@Test(timeout=100)` para limitar a execução do teste a 100 milissegundos, ou `@Ignore` (além do `@Test`) para que este teste específico não seja executado.

Para ajudar na preparação do teste, as rotinas anotadas com `@Before` e as rotinas estáticas anotadas com `@BeforeClass` são executadas antes de cada teste e antes de todos os testes de uma determinada classe, respetivamente. Por oposição, a tarefa de limpar o lixo deixado pelo teste, por forma a garantir que o sistema se encontra na mesma condições em que estava antes do início do teste, recai nas rotinas anotadas com `After` e `@AfterClass` para todos os testes da classe.

Embora saindo fora do âmbito deste documento, a ferramenta **junit** permite que os testes recebam parâmetros (`@Parameter`), permitindo testar diversas combinações de valores de entrada com um mesmo teste. O testes também podem ser agrupados (`@Category`) podendo ser executados em grupos distintos.

8.1.3 Teste de serviços

No caso de aplicações suportadas por sistemas baseados em transações, como a **fénix-framework** (ver seção 4), é possível utilizar as capacidades de retrocesso (*rollback*) para deixar o sistema no mesmo estado que se encontrava antes do início do teste. Para tal, pode-se criar uma classe abstrata que estabelece a infraestrutura do teste, e da qual todas as restantes classe de teste derivam. Esta classe pode ser utilizada para testar o serviços que a aplicação oferece, e até testes a partes específicas do domínio.

```
public class AbstractServiceTest {
    @Before
    public void setUp() throws Exception {
        try {
            FenixFramework.getTransactionManager().begin(false);
            populate();
        } catch (WriteOnReadError |
                NotSupportedException |
                SystemException e1) {
            e1.printStackTrace();
        }
    }
    @After
    public void tearDown() {
        try {
            FenixFramework.getTransactionManager().rollback();
        } catch (IllegalStateException |
                SecurityException |
                SystemException e) {
            e.printStackTrace();
        }
    }
    public void populate() {
        // transaction based setUp
    }
}
```

Esta classe, ou outras classes abstratas derivadas desta, devem também incluir atributos ou métodos comuns a dois ou mais testes. As classes concretas testam serviços específicos, explorando as diversas combinações de argumentos e os valores mais críticos.

```
public class OneServiceTest {
    public void populate() {
        // super.populate() // global setUp
        // service specific setUp
    }
    @Test
    public void one() {
        // test specific setUp
        // service.execute();
        // assert....
    }
}
```

```
    }  
    // more tests  
}
```

8.2 Testes de cobertura

Que percentagem do código foi efetivamente testada pelos testes desenvolvidos?

O objetivo da análise de cobertura é verificar quais são as partes do código que estão efetivamente a ser testadas pelos testes executados. Um programa está aceitavelmente testado se um número significativo das suas linhas de código foram executadas e se as condições foram testadas para os casos verdadeiro e falso.

As ferramentas de análise de cobertura dividem-se naquelas que adicionam código requerem a re-compilação do código fonte e aquelas que manipulam o código diretamente, antes ou durante a execução. Em Java existem diversas ferramentas, como o **JCov**, o **JaCoCo**, o **Clover**, o **Emma** ou o **cobertura**.

O **cobertura** é uma ferramenta de análise de cobertura para **java** que requer a re-compilação do código fonte de fácil utilização. Em **maven** o **cobertura** é invocado com `mvn cobertura:cobertura`, recompilando e re-executando todos os testes, produzindo os resultados em `target/site/cobertura/` sob a forma de páginas *web*.

8.3 Testes de integração e de sistema

Os teste de integração e de sistema recorrem a sequências de teste que envolvem a invocação de inúmeros serviços. Assim, não existe uma única transação mas várias transações envolvidas num só teste. Consequentemente, estes teste não podem derivar do `AbstractServiceTest` acima. Além disso, para deixarem o sistema no seu estado inicial, antes da realização do teste, é necessário desfazer, caso a caso, os efeitos secundários de cada teste.

Quer os testes de integração como os testes de sistema implicam a invocação sequencial de diversos serviços, permitindo verificar o comportamento dos serviços em conjunto. A principal diferença reside no facto de os testes de sistema serem testes opacos (*black-box*) e desconhecerem as particularidades de cada serviço. Nos testes de sistema, as operações correspondem a pedidos de alto nível, simulados de tal forma que pareçam ser realizados por um utilizador ao terminal, por exemplo *Monkey tests*.

8.4 Testes com mock-ups

Numa associação entre duas classes A e B, testar A implica testar B, e *vice versa*. Para garantir a independência do software a testar, seja ele A ou B, é necessário garantir que a outra classe, que não está a ser testada, se comporta como esperado. Só desta forma é possível isolar possíveis erros na classe em teste.

