DEI
DEPARTAMENTO
DE ENGENHARIA INFORMÁTICA
TÉCNICO LISBOA

O desenvolvimento de aplicações em Programação com Objetos

João D. Pereira

Nuno Mamede

Outubro de 2013

Resumo

O objetivo deste documento é apresentar o modo de desenvolvimento de aplicações a usar durante a concretização do projeto da disciplina *Programação com Objetos*. Este documento descreve a arquitetura genérica do projeto a desenvolver nesta disciplina e uma framework a utilizar na concretização do projeto. É ainda descrita a forma de avaliação automática utilizada nesta disciplina e alguma informação sobre o processo de compilação e execução de programas Java e construção de arquivos JAR.

1 Arquitetura Genérica

As aplicações a desenvolver no contexto da disciplina de Programação com Objetos seguem a arquitetura em Camadas. Neste tipo de arquitetura, uma aplicação é constituída por várias camadas. Cada camada tem uma semântica bem definida e apenas comunica com as camadas adjacentes.

Este tipo de arquitetura tem algumas vantagens importantes. Uma é limitar o impacto das alterações de uma camada nas restantes camadas da aplicação. Apenas as camadas adjacentes à camada alterada poderão ter que ser modificadas. Outra é que é possível alterar completamente a concretização de uma camada e desde que se respeite a interface dessa camada não é necessário alterar nenhuma das restantes camadas da aplicação. No contexto das aplicações a desenvolver na disciplina de Programação com Objetos, as aplicações vão ter três camadas:

- Domínio;
- Serviços; e
- Interface com o utilizador.

A camada de Domínio contém as entidades e respetivas funcionalidades que pertencem ao domínio da aplicação a desenvolver. A camada de Domínio vai ser constituída por um conjunto de classes que modela e realiza o problema a resolver. Por exemplo, numa aplicação bancária, as entidades do domínio seriam as classe `Banco`, `Cliente`, `Conta`, `DepsitoPrazo`, etc.

A lógica de negócio associada a cada entidade do domínio deve estar concretizada na própria entidade. Desta forma é mais fácil perceber a semântica de cada entidade, dado que está concentrada na classe que representa a entidade e não espalhada por várias classes da aplicação. Esta concentração facilita a manutenção e evolução da aplicação. Ao mesmo tempo evita-se que a mesma funcionalidade acabe por ser replicada em várias classes da aplicação.

A camada de serviços é uma camada simples e que representa apenas as funcionalidades da aplicação a desenvolver que se querem disponibilizar ao utilizador. Por exemplo, uma funcionalidade no contexto de uma aplicação bancária pode ser a criação de uma conta para um dado cliente de um dado banco. Cada funcionalidade a apresentar ao utilizador deve ser representada por uma subclasse da classe `Command` (ver secção 2.3). Estas subclasses normalmente são muito simples e devem corresponder a invocar a funcionalidade correta de uma dada entidade do Domínio. Não devem conter lógica de negócio por forma a evitar-se o descrito anteriormente. Por isso, cada subclasse deve não deve realizar operações de cálculo

ou gestão (que representam lógica de negócio) mas deve apenas recolher dados, passá-los a um ou mais métodos de entidades do Domínio, e caso seja necessário apresentar o resultado da invocação desses métodos.

Finalmente, a camada de Interface com o Utilizador está relacionada, tal como o seu nome indica, com a interface com o utilizador. Esta camada e a de Serviços são concretizadas com o auxílio de uma framework. Esta framework (descrita na secção 2) facilita a construção da interface com o utilizador e da especificação de funcionalidades a disponibilizar ao utilizador. O tipo de interface suportado por esta framework pode ser visto como um conjunto de ecrãs, em que cada ecrã tem um determinado conjunto de opções. Cada opção está associada a uma dada funcionalidade a disponibilizar ou a um novo ecrã.

Ao concretizar aplicações com esta arquitetura é possível alterar a forma como se interage com o utilizar (por exemplo utilizar uma interação gráfica em vez de textual) sem que isso tenha qualquer impacto na camada de Domínio.

