Docentes
João Paulo Barraca <jpbarraca@ua.pt>
André Zúquete <andre.zuquete@ua.pt>
Bernardo Cunha <mbc@ua.pt>

TEMA 6

Ferramentas de Automação

Objetivos:
- Automação em projetos de software
- GNU make, automake e autoconf

6.1 Automação em Projetos

Durante o desenvolvimento de um projeto de software, ou mesmo de um documento técnico em LaTeX é comum a realização de operações de forma repetitiva a cada nova versão. Um exemplo é a necessidade de compilar o código produzido de forma poder validar a sua execução, gerar documentação, ou publicar novas versões para um servidor. No caso de se utilizar LaTeX, e caso se utilizem bibliografias automáticas, ou acrónimos, é necessária a execução de várias ferramentas, de forma sequencial.

Considere o caso comum no desenvolvimento de uma aplicação simples em *Java*. De uma forma geral, e para evitar problemas, é necessário apagar todas as classes locais

```
rm -f *.class
```

De seguida é necessária a compilação de todos os ficheiros de código fonte:

```
javac *.java
```

Se se pretender gerar documentação no formato *JavaDoc* será necessário:

- Criar um diretório para armazenar a documentação: `mkdir doc`
- Caso existe o directório, o seu conteúdo deve ser apagado: `rm -rf doc/*`
- Gerar a documentação: `javadoc -d doc classname`

Voltar a gerar a documentação exige a repetição do processo.

No caso de se utilizar LaTeX com bibliografias e acrónimos, torna-se necessário executar uma sequência tal como:

- Compilar o programa e criar a lista de índices, referências e acrónimos: `pdlatex doc.tex`
- Repetir o passo pois podem ter sido introduzidas referências novas: `pdflatex doc.tex`
- Colocar as referências no texto, e a lista no final: `bibtex doc`
- Compor o documento final com acrónimos e referências: `pdflatex doc.tex`

Estes passos têm de ser repetidos sempre que a lista de referências ou acrónimos é alterada.

Outro problema frequente refere-se ao processo de distribuição de código fonte não compilado. Em grande parte das linguagens atualmente utilizadas, podem existir dependências de bibliotecas, módulos ou classes externas às tradicionalmente incluídas na linguagem. Ora, como garantir que num dado sistema que se pretende compilar um dado programa, existem todas as dependências necessárias? Mesmo quando a aplicação é distribuída de forma compilada esta questão ainda se coloca, como já pode verificar ao utilizar o comando `apt-get`.

De forma a agilizar o processo de desenvolvimento, distribuição e adaptação a sistemas heterogéneos, existem ferramentas que auxiliam o programador ou escritor técnico, automatizando processos, ou permitindo a detecção dos módulos existentes e a adaptação dinâmica.

Neste guião irão ser abordadas 2 funcionalidades principais:

- Automatização de tarefas
- Detecção do ambiente para instalação

6.2 GNU make

A ferramenta *make***gnu-make** permite determinar que partes de um software necessitam de ser compiladas. É particularmente útil quando o tamanho de um dado programa é considerado grande pois possibilita se detete que partes necessitam de compilação, reutilizando ao máximo o trabalho que já se teve ao compilar outras partes. Considere que alguns programas podem demorar largos minutos ou mesmo horas a compilar o código fonte.

Embora o foco do *make* seja a compilação de código fonte, como no desenvolvimento de programas, ou mesmo de documentos LaTeX, de uma forma mais genérica, pode-se considerar que o *make* permite especificar regras e dependências entre estas regras.

Na base da ferramenta *make* estão ficheiros denominados de *makefiles*, e com tipicamente chamada de `Makefile`. Este ficheiro define os relacionamentos entre regras e ficheiros em disco, fornecendo comandos para actualizar ou gerar cada parte. Num programa considera-se que as partes são classes ou objectos binários, sendo que estes são obtidos através da compilação de ficheiros com códifo fonte.

Utilizar o *make* é tão simples como simplesmente executar `make`. Sem a existência de um ficheiro `Makefile`, o *make* irá apresentar um erro. No entanto, caso exista, este ficheiro é automaticamente processado.

Exercício 6.1

Crie um directório denominado por guiao-6 e mude para ele.

Execute o comando `make` e verifique o seu resultado.

Crie um ficheiro vazio chamado `Makefile` e volte a executar `make`. Compare o resultado com a execução anterior.

