**ARQUITECTURA DE UMA APLICAÇÃO DESKTOP**

**EXEMPLO JAVA**

**F. Mário Martins (PSJ - 2018)**

**1.- Introdução Teórica.**

Desde o início dos anos 80 que o aumento da complexidade das aplicações conduziu ao desenvolvimento de linguagens mais seguras (cf. Pascal), mais modulares (cf. Modula), a novas formas de associar dados e respectivas operações (cf. Tipos Abstractos de Dados), a novos paradigmas de programação (cf. Bases de Dados Relacionais e POO) e a novas arquitecturas de software, visando satisfazer alguns princípios básicos da Engenharia de Software (ES).

Um dos princípios básicos da ES, designa-se por ***princípio da separação das camadas***, e sugere de forma clara a separação entre os *dados* (e suas *operações)* e a *apresentação* das aplicações, de modo a que alterações da apresentação (ou ***layout***) não afectem os dados e vice-versa. Surgem assim regras básicas como não introduzir I/O básico nos algoritmos de tratamento dos dados porque assim passam a estar dependentes da apresentação e outras.

O **padrão arquitectural** designado **MVC** (Model-View-Controller), foi descrito pela primeira vez em 1979 por Trygve Reenskaug, que trabalhava na linguagem Smalltalk, na Xerox Palo Alto Research Center (PARC), e separa a aplicação em ***3 camadas***, introduzindo o designado ***Controller*** entre a apresentação e a camada computacional. Smalltalk-80 foi leccionada como linguagem de base de POO no DI/UM desde 1984 até 1996, quando se introduziu Java.

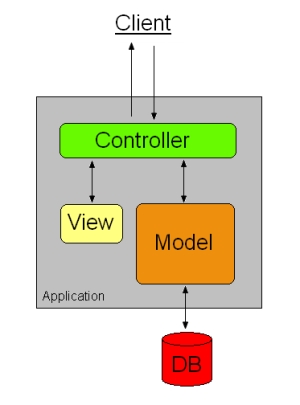
O ***Model*** (modelo) é a camada computacional, lógica, ou de "business", como hoje se chama também, e contém os dados e os algoritmos para o seu processamento, as pré-condições, os algoritmos de validação e o acesso a ficheiros ou bases de dados. O Model tem programada toda a possível computação a realizar, ***apenas não sabe quando a deve realizar***.

A ***View*** (vista ou apresentação) é a camada de apresentação e interacção com o utilizador. Pode ser textual ou gráfica, e a sua missão é criar e gerir o layout, receber os dados de entrada do utilizador e apresentar os resultados, instruções, avisos, mensagens de erro, etc.

O ***Controller*** é a camada fundamental de controlo do fluxo de execução da aplicação, e o mediador entre o model e a view. As regras de encapsulamento determinam que o Model não "conhece" a View, e a View não "conhece" o Model, ou seja, são módulos completamente independentes, acessíveis somente através das suas APIs (sejam programados em que linguagem forem) e, obviamente, não são entidades globais à aplicação. Porém, o Controller terá necessidade de comunicar à View e ao Model as modificações que estes devem fazer nos seus estados internos, em geral tratando de sincronizar os estados destas duas camadas. Para tal, é absolutamente indispensável que o Controller conheça a API da View e a API do Model, ou seja, as funções ou métodos que poderá invocar de cada um. Por isso, o Controller terá que possuir internamente (no seu estado) uma variável que lhe dê acesso via API ao Model e uma variável que lhe dê acesso à API da View. Tudo é encapsulado em módulos e tudo é acedido usando apenas as APIs. Nada é global.

O designado ***Programa Principal*** (ou Main) de uma aplicação deste tipo, qualquer que seja a sua forma e a sua linguagem de implementação, deverá criar um Model inicial, criar uma View inicial, inicializar o Controller, indicando-lhe qual o model e qual a view, e, depois, pôr em execução o Controller invocando o método adequado (cf. start(), launch(), init() ou outro).