2 Descrição da Framework

As classes que constituem a framework utilizada para o desenvolvimento das aplicações na disciplina *Programação com Objetos* pertencem ao pacote `pt.utl.ist.po.ui`. A interação com o utilizador é realizada apenas através das entidades `Menu`, `Form` e `Display`. A especificação das operações a disponibilizar ao utilizador é realizado através da entidade Command. Um menu representa uma lista de opções de ações que o utilizador pode escolher. A classe Form permite ler dados simples introduzidos pelo utilizador da aplicação: números inteiros ou reais, cadeias de carateres ou valores booleanos (sim ou não). A classe `Display` é utilizada para informar o utilizador dos resultados de uma interação anterior, apresentando ao utilizador as mensagens (cadeias de carateres) pretendidas.

De seguida vão ser descritos os vários passos a realizar durante a concretização da interface com o utilizador para uma aplicação genérica, descrevendo-se ao mesmo tempo, com algum detalhe, as funcionalidades principais das classes principais da framework a utilizar no desenvolvimento desta interface.

A framework a ser utilizada no desenvolvimento dos projetos da disciplina de Programação com Objetos tem os seguintes objetivos:

- facilitar o teste automático da funcionalidade dos projetos;
- suportar diferentes tipos de interfaces;

- garantir a independência entre a camada de Domínio do projeto e as restantes camadas do projeto; e
- facilitar o desenvolvimento da interface com o utilizador.

2.1 A interligação entre a camada Domínio e a interface com o utilizador

Dado que o domínio da aplicação deve ser independente da interface com o utilizador utilizada, é o código que contém a interação com o utilizador que tem que invocar operações sobre entidades do domínio por forma a realizar os pedidos feitos pelo utilizador e não o contrário (o código do domínio a invocar código relacionado com a interação com o utilizador). Devido a esta razão, o código relacionado com a interação com o utilizador é normalmente parametrizado com uma instância de uma dada classe do domínio que contém a a concretização da funcionalidade pretendida pelo utilizador.

Tendo em conta a forma forma como a interface com o utilizar é especificada na framework, a aplicação a concretizar poderá ter que criar uma entidade adicional na camada de Domínio que representa o ponto de entrada na camada de Domínio. Este ponto de entrada, designado por `App` neste documento, deve depois ser passado ao menu principal da aplicação por forma a ser possível invocar ações sobre o domínio como resultado das escolhas realizadas pelo utilizador.

Note que este ponto de entrada não necessita de ser sempre o mesmo para toda a aplicação. Normalmente este ponto de entrada varia de menu para menu. Depende fundamentalmente da entidade sobre a qual as operações a disponibilizar ao utilizador dizem respeito. Por exemplo, numa aplicação bancária, o ponto de entrada pode ser uma conta bancária ou um cliente de um banco para o menu que disponibiliza as operações a realizar sobre uma conta ou cliente, respetivamente.

Na aplicação a desenvolver é necessário concretizar o seu ponto de entrada. Normalmente, neste ponto de entrada deve ser criado uma instância da classe `App` e o menu inicial da aplicação a desenvolver. Considere que este ponto de entrada é o método `main` da classe `Main`. O exemplo seguinte a funcionalidade mínima que deve estar presente neste método:

```
public static void main(String args[]) {
   App app = new App();
   Menu MainMenu = new MainMenu(app);
   IO.setTitle("App Menu");
   mainMenu.open();
```

```
   IO.close();
}
```

O menu inicial da aplicação é representado neste exemplo pela classe `MainMenu` e deve ser concretizado pelo programador da aplicação. Esta classe estende a classe `pt.utl.ist.po.ui.Menu` disponibilizada pela framework. A colocação de um título através da invocação de `IO.setTitle()` é facultativo e serve para identificar a janela de interação em algumas interfaces gráficas. Para ativar um menu, tornando-o visível e permitindo ao utilizador a seleção de opções associadas ao menu, basta invocar o método `open()` sobre o menu. Quando a interação com o utilizador termina deve ser invocado o método `IO.close()`. A invocação deste método permite libertar recursos como, por exemplo, janelas em ambientes gráficos.