Caso o ficheiro `Makefile` tenha um nome diferente de `Makefile` o *make* pode ser executado na forma `make -f nome-da-makefile`. Isto é útil caso existam ficheiros `Makefile` diferentes para diferentes conjunto de ações (ex. *Linux* vs *OS X*).

6.2.1 Formato dos ficheiros `Makefile`

O conteúdo dos ficheiros `Makefile` é texto que define como o *make* deve prosseguir. Segue uma sintaxe reduzida em que as linhas são processadas de cima para baixo. Existem

apenas 3 tipos de linhas:

- **Regras**: possuem o formato *nome:*. Ver a Subseção 6.2.2.
- **Definição de variáveis**: possuem o formato *nome-da-var* = *valor*.
- **Linha comentada**: Inicia-se com o caráter "#".

Estes ficheiros não são compilados, sendo interpretados directamente pelo *make*. De forma a estruturar melhor o conteúdo, é possível a um `Makefile` incluir outra através da diretiva `include`, como demonstrado a seguir:

```
include subMakefile
```

6.2.2 Regras

A base de um ficheiro `Makefile` é uma lista de regras com *alvos*, *dependências* e uma receita. Pode-se considerar que o formato típico de uma regra é o seguinte:

```
alvos:  dependencia1 dependencia2
          receita
```

O *alvo* é normalmente o nome do ficheiro que se pretende gerar (ex. `documento.pdf`), mas também pode ser uma ação. Considerando a compilação de um programa *Java*, poderia-se criar um `Makefile` apenas com a seguinte regra:

```
prog1-ex12:
           @echo "Compilar ficheiro"
           javac Prog1-EX12.java
```

Sendo que o comando `javac Prog1-EX12.java` poderia ser invocado executando:

```
make prog1-ex12
```

Escrever receitas de compilação é simples, no entanto têm de ser respeitadas regras e boas práticas.

- A receita deve ser indentada e tipicamente é alinhada com o separador `:`.

- Uma receita pode ser apenas uma linha em branco, o que neste caso corresponde a uma receita vazia que nada faz.

- Se uma linha for demasiado longa, esta pode ser dividida utilizando o caráter "\".

- Podem ser utilizadas variáveis nas receitas através da sintaxe `$(VARNAME)`

- Se uma linha da receita iniciar com o caráter `@`, esta linha não é apresentada ao utilizado. Por exemplo `@echo LABI` irá apresentar a mensagem `LABI`, mas irá omitir a apresentação da instrução.

- As receitas são executadas de forma linear. Se a execução de uma linha falhar (ex. erro de sintaxe), a execução de toda a receita é abortada. Pode-se declarar que uma linha é opcional se ela for iniciada com "-".

- É possível executar receitas de que invocam outras receitas, mesmo que estas estejam noutros directórios.

As linhas seguintes exemplificam alguns dos aspetos descritos.

```
regra-vazia:

prog1-ex12:
	@echo \
	    "Compilar ficheiro"
	-rm -f *.class
	javac Prog1-EX12.java
```

Existem alguns alvos que são universalmente utilizadas, ou tão comuns que são de um standard de facto. A saber:

- ***all***: Executa um processo completo. Por exemplo, todos os passos intermédios para construção da bibliografia e acrónimos, mais a geração do documento final. Outro exemplo seria a compilação de todos os ficheiros da aplicação e a geração do ficheir executável final.

- ***clean***: Limpa todos os ficheiros temporários ou auxiliares, deixando apenas ficheiros de código fonte.

Exercício 6.2

Dentro do directório `guiao-6`, crie um sub-directório chamado `doc-latex`.

Neste directório coloque um ficheiro LaTeX muito simples chamado `doc.tex`.

Crie uma `Makefile` com alvos para compilar e limpar os ficheiros temporários.

Verifique que consegue invocar os alvos e que executam como esperado.

Como referido, os alvos podem possuir dependências de forma a criar sequências de comandos. As linhas seguintes exemplificam como se poderia adicionar uma regra que limpava todos os ficheiros `.class` antes da compilação:

```
clean:
	rm -f *.class

prog1-ex12: clean_
	@echo "Compilar ficheiro"
	javac Prog1-EX12.java
```

Exercício 6.3

Altere a `Makefile` que criou anteriormente de forma a utilizar o mecanismo de dependências. Neste caso, a compilação principal terá como dependência um alvo que cria os índices (`pdflatex nome.tex`) e outro que compila a bibliografia (usando `bibtex`). Insira uma pequena bibliografia para testar.