A imagem seguinte visa ilustrar de forma genérica os relacionamentos entre as 3 entidades do modelo MVC, não se concretizando se elas se relacionam por invocação de métodos, por eventos, por ambos, ou até por protocolos especiais, dadas as imensas implementações existentes.



Em POO, em especial na codificação em JAVA de uma aplicação Desktop, o Model, a View e o Controller são programados em classes. As suas APIs são tudo o que programamos como **public** nos respectivos métodos e os seus dados são todos **private**. As instâncias de cada uma das classes necessárias a criar a aplicação são variáveis locais ao método que no programa principal vai criar e iniciar a execução da aplicação, ou seja, do método **public static void main(String[] args);** da classe especial que é o **programa principal**, cf. GestUnivApp, ChonoApp, etc.

Seguindo estes princípios estruturais básicos podemos até escrever um **esqueleto padrão para uma aplicação desktop Java** que satisfaz todos os preceitos do modelo MVC para além dos preceitos de escrita do código das suas classes (e interfaces, se a programação for baseada em interfaces). Simplesmente usando classes seria então algo como:

package ...

import ....

**public class GestAcademicaMVCApp {**

// Método auxiliar para carregar os dados; Exº leitura de um ficheiro;

// O método devolve uma instância de GestAcademicaModel e não void ...

private static GestAcademicaModel createData() { ........................... }

**public static void main(String[] args) {**

// Model ou Estado; Criado usando o método auxiliar de carregamento de dados

**GestAcademicaModel model = createData();**

// A View pode ser baseada em menus ou gráfica;

// Se for por menus, estes são criados na View e têm o texto das opções e

// métodos para fazer display de dados.

**GestAcademicaView view = new GestAcademicaView();**

// O Controller é que trata do fluxo de execução da Aplicação. Por isso, tem de

// "conhecer" (ter acesso ao) o Model e tem que "conhecer" a View

// Usa-os sempre através dos métodos das respectivas API.

**GestAcademicaController control = new GestAcademicaController();**

**control.setModel(model);**

**control.setView(view);**

// Model e View são injectados no Controlador e não passados no construtor.

// O Controlador tem duas variáveis de instância private de tipo

// GestAcademicaModel e GestAcademicaView, onde guarda o model e a view;

// O Controlador decide quais os menus ou componentes gráficos a apresentar e

// quando, lê os inputs do utilizdor e chama métodos do model e da view;

// Só o controlador faz isto. Se quisermos usar um model2 ou uma view2

// (implementações alternativas) invoca-se setModel() ou setView().

//

// Agora é só executar a aplicação, ou seja, executar o controlador.

**control.startController();**

// E aplicação terminará quando o utilizador decidir.

**System.exit(0);**

**}**

**}**

Se a programação for baseada em interfaces, usam-se as interface Java como tipos, e todas as classes que implementarem as interfaces são compatíveis. O padrão é o mesmo mas o código final é muito mais genérico e extensível (via polimorfismo).

package ...

import ....

**public class GestAcademicaMVCApp {**

// Auxiliar para os dados; Exº leitura de um ficheiro;

// O método devolve um InterfGestAcademicaModel e não void ...

private static InterfGestAcademicaModel createData() { ........................... }

**public static void main(String[] args) {**

// Model ou Estado

**InterfGestAcademicaModel model = createData();**

// A View pode ser baseada em menus ou gráfica;

// Se for por menus, estes são criados na View e só têm o texto das opções.

**InterfGestAcademicaView view = new GestAcademicaView();**

// O Controller é que trata do fluxo de execução da Aplicação. Por isso, tem de

// "conhecer" (ter acesso ao) o Model e tem que "conhecer" a View, mas via API.

**InterfGestAcademicaController control = new GestAcademicaController();**

**control.setModel(model);**

**control.setView(view);**

// Model e View são injectados no Controlador e não passados no construtor.