2.2 Construção de um Menu

Tal como já foi indicado anteriormente os menus a concretizar pelo programador são classes derivadas da classe `pt.utl.ist.po.ui.Menu` disponibilizada na framework. O construtor da classe `Menu` recebe o título do menu (do tipo `String`) e o vetor de opções (do tipo `Command<?>[]`) associado ao menu a construir.

Cada opção é representada por uma instância de uma classe derivada da classe `pt.utl.ist.po.ui.Command` disponibilizada na framework. Estas opções podem ser realizadas por classes anónimas, como se pode ver no exemplo seguinte, ou por classes com nome (ver secção sobre a classe `Command`). De seguida apresenta-se um exemplo da criação do menu `MainApp`:

```
public class MainMenu extends Menu {
   public MainMenu(App app) {
      super(MenuEntries.TITLE, new Command[] {
         new Command<App> (false, "Título Opção 1", app) {
            public final void execute() {
               App app = entity();
               Display d = new Display(title());
               d.add("MyMenu.execute() called");
               d.display();
            }
         },
         new Opcao2(app),
         new SubMenu(app)
      });
      entry(2).invisible();
   }
}
```

No exemplo indicado, `MenuEntries.TITLE` representa uma variável estática do tipo *String* da classe `MenuEntries` que deverá conter o título a associar a este menu (por exemplo, *"Comandos do Menu Principal"*).

Os dois métodos mais importantes da classe `Menu` são `open()`, que apresenta o conjunto de opções ao utilizador, e `entry(intpos)`, que devolve o comando que representa a opção indicada na posição `pos`.

As opções associadas a um menu podem ser visíveis ou não. Por vezes dependendo do estado do domínio da aplicação há opções que não fazem sentido estar disponíveis. A invocação da operação `entry(2).invisible()` torna a segunda opção do menu `MainMenu` não selecionável, até que o método `visible()` seja invocado sobre a instância de `Command` que representa esta opção.

Considerando a interface em modo texto suportada pela framework, a execução do método `open()` sobre uma instância da classe `MainMenu` apresentaria ao utilizador o seguinte:

```
Comandos do Menu Principal
1 - Título Opção 1
3 - Título opção 3
0 - Sair
Escolha uma opção:
```

Relembra-se que não é apresentado nada relativo à opção dois porque esta se encontra invisível. Uma vez que ela passe a estar visível será indicado uma nova linha entre as linhas correspondentes às opções `1` e `3` com o texto *2 – Título da opção 2*.

Uma vez invocado o método `open` sobre um menu, entra-se num ciclo em que se pede ao utilizador qual é a opção que quer realizar, lê-se a opção pretendida pelo utilizador e executa-se a instância da classe `Command` correspondente à opção indicada. Quando é escolhida a opção `0`, a execução do menu termina e volta-se ao ponto seguinte no código em que o menu foi *aberto* (invocação do método `open`). Caso seja indicada uma opção inválida, esta é ignorada e é pedido ao utilizador que indique uma nova opção.

Esta abordagem para a criação dos menus tem a desvantagem de definir uma subclasse de Menu com apenas um método (o construtor). Uma outra abordagem seria ter uma única classe que seria responsável por criar todos os menus da aplicação. Cada menu a utilizar na aplicação seria criado e iniciado por um determinado método estático desta classe. O exemplo seguinte descreve a classe *AppMenuBuilder* que segue esta abordagem:

```
public class AppMenuBuilder {
     public static void openMainMenu(App app) {
         Menu main;
         main = new Menu(MenuEntries.TITLE, new Command[] {
             new Command (false, "Título Opção 1", app) {
                 public final void execute() {
                     App app = entity();
                     Display d = new Display(title());
                     d.add("MyMenu.execute() called");
                     d.display();
                 }
             },
            new Opcao2(app),
            new SubMenu(app)
         });

         main.entry(2).invisible();
         main.open();
     }
     // ...
 }
```

Por razões de simplificação, neste exemplo apenas se apresenta o método estático com a responsabilidade de criar e abrir o menu com os comandos da menu principal. Sempre que no código da aplicação fosse necessário abrir o menu principal teria que ser executado a seguinte instrução:

```
AppMenuBuilder.openMainMenu(app);
```

onde *app* contem uma referência para uma instância da classe *App*.