A utilização de *mock-ups* permite determinadas rotinas da outra classe devolvam valores predefinidos, aqueles que são estritamente necessários que a outra classe devolva

para testar uma funcionalidade específica da classe em teste. Assim, os *mock-ups* devem ser definidos teste a teste e instrumentados para devolver os valores relevantes para esse teste específico. Isto permite que a outra classe, ou outras classes, envolvidas no teste tenham erros ou estejam ainda total ou parcialmente incompletas. Tal permite ainda simular situações reais, como sobrecarga de rede, que podem ser difíceis de reproduzir em ambiente de teste.

Por exemplo, para testar o resultado da expressão $\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$ sem ter uma versão funcional da rotina *sqr*t(), recorre-se ao *mock-up* da rotina *sqr*t(), instrumentada para devolver apenas os valores a utilizar nos testes. Para testar $a = 2, b = 5, c = 2$, configura-se *sqr*t() para devolver, neste teste, o valor 3, pois $b^2 - 4ac = 9$ e $\sqrt{9} = 3$. Noutro teste, com $a = 4, b = 12, c = 9$, *sqr*t() devolve sempre 0.

```
class Resolve {
    double a, b, c;
    public Resolve(double a, double b, double c)
        { this.a = a; this.b = b; this.c = c; }
    public double resolve() { return (-b+sqrt(b*b-4*a*c))/(2*a); }
    public double other() { return (-b-sqrt(b*b-4*a*c))/(2*a); }
    public double sqrt(double x) { /* FIXME */ return 0.0; }
}
```

8.5 jmockit com MockUps

A ferramenta **jmockit** (jmockit.org) é uma plataforma para construção de *mock-ups* para teste, compatível com o **junit**. A construção de *mock-ups* pode ser realizada de diversas formas. O caso mais simples consiste em substituir os métodos por *mock-ups*. No caso acima, temos

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;
import org.junit.runner.RunWith;
import mockit.Mock;
import mockit.MockUp;
import mockit.integration.junit4.JMockit;
@RunWith(JMockit.class)
public class ResolveTest {
    @Test
    public void resolve() {
        new MockUp<Resolve>() {
            @Mock
            double sqrt(double x) {
                assertEquals("sqrt argument", x, 9.0, 1e-9);
                return 3.0; }
        };
        Resolve r = new Resolve(2, 5, 2);
        assertEquals("first root", -0.5, r.resolve(), 1e-9);
        assertEquals("second root", -2.0, r.other(), 1e-9);
    }
}
```

Neste caso, contrói-se um `MockUp` da classe `Resolve`, na qual se substitui (`@Mock`) o método `sqrt()` pela solução requerida para este teste. A classe de teste é anotada com `@RunWith(JMockit.class)`, indicando que contém *mock-ups*. Notar que o `assertEquals` dentro do método `sqrt()` garante que foi efetivamente invocado com o valor 9, enquanto que os outros `asserts` garantem que as raízes calculadas estão corretas.

Para executar o teste, deve compilar o código com o comando

```
javac -cp .:jmockit-1.21.jar:junit-4.12.jar Resolve.java ResolveTest.java
```

e executar o **junit** sobre a classe resultante

```
java -cp .:jmockit-1.21.jar:junit-4.12.jar:hamcrest-core-1.3.jar \
    -javaagent:jmockit-1.21.jar org.junit.runner.JUnitCore ResolveTest
```

Quando integrada dentro do ciclo de desenvolvimento da ferramenta **maven** (ver seção 2), basta incluir a dependência no ficheiro `pom.xml` (ver seção 2.4).

A utilização de *expectations*, uma outra forma de realizar *mock-ups*, permite ter uma maior controlo sob o teste. Neste caso, é possível fazer *mock-up* de algumas chamadas a um método *mocked*, mas não necessariamente a todas as invocações, como no caso anterior. Assim, o método *mocked* só é ativado se a instância, o número de argumentos ou o número de chamadas estiver de acordo com a especificação. Nos restantes casos o método original continua a ser invocado. As instâncias *mocked* deverão ser anotadas `@Mocked` se o comportamento desejado for igual para todas e `@Injectable` caso se pretenda que cada instância, de uma mesma classe, tenha um comportamento distinto.

A utilização de *verifications* permite verificar, *à posteriori*, se os invocações foram efetivamente realizadas, verificando os seus argumentos e número vezes que foram invocadas. Neste caso, o código deve existir e estar funcional, permitindo o *verifications* garantir que o teste executou o métodos verificados e com os argumentos esperados.