// O Controlador tem duas variáveis de instância private de tipo

// InterfGestAcademicaModel e InterfGestAcademicaView (interfaces Java);

// As variáveis model e view são instâncias compatíveis. O Controlador apresenta

// os menus ou componentes gráficos, lê os inputs e chama métodos do model e

// view. Só o controlador faz isso. Se quisermos usar um model2 ou uma view2

// (implementações alternativas) invoca-se setModel() ou setView().

//

// Agora é só executar a aplicação, ou seja, executar o controlador.

**control.startController();**

**System.exit(0);**

**}**

**}**

Se, por exemplo, a nossa aplicação tiver uma interface com o utilizador por menus textuais, apenas teremos que criar a view adequada usando os menus, mas tendo sempre em consideração que não é o código dos

menus que deve invocar métodos do model mas sim o código do controlador, que usa os menus para saber qual o fluxo indicado pelo utilizador nas suas opções. Confundir menus com formas de chamar métodos é muito comum mas é errado pois estaremos a juntar a vista com fluxo, ou seja, View e Controller.

O código do controlador é também muito simples (cf. exemplo):

**public class GestAcademicaController {**

**private GestAcademicaModel academia; // model**

**private GestAcademicaView viewMenus; // view**

**// métodos de instância**

public void setView(GestAcademicaView m) { menus = m; }

public void setModel(GestAcademicaModel acad) { academia = acad; }

**//**

**public void start() {**

Menu menu; String opcao;

**// Início do fluxo de execução**

do {

menu = menus.getMenuMain();

menu.show();

opcao = lerOpcao();

switch (opcao) {

case "I": ler\_inserir\_Aluno(); // lê dados e insere no modelo via método

break;

case "C": consultar\_Aluno(); // lê dados e invoca método do modelo

break;

..............................

default: System.out.println("Opção Errada!"); break;

}

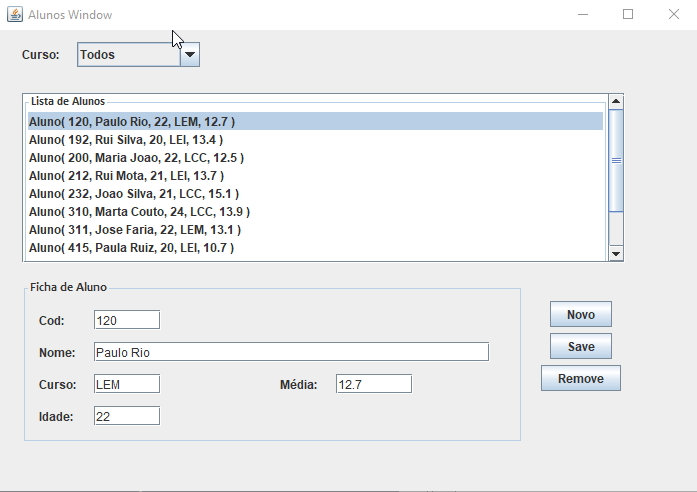
while (!opcao.equals("S");

........................................

**}**

**}**

Se, por exemplo, a View fosse programada em Swing, o programa principal seria tal como apresentado na página anterior, substituindo a view de menus por uma instância instância da classe view escrita em Swing. Ajustado o método start() do controlador (já que Swing se baseia em eventos e teríamos que programar no controlador o código de resposta a cada um dos eventos gerados na View), teríamos o mesmo model mas agora apresentado por uma IU gráfica. Exemplo:



Seja a interface textual ou gráfica, temos uma aplicação por camadas, modular (MVC), protegida, com classes altamente independentes e, em consequência, facilmente modificável e extensível.

------------------------------------------------------------------------------------------------------------------

O modelo MVC tal como apresentado, é especialmente adequado para aplicações desktop, mesmo usando ferramentas para criação de interfaces gráficas com o utilizador (GUI), tais como Swing ou JFX. Nestes casos, depois de deixarmos uma ferramenta gerar automaticamente todo o código da View, é necessário apenas ter uma metodologia para passar o tratamento dos eventos para o controlador.