2.3 A classe Command

Cada classe `Command` a concretizar representa a funcionalidade associada a uma dada opção disponibilizada ao utilizador via um menu. Normalmente, as instâncias das subclasses de `Command` necessitam de conhecer uma instância (ou várias) de uma entidade do domínio sobre a qual vão realizar uma ou mais ações por forma a realizarem a funcionalidade pretendida. Por esta razão, a classe `Command` é uma classe genérica que recebe como parâmetro a classe principal da aplicação, ou eventualmente uma outra classe (normalmente pertencente ao domínio da aplicação). A classe indicada (designada como *classe parâmetro*) permitirá ao comando a desenvolver realizar a sua funcionalidade. Depois, no processo de criação de

instâncias da subclasse `Command` deverá ser passado uma instância da *classe parâmetro* no construtor da super classe `Command`. Esta instância pode ser acedida dentro da subclassse de Command através do método `entity()`. Esta instância é normalmente recebida como um argumento do construtor da subclasse de Command a desenvolver. Os comandos são criados dentro do contexto de um menu pelo que a instância do domínio a passar aos construtores das subclasses de Command tem que ser conhecida pelo menu.

Um comando tem duas propriedades: ser o último comando e estar visível. Por omissão, quando um comando é criado ele está visível. Se um comando estiver invisível, então a opção correspondente a este comando não será apresentada ao utilizador pelo menu a que este comando pertence. A alteração do valor desta propriedade é realizado através da invocação dos métodos `visible()` (torna o comando visível) e `invisible()` (torna o comando invisível).

Por vezes acontece que quando um dado comando de um menu é executado já não faz sentido apresentar novamente a lista de opções deste menu e deve-se sair do menu. Por exemplo, considerando a aplicação *editor*, podemos ter um menu que apresenta as funcionalidades que podem ser executadas sobre a forma geométrica selecionada. Uma dessas funcionalidades pode ser apagar a forma selecionada. Quando a opção correspondente a esta funcionalidade é escolhida pelo utilizador não faz sentido voltar a apresentar as operações disponíveis para a forma em causa dado que ela já foi removido do editor.

Esta característica dos comandos é representada pela propriedade *ser o último comando*. O valor desta propriedade é indicado no momento de criação de um comando através do construtor da classe `Command`. Do ponto de vista de um menu, a forma como esta funcionalidade é suportada é a seguinte. Quando um menu é aberto, ele fica em ciclo até o utilizador escolher a opção `0` ou então até ser executado um comando que tenha esta propriedade com o valor verdadeiro.

Existem vários construtores disponíveis na classe `Command`. Os mais importantes são:

- `Command(booleanlast,Stringtitle,Entityentity)` – o comando criado pode ser o último comando a ser executado neste menu (dependendo do valor de *last*), tem o título *title* e está associado à entidade de domínio *entity*.
- `Command(booleanlast,Stringtitle)` – semelhante ao caso anterior mas o comando criado não está associado a qualquer entidade do domínio.
- `Command(Stringtitle)` – o comando criado tem o título *title*, não é o último comando a ser executado e não tem nenhuma entidade do domínio

associada.

O título de um comando é utilizado pelo menu quando apresenta a lista de opções associadas ao menu. Cada opção é representada pelo seu número e pelo título do comando associado à opção. O método `title()` de `Command` permite aceder ao título do comando.

A funcionalidade da subclasse de `Command` a realizar deve ser especificada no método `execute()`. Este método, definido na classe `Command`, é um método abstrato e que portanto deve ser substituído em cada subclasse concreta de `Command` com a funcionalidade pretendida. Caso aconteça alguma situação de erro no contexto da execução deste método, deve ser lançada uma exceção que descreva a situação de erro. A exceção a lançar deve ser uma subclasse de `pt.utl.ist.po.ui.InvalidOption`.