Crie uma dependência na compilação para a limpeza dos ficheiros auxiliares e verifique que funciona.

Exercício 6.4

Crie uma regra chamada `publish` que envie (utilizando `scp`) o documento produzido para a sua área no servidor *xcoa.ua.pt*.

Tenha em atenção que esta regra só deverá ser executada caso a compilação tenha sido previamente executada.

Uma funcionalidade importante do *make* é a deteção de modificação de ficheiros, apenas invocando a compilação caso o ficheiro com o código fonte tenha sido alterado.

No caso de se utilizar LaTeX, o ficheiro final apenas necessita de ser produzido caso algum dos ficheiros fontes contenha alterações. Isto é feito utilizando o nome dos ficheiros como alvos ou dependências nas regras.

```
clean:
      rm -f *.class

Prog1-EX12.class: Prox1-EX12.java
          @echo "Compilar ficheiro"
          javac Prog1-EX12.java

all: Prog1-EX12.class
```

No caso exemplificado, se o ficheiro `Prog1-EX12.java` tiver sido modificado depois do ficheiro `Prog1-EX12.class`, o que se determina pela sua hora de modificação, ele é compilado novamento. Caso contrário, deveria ser apresentada uma mensagem semelhante

a: `make: 'Prog1-EX12.class' is up to date.`

Exercício 6.5

Sabendo que o ficheiro `.bib` é convertido para um ficheiro `.bbl` depois de um ficheiro `.aux` ser criado, que o documento final depende do ficheiro `.tex`, dos índices (ex. `.toc`), e bibliografia (`.bbl`), re-escreva as regras de forma a utilizarem o nome dos ficheiros e assim usufruir da detecção de alterações.

Verifique que necessita de alterar o ficheiro `.tex` para que a compilação seja repetida.

Uma forma cómoda de especificar estas dependências, especialmente quando existem muitos ficheiros a ser monitorizados, é construir uma lista de referências e utilizar variáveis. O exemplo que se segue define a variável `objects`, define como compilar qualquer ficheiro com extensão `.java` e quais as dependências do alvo `all`.

```
objects = A.class B.class C.class

%.class:
	javac $*.java

all: $(objects)
```

6.2.3 Sub-Directórios

Frequentemente os código fonte de um programa encontra-se dividido por módulos, ou pacotes, colocados em directórios diferentes. Ora, para compilar todo o programa é necessário compilar todos os seus módulos pela ordem correta. O sistema *make* tem suporte para este processo através de um mecanismo que permite invocar um `Makefile` a partir de outros ficheiros `Makefile`.

Exercício 6.6

No directório `guiao-6` crie um directório `src` e coloque algum código java que possua de outra disciplina.

Crie uma `Makefile` que permita compilar este código através do alvo *all*.

Não se esqueça de criar o alvo *clean*.

Dada a estrutura criada, deverá ter um directório `guião-6` com dois sub-directórios `doc` e `src`. Seria interessante que uma `Makefile` existindo no directório pai pudesse invocar a construção do código fonte e da documentação. As linhas seguintes demonstram uma base como tal pode ser feito. Neste caso, é necessário utilizar a palavra reservada `.PHONY` para indicar que os nomes não se devem referir a ficheiros no disco. Se assim fosse, nunca nada seria compilado pois os alvos (neste caso directórios) já existem.

```
SUBDIRS = doc src

.PHONY: subdirs $(SUBDIRS)

subdirs: $(SUBDIRS)

$(SUBDIRS):
	$(MAKE) -C $@
```

Neste caso foi utilizada a sequência `$@`. A isto chama-se uma acção, existindo várias acções possíveis:

- `$@` : nome completo do alvo.
- `$?` : dependências mais recentes que o alvo.
- `$*` : nome do alvo sem extensão.
- `$<` : nome da primeira dependência.
- `$^` : nome de todas as dependências (separadas por espaço).

Exercício 6.7

Adapte a estrutura demonstrada anteriormente de forma a que possa propagar a compilação e a limpeza (*clean*) para o código e para a documentação.