O modelo MVC é tão relevante que hoje em dia é, quase por omissão, aplicado em diversos *frameworks*, que precisamente auxiliam o programador na criação da view e do controller para um dado model em aplicações para telemóveis, em aplicações web, etc.

A figura seguinte ilustra um modelo comum de implementação de MVC para aplicações web (cf. JSP, JSF, ASP.MVC), em que o cliente usa um browser e envia pedidos HTTP que são respondidos pelo servidor usando páginas HTML, depois de as várias camadas comunicarem entre si.



Os grandes frameworks de desenvolvimento de aplicações Java Web, cf. Spring MVC, Struts2.0, Wicket, etc. usam MVC, introduzindo cada um as suas particularidades.

**2.- Exemplo Prático (SIGAC - Sistema de Gestão de Alunos e Cursos)**

O exemplo prático simples que vamos usar para ilustrar as vantagens do modelo MVC, será um pequeno Sistema de Gestão de Alunos e Cursos (SIGAC).

**2.1.- O Model**

Sobre cada aluno pretendemos guardar o seu número (único), o seu curso (MIEI, LCC, MIEGSI, etc), o seu nome, a sua idade e a sua média actual (nº real).

A classe **Aluno** deve disponibilizar as seguintes operações:

* Sets e Gets;
* Construtores: vazio, por partes e de cópia;
* Métodos toString() e clone();
* Métodos que devolvem a idade e a média como String;

Assim, a ficha de cada aluno será da forma textual exemplo:

**Aluno("a92189", "Rui Silva", "MIEI", 20, 14.7)**

A classe **GestAcademicaModel** representará o "model" desta aplicação e numa primeira versão todas as informações sobre os alunos serão representadas por uma variável de instância do tipo **Set<Aluno>** implementada por um **HashSet<Aluno>.**

Esta classe deverá implementar (para além dos construtores) as seguintes operações sobre alunos,

* void insereAluno(String codAluno);
* boolean existeAluno(String codAluno);
* Aluno getAluno(String codAluno);
* void removeAluno(String codAluno);
* int numDeAlunos();
* Aluno[] toArrayAlunos();
* List<Aluno> toListAlunos();

e as seguintes operações envolvendo cursos:

* List<Aluno> alunosDoCurso(String curso);
* Aluno[] toArrayCurso(String curso);
* Set<String> cursos();
* boolean existeCurso(String codCurso);
* int numAlunosDoCurso(String curso);

Um esboço desta classe seria:

**public class GestAcademicaModel {**

private Set<Aluno> fichas = new HashSet<>();

public GestAcademicaModel() {}

public GestAcademicaModel(Set<Aluno> alunos) {

fichas = alunos;

}

**// métodos de instância**

......................

**}**

Numa 1ª versão, e por razões de simplificação, os dados iniciais para o "modelo" serão criados no programa principal através de um método de classe auxiliar. Posteriormente poderão ser lidos a partir de um ficheiro.

import ...

**public class GestAcademicaMVCApp {**

// createData() inicial; a substituir !

**private static GestAcademicaModel createData() {**

List<Aluno> fichas = Arrays.asList(

new Aluno("212", "Rui Mota", 21, "LEI", 13.7),

new Aluno("120", "Paulo Rio", 22, "LEM", 12.7),

.........

new Aluno("200", "Maria Joao", 22, "LCC", 12.5) );

**return new GestAcademicaModel(new HashSet<>(fichas));**

}

**public static void main(String[] args) {**

// Modelo MVC

GestAcademicaModel model = createData();

.......

**}**

**}**

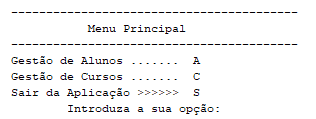
**2.2.- A View**

A View da aplicação será construída tendo por base menus textuais. Uma interface com o utilizador (IU) baseada em menus é uma forma simples de estruturar a funcionalidade da aplicação e apresentar aos utilizadores as várias opções disponíveis a cada momento.