Por vezes, uma opção pode corresponder a apresentar um novo menu. Neste caso, a classe `Command` a construir deverá criar uma instância do novo menu a presentar e invocar o método `open()` sobre esta instância para o ativar. Quando este outro menu terminar (o utilizador escolheu a opção `0` associada a este menu), o controlo do fluxo da aplicação regressa ao menu atual. O exemplo seguinte apresenta como deve ser codificada esta situação:

```
public class SubMenu extends Command<App> {
   public SubMenu(App app) {
      super(false, "SubMenu title", app);
   }

   public final void execute() {
      Menu m = new OtherMenu(entity());
      m.open();
   }
}
```

2.4 Pedido de valores ao utilizador

Pode acontecer que para realizar uma dada funcionalidade a aplicação tenha que pedir ao utilizador para inserir determinados dados. A leitura dos dados tem que ser realizada através da classe `pt.utl.ist.po.ui.Form` disponibilizada na framework a utilizar no desenvolvimento da aplicação. Esta classe permite agrupar diversos pedidos de leitura de dados numa só interação. Os pedidos de leitura resumem-se a valores de quatro tipos da linguagem Java: `int`, `float`, `boolean` e `String`. Para cada valor a pedir ao utilizador deverá ser criada uma instância

das classes `InputInteger`, `InputFloat`, `InputString` e `InputString`[1]. A classe a utilizar para ler os dados pretendidos depende do tipo de dados a ler. Por exemplo, caso se queira ler um número inteiro deve-se utilizar uma instância da classe `InputInteger`.

Cada um dos construtores destas subclasses de `Input` tem que receber a `Form` onde deve ser integrado o pedido e uma mensagem descritiva do pedido (por forma a que o utilizador saiba o que deve introduzir). Existe ainda um separador, representado pela classe `InputNone`, que é utilizado apenas por razões estéticas para separar pedidos dentro do mesmo *form*. Este tipo de `Input` não pede ao utilizador para inserir qualquer valor quando é processado pelo *form*, apenas serve para apresentar a mensagem que lhe está associado.

Para pedir ao utilizador para inserir os diversos valores agrupados num *form* deve-se utilizar o método `parse()` da classe `Form`. Ao executar-se este método sobre uma instância de `Form`, vão ser realizados os pedidos de inserção de dados agrupados no *form*. A ordem pela qual os pedidos são feitos é a mesma pela qual foram associadas a este *form* as instâncias das subclasses `Input` (ou seja, a ordem pela qual as instâncias de `Input` foram criadas). O código seguinte mostra o método `execute()` da subclasse de `Command` responsável por criar uma linha na aplicação *editor*:

```
public final void execute() throws InvalidOperation {
  Form f = new Form(title());
  InputInteger x1 = new InputInteger(f, "Origin X coordinate? ");
  InputInteger y1 = new InputInteger(f, "Origin Y coordinate? ");
  InputInteger x2 = new InputInteger(f, "End X coordinate? ");
  InputInteger y2 = new InputInteger(f, "End Y coordinate? ");
  f.parse();

  Line l = new Line(entity(), x1.value(), y1.value(),
                    x2.value(), y2.value());
  // apresentação de informação ao utilizador
}
```

Uma vez efetuados todos os pedidos ao utilizador de introdução de dados é necessário aceder a cada um dos valores inseridos. Esta funcionalidade é realizada através do método `value()` de cada uma das instâncias de `Input` introduzidas no *form*.

Atenção: *Só podem* ser realizados pedidos de inserção de dados utilizando a funcionalidade da classe `Form`. Casos os dados sejam lidos utilizando outro meca-

[1]Estas quatro classes encontram-se disponíveis no pacote `pt.utl.ist.po.ui` e são subsclasses da classe abstrata `Input`.

nismo disponível no Java, o seu projeto terá uma avaliação automática igual a `0`, como explicado na secção 3.

2.5 Apresentação de mensagens ao utilizador

A apresentação de mensagens para o utilizador realiza-se através da classe `Display`. A funcionalidade desta classe é suportada por quatro métodos:

- `display(booleanforce)`,
- `display()`
- `add(Stringmsg)`
- `addNewLine(Stringmsg)`

O texto a ser escrito pela aplicação é indicado no argumento passado aos métodos `add(Stringmsg)` e `addNewLine(Stringmsg)`. Estes dois métodos vão recolhendo o texto a apresentar ao utilizador, o qual vai ser sendo guardado na instância de `Display` que está a ser utilizada. A diferença entre estes dois métodos é que o método `addNewLine(String)` cria uma nova linha de texto (adição de "\n") antes de adicionar o texto passado como argumento ao texto já previamente recolhido.