6.3 GNU Automake e Autoconf

O *automake* **gnu-automake** é uma ferramenta que, baseando-se no *make* possibilita a gestão da compilação para programas mais complexos, e combinado com a ferramenta *autoconf* **gnu-autoconf** facilita a distribuição dos programas na sua forma de código fonte. Em particular, possibilita que os ficheiros `Makefile` sejam construídos de forma dinâmica, adaptando-se às condições existentes (ex. localização de ficheiros ou programas)

no sistema alvo. É ainda útil para que durante esta adaptação ao sistema alvo, seja possível verificar a existência de dependências sem os quais o programa não poderia ser compilado. Por exemplo, para os programas desenvolvidos nas disciplinas iniciais da Universidade de Aveiro é necessário que exista o *Java* na sua versão 1.6. Para os relatórios de Laboratórios de Informática é necessário que exista o LaTeX com suporte para `pdflatex` e todos os packages declarados no documento.

De uma forma muito simples pode-se considerar que o sistema necessita de dois tipos de ficheiros: Templates para ficheiros `Makefile` com nome `Makefile.am` e uma configuração de configuração com nome `configure.ac`. O primeiro é utilizado para facilitar a escrita de ficheiros `Makefile` complexa e será abordado na Subseção 6.3.2, enquanto o segundo é utilizado para validar a integridade do sistema e existência de todas as dependências e será abordado na Subseção 6.3.1.

6.3.1 GNU Autoconf

O ficheiro `configure.ac` contém uma série de macros que indicam que validações executar e como agir face a esse resultado. Pode considerar que os macros irão efetuar validações no sistema (ex. existência de *Java*) ou determinar parâmetros operacionais (ex. localização do compilador `javac` e sua versão). Da perspectiva de um aluno isto é extremamente importante pois permite enviar código (p.ex. para o docente) junto com a validação de que o ambiente de avaliação tem os componentes necessário à correta execução.

No início deste ficheiro, terá de ser declarado qual o nome do pacote de software, a sua versão e qual o ficheiro principal de código fonte. O exemplo seguinte demonstra o que seria esperado para um qualquer programa *Java* a realizar como parte do pacote *guiao-6*. Também é utilizada a macro `AC\_PROG\_GCJ` pois vamos utilizar uma aplicação *Java*.

```
AC_INIT([src/foo.java])
AM_INIT_AUTOMAKE([guiao-6], [0.1])
AM_PROG_GCJ
```

Após a criação deste ficheiro é necessário inicializar o sistema *autoconf*. Para isso é necessário executar os commandos:

- `aclocal`: Cria uma base de dados local para utilização no projeto. Esta base de dados é construída com base nas macros que se encontrem definidas no ficheiro `configure.ac`.
- `automake -a`: Processa os ficheiros `Makefile.am` e gera ficheiros adicionais necessários para a instalação.

· `autoconf`: Gera o *script* `configure`.

Exercício 6.8

Na raíz da sua área de trabalho crie uma pasta chamada `guiao-6-auto`. Lá dentro crie dois directórios: `src` e `doc`. No directório `src` coloque um qualquer programa em *Java*[a].

No directório `guiao-6-auto` crie um ficheiro chamado `configure.ac` com as macros apresentadas anteriormente.

Execute `aclocal`, `automake -a` e `autoconf`. Verifique que existe um ficheiro chamado `configure`.

Execute o ficheiro `configure`.

[a]Caso não tenha um programa disponível, escreva um que imprima uma mensagem fixa tal como "Laboratórios de Informatica"

Até agora o script `configure` apenas irá realizar algumas verificações muito básicas e incompletas para qualquer projeto. É necessário adicionar macros de forma a configurar o projeto para que este possa ser produzido no sistema. A sequência de macros a utilizar varia para cada projeto de acordo com as suas necessidades. Neste caso em concreto, será necessário realizar validações em relação à existência de compiladores *Java* na versão correta e do sistema de produção de documentos LaTeX.

A macro `AC_CHECK_PROG` permite verificar a existência de programas específicos no sistema e segue a sintaxe seguinte:

```
AC_CHECK_PROG (variável, programa-a-verificar, se-encontrado,
      se-não-encontrado, caminho, ignorar)
```

Em que:

· `variável`: o nome de uma variável onde se armazena o resultado do teste.

· `programa-a-verificar`: o nome do programa a verificar (ex. `javac`).

· `se-encontrado`: que valor deverá a variável ter caso o programa seja encontrado.

· `se-não-encontrado`: que valor deverá a variável ter caso o programa não seja encontrado (opcional).

- `caminho`: directórios onde procurar (opcional).
- `ignorar`: localizações que se ignoram. Serve para excluir versões antigas ou conhecidas por não funcionarem como necessário.