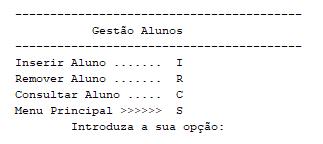
Os menus podem ser eles própios implementados de várias formas, mas no limite, um menu não é mais do que uma lista de linhas de texto (lista de strings).

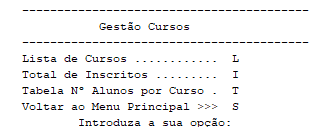
Para estruturar as opções de acesso à funcionalidade a aplicação vai ser dividida em duas subfuncionalidades a que chamaremos: Gestão de Alunos e Gestão de Cursos.

Assim, teremos um menu principal,



e dois submenus correspondentes a cada uma das subfuncionalidades identificadas.





Como sempre, todas as entradas do utilizador devem ser validadas.

A criação da classe que vai definir a camada de apresentação, a classe **GestAcademicaView**, consiste em definir cada um dos os menus da aplicação (3 neste caso), guardá-los numa estrutura e escrever o código dos métodos que permitam ao controlador ter acesso a cada um dos menus que num dado momento devem ser apresentados ao utilizor. A View é também responsável por todas as formas de apresentação dos resultados, que são métodos para output que também serão invocáveis pelo controlador para apresentar resultados.

Por exemplo, a classe **GestAcademicaView** terá uma estrutura do tipo:

**public class GestAcademicaView {**

public GestAcademicaView() {

menusGestAcademica = initView(); //

}

private Menus menusGestAcademica = new Menus();

// Cria a View baseada em Menus textuais

**public static Menus initView() {**

Menus menusSIGCA = new Menus();

Opcao op1, op2, op3, op4; // melhor: ArrayList<Opcao>

**// Criação do Menu de Gestão de Alunos;**

**// Layout definido aqui.**

op1 = new Opcao("Inserir Aluno ....... ", "I");

op2 = new Opcao("Remover Aluno ....... ", "R");

op3 = new Opcao("Consultar Aluno ..... ", "C");

op4 = new Opcao("Menu Principal >>>>>> ", "S");

List<Opcao> linhas = Arrays.asList(op1, op2, op3, op4);

Menu menuAlunos = new Menu(linhas, " Gestão Alunos");

menusSIGGA.addMenu(1, menuAlunos);

....... **// definição dos outros menus**

return menusSIGCA;

**}**

...... **// outros métodos static auxiliares e de instância**

**}**

Haverá também que definir as classes **Opcao** (texto e etiqueta de uma linha de um menu), **Menu** e **Menus** (a estrutura dos menus, no exemplo definida como um **Map<Integer,Menu>**).

**2.3.- O Controller**

O Controller será a camada de ligação do Model com a View, e a camada onde **programamos o fluxo de execução da aplicação**.

Não existindo questões prévias a colocar ao utilizador por razões de configuração (por exemplo, de que fonte carregar os dados, etc.), em geral o controlador, depois de receber do programa principal a instância do model e a instância da view com que irá trabalhar (ou via construtor ou, ainda melhor, usando métodos **setView()** e **setModel()**), inicia a execução da aplicação.

Portanto o Controller terá sempre uma estrutura de código típica, semelhante à que se esboça em seguida, a título de exemplo.

**public class GestAcademicaController {**

private GestAcademicaModel model;

private GestAcademicaView viewTxt;

public void setModel(GestAcademicaModel alunos) {

model = alunos;

}

public void setView(GestAcademicaView txtMenus) {

viewTxt = txtMenus;

}

**//-- Métodos auxiliares que invocam a API do Model**

**//-- para as operações de Gestão de Alunos -------**

private void flowAlunos() { ... }

private void insereAluno() { ... }

.......