O texto assim recolhido é apresentado ao utilizador quando é invocado `display()` ou `display(booleanforce)`. O primeiro método corresponde à execução do segundo com o argumento *false*. Caso não tenha sido inserido qualquer texto desde a invocação anterior do método `display`, então este método não faz nada a não ser que seja invocado com o argumento *true*, o que vai forçar a executar o código associado a este método mesmo que não tenha sido adicionado texto. De seguida apresenta-se um exemplo de utilização de `Display`:

```
public final void execute() throws InvalidOperation {
   // criação do Form f
   f.parse();

   Line l = new Line(entity(), x1.value(), y1.value(),
                     x2.value(), y2.value());
   Display d = new Display(title());
   d.add("-------- NEW OBJECT --------");
   d.addNewLine("Line #" + l.getId());
   d.addNewLine("---------------------------");
   d.display();
}
```

onde se apresenta uma mensagem ao utilizador, com três linhas de texto, sempre que o utilizador escolhe a opção de criação de uma *linha*.

2.6 Interfaces com o utilizador suportadas pela framework

A framework suporta dois tipos de interface com o utilizador: textual e gráfica. A interface com o utilizador por omissão da framework é a textual. Este tipo de interface é o utilizado pelo corpo docente da disciplina Programação com Objetos para efetuar os testes automáticos das aplicações submetidas pelos alunos. Assim, ao desenvolver o seu projeto deve garantir o bom funcionamento da aplicação na interface textual. A execução da aplicação com a interface textual pode ser efetuada numa janela de texto e não necessita de quaisquer parâmetros específicos:

```
java aplic.Main
```

Note que caso a variável de ambiente `CLASSPATH` não esteja corretamente definida é necessário indicar o seu valor em cada invocação do programa *java* ou *javac*:

```
java -cp .:Dir1:Dir2/ap.jar: aplic.Main
```

em que *Dir1* representa um diretório que contém classes a serem utilizadas nesta aplicação e *Dir2/ap.jar* representa a localização do arquivo JAR *ap.jar*, o qual deverá conter classes também necessárias para a execução ou compilação da aplicação.

A utilização da interface gráfica (`swing`) suportada pela framework é efetuada atribuindo o valor *swing* à propriedade *ui*:

```
java -Dui=swing aplic.Main
```

Embora experimental, também é possível executar a aplicação a partir de um browser utilizando um applet. Para tal basta incluir a sua invocação num ficheiro de hipertexto (*.html*). Neste caso a execução da aplicação desenvolvida já não se inicia pelo método `main()` da classe principal da aplicação. A classe de arranque neste caso é sempre `pt.utl.ist.po.ui.AppletInteraction`. Como parâmetro é necessário definir a localização do método `main()` de entrada da aplicação. Isto é feito através do preenchimento do parâmetro *mainClass*:

```
<applet width=300 height=300
        code="pt.utl.ist.po.ui.AppletInteraction"
        archive = 'aplic.jar'>
  <param name="mainClass" value="aplic.Main" >
</applet>
```

3 Testes de software

Por forma a validar o correto funcionamento das aplicações realizadas pelos alunos, o corpo docente da disciplina de Programação com Objetos aplica um conjunto de testes automáticos aos projetos submetidos pelos alunos. Cada teste é constituído pelos dados a serem inseridos durante a execução do teste (entrada do teste) e pelas mensagens que devem ser escritas pelo programa durante a execução do teste (saída esperada). Após ser aplicado um teste, considera-se que o teste passou com sucesso (ou seja, o programa teve o comportamento esperado) se as mensagens escritas pelo programa durante a execução do teste forem iguais à saída esperada do teste. Caso sejam diferentes, o teste falhou.