A variável pode depois ser utilizada numa condição para agir de forma adequada. Aplicado ao caso do `javac` podemos definir:

```
AC_CHECK_PROG(EXISTE_JAVAC,javac,yes)

if test "$EXISTE_JAVAC"; then
    AC_MSG_NOTICE([[Compilador de Java encontrado.])
else
    AC_MSG_ERROR([[Compilador de Java em falta.])
fi
```

Repare como a variável `$EXISTE_JAVAC` foi utilizada numa condição para informar o utilizador que o compilador foi encontrado (`AC_MSG_NOTICE`), ou para abortar a configuração com uma mensagem (`AC_MSG_ERROR`) de erro. Caso o programa `javac` não exista, o resultado deverá ser o seguinte:

```
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for a thread-safe mkdir -p... /bin/mkdir -p
checking for gawk... no
checking for mawk... mawk
checking whether make sets $(MAKE)... yes
checking for javac... no
configure: error: Compilador de Java em falta.
```

Exercício 6.9

Utilizando a metodologia anterior escreva no seu ficheiro `configure.ac` todas as verificações que acha necessárias para construir devidamente a documentação e o código fonte *Java*.

Depois terá de voltar a executar os comandos `aclocal`, `autoconf` e `automake`.

Verifique a execução do novo `configure` gerado.

Além de se saber que existem as aplicações necessárias, também é necessário saber se as aplicações possuem a versão correta, ou são capazes de suportar as funcionalidades pretendidas. Até agora, verificou-se que existe LaTeX e *Java* mas não é sabido se o sistema consegue realmente compilar documentos, ou se a versão de *Java* é a correta. Determinar a versão de uma aplicação implica utilizar funcionalidades da `bash`, nomeadamente executar comandos e filtrar o seu resultado.

```
AC_MSG_CHECKING([Verificando versao do Java])
JAVA_VERSION=$(java -version 2>&1 |grep "java version" |cut -d '.' -f 2)
AC_MSG_RESULT($JAVA_VERSION)

if test $JAVA_VERSION -lt 6; then
        AC_MSG_ERROR([Necessario Java versao 6])
fi
```

No exemplo anterior o comando `grep` filtra as linhas da execução do comando `java -version` de forma a obter apenas a linha com informação de versão. O comando `cut -d '.' -f 2` separa esta linha pelo caráter "." e devolve o segundo item. Repare que na comparação não é utilizado o operador "<" como está habituado a utilizar, mas sim o operador "`-lt`" que possui o mesmo significado: *less than*. Isto é uma particularidade da `bash` que utiliza "`-lt`" quando os operandos são inteiros e "`==`" quando os operadores são sequências de caráters (Strings).

Exercício 6.10

Execute o comando `java -version` e analise o que é impresso no ecrã e reconstrua o comando do exemplo.

Adicione ao seu `configure.ac` as verificações necessárias para verificar que possui *Java* versão 6 e `pdflatex` versão 2.4.

Também é possível verificar pela existência de bibliotecas específicas, ou no caso de se utilizar LaTeX a existência de *packages*. No entanto, para o caso de LaTeX ou *Java* não existem macros que realizem estas funções automaticamente, pelo que se torna necessário utilizar comandos externos.

Para verificar se existe um *package* de *Java* pode-se utilizar o comando `kpsewhich nome-da-package.sty` e observar o seu resultado. Para isto recorre-se à macro `AC_MESSAGE_CHECKING` para indicar uma mensagem de um teste personalizado.

```
AC_MSG_CHECKING(Verificando existencia do package biblatex)
HAVE_BIBLATEX=$(kpsewhich biblatex.sty)
AC_MSG_RESULT($HAVE_BIBLATEX)
if test "x$HAVE_BIBLATEX" == "x"; then
        AC_MSG_ERROR(Falta package biblatex)
fi
```

Esta fórmula para escrever testes pode ser utilizada para qualquer validação que se pretenda. Por exemplo, para verificar se existem ficheiros em localizações específicas.

Exercício 6.11

Adicione testes ao seu `configure.ac` de forma a validar a existência das *packages* que utiliza.