**//-- Métodos auxiliares que invocam a API do Model**

**//-- para as operações de Gestão de Cursos -------**

private void flowCursos() { ... }

private void removeALuno() { ... }

.......

**// Método invocado pelo programa principal para execução**

**public void startFlow () {**

**// Início do fluxo de execução**

Menu menu = ........;

String opcao;

do {

menu.show();

opcao = Input.lerString();

opcao = opcao.toUpperCase();

switch(opcao) {

case "A" : flowAlunos(); break;

case "C" : flowCursos(); break;

case "S": break;

default: out.println("Opcão Inválida !"); break;

}

}

while(!opcao.equals("S"));

**}**

**}**

Os métodos auxiliares do controlador (por isso são privados) são imprescindíveis para a estruturação e legibilidade do código, que por vezes atinge grande dimensão e complexidade.

**2.4.- O Programa Principal**

O programa principal de uma aplicação Java consiste de uma classe particular, por exemplo **GestAcademicaMVCApp**, que conterá o método **main()** que desencadeará a execução. Com base nos nomes que demos às classes das 3 camadas, o código desta classe será simplesmente:

/\*\*

\* @author fmm - PDSJ 2018

\*/

import java.util.\*;

import static java.lang.System.out;

**public class GestAcademicaMVCApp {**

**// Método auxiliar p/ carregamento de dados**

**// createData() inicial; a substituir !**

**private static GestAcademicaModel createData() {**

List<Aluno> fichas = Arrays.asList(

new Aluno("212", "Rui Mota", 21, "LEI", 13.7),

.......

return new GestAcademicaModel(new HashSet<>(fichas));

**}**

**public static void main(String[] args) {**

**// Modelo MVC**

GestAcademicaModel model = createData();

GestAcademicaView view = new GestAcademicaView();

GestAcademicaController control =

new GestAcademicaController();

control.setModel(model);

control.setView(view);

control.startFlow();

//--------------------------------

out.println("Fim da Aplicação >> "

+ java.time.LocalDateTime.now());

System.exit(0);

**}**

**}**

**3.- Extensibilidade da Aplicação.**

A utilização do modelo MVC torna as aplicações muito mais modulares e estruturadas, sendo um modelo essencial em particular quando se desenvolvem aplicações complexas.

Por exemplo, é muito comum que ao longo do tempo de vida de numa aplicação a implementação do modelo seja modificada, por exemplo porque se encontrou uma implementação mais eficiente ou mais clara de manter, ou mesmo usando outras técnicas. Uma aplicação mal estruturada não sobrevive a uma reimplementação do model. MVC sim.

Vamos experimentar o que acontece, quando usamos o modelo MVC , para a arquitectura da nossa aplicação e alteramos o model.

Em especial vamos mudar a implementação de **GestAcademicaModel** passando a informação a ser representada por um **Map<String, Aluno>** que a cada código de aluno faz corresponder a sua ficha.

Teríamos agora o seguinte código para o model:

**public class GestAcademicaModel {**

private Map<String,Aluno> fichas = new TreeMap<>();

public GestAcademicaModel() {}

public GestAcademicaModel(Map<String,Aluno> alunos) {

fichas = alunos;

}

**// métodos de instância**

......................

**}**

Quais as implicações desta mudança do model na nossa aplicação ?

Bom, como os dados têm outra estrutura, o método **createData()** do programa principal tem que ser adaptado. Mas note-se que foi porque os dados mudaram! Ficaria:

**private static GestAcademicaModel createData() {**

Map<String,Aluno> fichas = new HashMap<>();

fichas.put("212",

new Aluno("212", "Rui Mota", 21, "LEI", 13.7));

.......

return fichas;

**}**

***E nada mais teria de ser modificado no resto da nossa aplicação !***

**------------------------------------------------------------------------------------------------------------------**

**Nota:**

Poderíamos ainda generalizar mais a aplicação se usassemos interfaces Java para além das classes. Verificar como !