Para realizar os testes de software de uma forma automatizada, incluindo os testes disponibilizados ao longo do semestre, devem ser definidas duas propriedades quando o programa é executado. Estas propriedades, designadas por *in* e *out*, permitem redirecionar a interface com o utilizador, quando em modo texto. O valor indicado na propriedade *in* representa um ficheiro que contem os dados a serem lidos durante a execução do programa. O valor indicado na propriedade *out* indica um ficheiro onde serão guardadas as mensagens escritas pelo programa durante a execução do teste. Por exemplo, a execução da aplicação `aplic.Main` no contexto do seguinte teste:

```
java -Din=in.txt -Dout=out.txt aplic.Main
```

lê os dados do ficheiro *in.txt* e guarda as mensagens escritas pelo programa durante a sua execução no ficheiro no ficheiro *out.txt*.

Tendo em conta a forma como os testes automáticos são realizados, é fundamental que a leitura dos dados inseridos pelo utilizador seja realizado através das classes `Form` e `Input` (como indicado na secção 2.4). A escrita de mensagens deve ser realizada através da classe `Display` (como indicado na secção 2.5). Caso a aplicação desenvolvida não esteja de acordo com estas duas restrições haverá testes automáticos que poderão falhar apenas devido ao fato de o redirecionamento da entrada e saída de dados da aplicação não ser possível de fazer nas situações em que o programa lê ou escreve dados sem utilizar os mecanismos descritos nas secções 2.4 e 2.5. É da responsabilidade dos alunos garantir que isto não acontece, caso contrário terão uma avaliação de 0 em todos os testes que falhem.

4 Construção de arquivos JAR

O projeto desenvolvido pelos alunos deve ser entregue para avaliação como um ficheiro *jar*. A construção de um ficheiro jar (Java ARchive) permite agrupar

vários ficheiros num só ficheiro.

No caso de o ficheiro *jar* conter os ficheiros compilados (*.class*) da aplicação é possível indicar que o ficheiro *jar* pode ser executado como uma aplicação. Neste caso é necessário declarar um manifesto (ficheiro *MANIFEST.MF*) onde se indica, por exemplo, a classe de arranque (*Main-Class*) ou outros ficheiros *.jar* a utilizar (*Class-Path*). Por exemplo, considerando que a classe de arranque é `aplic.Main` e que o código da aplicação necessita de aceder às classes guardadas no ficheiro *pt.jar*, o ficheiro *MANIFEST.MF* deve conter o seguinte:

```
Main-Class: aplic.Main
Class-Path: pt.jar
```

Um ficheiro *.jar* executável, chamado *aplic.jar*, pode ser criado com o seguinte comando:

```
jar -cfm aplic.jar MANIFEST.MF *.class
```

onde **.class* representa todos os ficheiros *.class* da aplicação. A aplicação pode agora ser executada com o seguinte comando:

```
java -jar aplic.jar
```

Um ficheiro *.jar* também pode conter ficheiros com outras extensões que não apenas *.class*. O projeto a submeter para avaliação apenas deve ser guardado num ficheiro *.jar* **apenas** deve conter os ficheiros fonte (*.java*) criados durante o desenvolvimento da aplicação.

4.1 Exemplos de aplicações

Os exemplos utilizados nas aulas prática `bank` e `editor` (e disponibilizados também na página da cadeira) correspondem a aplicações, já com alguma complexidade, que foram desenvolvidas utilizando a arquitetura de camadas e a framework. Estes exemplos devem ser analisados pelos alunos por forma a consolidarem melhor os conceitos expostos neste documento.

Nestes exemplos é disponibilizado o ficheiro *Makefile* que permite a realização de várias tarefas de forma automática. Estas tarefas são executas com o auxílio do programa make:

- `makeall` – compila o projeto, e cria dois ficheiros *.jar*, um com o código fonte da aplicação (ficheiros *.java*) e outro com o código compilado da aplicação (ficheiros *.class*)
- `makeclean` – apaga todos os ficheiros .class guardados no diretório da aplicação
- `makerun` – executa a aplicação em modo textual
- `makeswing` - executa a aplicação em modo *swing*.

O ficheiro *Makefile* disponibilizado nestas aplicações pode ser usado pelos alunos no desenvolvimento dos seus projetos. Apenas é necessário indicar o diretório onde se encontra a aplicação. Para isso é necessário editar este ficheiro e alterar o valor atribuído à variável *PROJ*.