Também pode ser importante validar se o sistema consegue realmente produzir um produto final pretendido. Neste caso será a documentação e a aplicação. Para isso é possível definir testes que efectivamente compilam pequenos programas de forma a validar o ambiente de compilação. Estes testes podem validar a existência de *packages* ou de *classes*. O exemplo seguinte valida se é possível compilar aplicações *Java* e se é possível utilizar a o método `exit` da classe *java.lang.System*.

```
# Preparar teste
AC_MSG_CHECKING(Verificando possibilidade de compilar programas Java)
    cat <<__EOF__ >conftest.java [
import static java.lang.System.*;

public class conftest {
  public static void main(String[] args) {
    exit(0);
  }
}]
__EOF__

# Compilar
javac conftest.java

# Testar resultado
if test $? = 0; then
    AC_MSG_RESULT([ok])
else
    AC_MSG_ERROR([ERRO])
fi
```

```
# Limpar ficheiros produzidos.
rm -f conftest.java conftest.class
```

Exercício 6.12

Construa testes de forma a verificar se o sistema é capaz de gerar documentos com os *packages* necessários e compilar aplicações *Java*.

Por fim, caso todos os testes tenha completado corretamente, é necessário gerar os ficheiros `Makefile` para compilar o programa. Embora não tenha sido abordado neste guião, é possível adaptar os `Makefile` de acordo com o que foi sendo detetado na execução do *script* `configure`.

```
AC_CONFIG_FILES([
  Makefile
  src/Makefile
  doc/Makefile
])
AC_OUTPUT
```

Exercício 6.13

Adicione ao seu ficheiro `configure.ac` a indicação para se gerarem os ficheiros `Makefile` necessários.

Verifique que ele o tenta fazer. Tenha em atenção que ainda não deu indicações de como construir estes ficheiros.

6.3.2 GNU Automake

A ferramenta *automake* encaixa na *autoconf* permitindo definir ficheiros `Makefile` de uma forma mais simplificada e adaptável ao sistema alvo. Neste caso são utilizados ficheiros `Makefile.am`, que a ferramenta converte para ficheiros `Makefile.in`. O *script* `configure` irá utilizar estes ficheiros para gerar os ficheiros `Makefile` finais.

Para o exemplo seguido é necessária a criação de um ficheiro `Makefile.am` na raíz do projeto indicando que sub-diretórios existem, seguido de ficheiros `Makefile.am` em cada directório. O conteúdo do ficheiro `Makefile.am` na raíz do projeto seria:

```
SUBDIRS = src doc
```

De realçar que a ferramenta *automake* também necessita que existam vários ficheiros de texto na raíz do projeto. Estes ficheiros não são necessários para os testes ou a compilação mas são obrigatórios de existir a quando da utilização desta ferramenta.

Exercício 6.14

Crie um ficheiro `Makefile.am` na raíz do seu projeto e outros em cada um dos directórios. Estes podem ser vazios.

Execute o comando `automake` e crie os ficheiros em falta. O conteúdo não é relevante. No final deverá obter vários ficheiros `Makefile.in` e o `automake` não deverá mostrar qualquer erro.

Verifique que o *script* `configure` gera ficheiros `Makefile` e inspecione o seu conteúdo.

Depois de existirem ficheiros `Makefile.am` o *script* `configure` irá gerar ficheiros `Makefile` com bastante informação e uma grande panóplia de alvos. Para além dos típicos *all* e *clean*, outros como *distclean* ou *install* são igualmente criados.

Exercício 6.15

Verifique que alvos são criados. Verifique por exemplo qual a utilidade do alvo *distclean*.

Exercício 6.16

Em cada um dos ficheiros `Makefile.am` adicione regras para os alvos *all* e *clean*, de forma a compilar e gerar documentação.

Na raíz do projeto execute `make`. Que observou?

Uma funcionalidde de um dos alvos é a de criar um pacote *.tar.gz* para distribuição imediata, contendo todo o programa e documentação[1]. Para isto é apenas necessário que se declarem os ficheiros a incluir neste pacote. Neste exemplo, para incluir o código fonte

[1]Para verificar o conteúdo execute: `tar -ztf nome-do-ficheiro.tar.gz`

será necessário adicionar a seguinte informação ao ficheiro `src/Makefile.am`:

```
bin_PROGRAMS = foo
foo_SOURCES = Foo.java

%.class:
    javac $*.java

foo : Foo.class

all: foo
```

Exercício 6.17

Corrija os seus ficheiros `Makefile.am` de forma a que ao executar `make dist` seja incluído todo o projeto num ficheiro *.tar.gz*.

Para incluir a documentação, outro directório ou ficheiro qualquer, é necessário que seja definida a variável `EXTRA_DIST` no ficheiro `Makefile.am` principal. Neste caso:

```
EXRA_DIST = doc/doc.tex
```

Exercício 6.18

Adicione o directório de documentação à lista de directórios a incluir no ficheiro *.tar.gz*.

Invoque `make dist` e verifique que funciona.