Universidade da Beira Interior
Faculdade de Engenharia
Departamento de Informática

Programação de Dispositivos Móveis
Aula 2
Licenciatura em Engenharia Informática
Licenciatura em Informática Web

Programming of Mobile Devices
Lecture 2
Degree in Computer Science and Engineering
Degree in Web Informatics

Sumário
Discussão do tema relativo ao desenvolvimento de Interfaces de Utilizador como forma de introduzir conceitos específicos ao desenho de aplicações interativas, partindo das Interfaces de Utilizador para programas sequenciais. Introdução à arquitetura para desenvolvimento de aplicações interativas conhecida por *Model-View-Controller*.

Summary
Discussion of the subject concerning user interface development as an excuse to introduce concepts specific to the design of interactive applications, starting with the usr interfaces for sequential programs. Introduction to the Model-View-Controller software development architecture.

1 Desenvolvimento de Interfaces
User Interface Development

1.1 Introdução
Introduction

Um dos mais importantes componentes de aplicações móveis, pelo menos daquelas que interagem com o utilizador, é a *interface do utilizador*. Note-se que **nem todas as aplicações móveis têm de ter uma destas interfaces**, já que algumas são apenas implementadas para **fornecer serviços ou executar tarefas em segundo plano**. A definição deste componente é simples:

> Uma *interface do utilizador* é um programa de computador, ou parte dele, que permite que o utilizador comunique com esse computador ou com outras partes do programa.

As Interfaces de Utilizador têm **evoluído muito** ao longo da história da informática, sendo essa evolução motivada quer **por desenvolvimentos de *hardware* quer de *software*.** De modo a introduzir o *desenvolvimento de Interfaces de Utilizador utilizando programação orientada por eventos*, talvez seja indicado começar por referir o modelo de interfaces usado em programas sequenciais. Assim, a próxima subsecção discute o modelo mencionado em último, enquanto que a seguinte discute o anterior.

1.2 Programas Sequenciais
Sequential Programs

Os modelos de interação **para programas sequenciais**, baseadas em dispositivos de entrada como o teclado, **são bastante simples**. Note-se, neste caso, são os programas que controlam o fluxo de interação de uma forma bastante rígida. Os utilizadores esperam que o programa peça *inputs* para aí interagirem com o mesmo (controlo interno). Em termos conceptuais e usando pseudo-código, o fluxo de execução destes programas pode ser ilustrado da seguinte forma:

```
while(true)
  pedir input
  ler input
  analisa (e processa) input
  toma ação em conformidade
  (eventualmente gera saidas)
```

Um dos **maiores problemas** desta abordagem é que se torna **difícil modelar, do ponto de vista da utilização do programa, formas de captar diferentes ações do utilizador**. Dada a sua arquitetura, isso será apenas possível recorrendo a uma **estratificação sequencial das funcionalidades oferecidas**. I.e., o utilizador precisa normalmente percorrer um fluxo modelado por uma árvore até chegar à funcionalidade que pretende, nas folhas da mesma. Por outro lado, **esta abordagem permite sempre que o programa funcione em linha de comandos**, o que pode constituir uma vantagem em muitos casos e para muitos cenários de aplicação, embora não frequentemente para dispositivos móveis. **Alguns Sistemas Operativos** (SOs) funcionam apenas em linha de comandos (e.g., o *Internetwork Operating System* (IOS)), e **a própria *shell* do SO usa esta abordagem**.

Existem muitos exemplos de aplicações bastante complexas que usam este modelo de interação. O editor de texto `vi` concretiza um desses exemplos. **A estratificação é por vezes conseguida através da introdução de *modos***, que permitem que o utilizador **navegue para um determinado conjunto** de ações a partir de determinado

ponto de execução. O fluxo de execução desses programas pode ser representado, de uma forma geral, como se mostra a seguir:

```
while(true)
  label 1
  modo 1:
    while(true)
      pedir input
      ler input
      analisa (e processa) input
      toma ação em conformidade
      if( muda de modo )
        goto 1
      (eventualmente gera saidas)

  modo 2:
    while(true)
      pedir input
      ler input
      analisa (e processa) input
      toma ação em conformidade
      if( muda de modo )
        goto 1
      (eventualmente gera saidas)
  ...
```

Embora o modelo de interação descrito **escale para qualquer conjunto de ações** (**à custa de variáveis de estado e de alguma complexidade programática**), **não é muito intuitivo** e **dá azo a confusões e erros**. O editor de texto `vi`, por exemplo, tem dois modos diferentes: o de *introdução de texto* e o de *entrada de comandos* e, para um principiante, é comum a situação de estar a tentar introduzir texto no modo *entrada de comandos* ou vice-versa; enquanto que num ambiente gráfico essa confusão poderia nem existir. Para além disso, a implementação do modelo **requer** normalmente **a especificação de muitas variáveis de estado** que controlam o caminho percorrido na árvore. A própria forma de como a mudança de modo deve ser programada nem sempre é limpa, na medida em que pode não ser feita sempre da mesma forma (e.g., no `vi`, a mudança de modo para *inserção de comandos* é feita pressionando a tecla `Esc`, enquanto que a mudança para o modo de *inserção* é feita escrevendo `i` e pressionando *Enter*).

Alguns **dispositivos físicos** também usam esta abordagem por modos na sua interface do utilizador. Por exemplo, os **comandos de televisão, *boxes* e equipamentos multimédia** contêm frequentemente um botão de mudança de modo para cada um dos dispositivos suportados. Note que algumas interfaces de utilizador de aplicações móveis ainda usam esta abordagem, principalmente em dispositivos mais simples, sem ecrã tátil. Por exemplo, alguns telemóveis para idosos contêm funcionalidades bastante avançadas, mas a aplicação de marcação de números de telefone, uma vez acedida, fica a aguardar a inserção do número de telefone pretendido ou o seu cancelamento. No caso de um smartphone, a mesma aplicação pode simultaneamente aguardar que se pressione prolongadamente o ecrã para despoletar a opção de colar (*paste*) o que estiver no *clipboard*.

1.3 Interfaces de Utilizador Orientadas por Eventos
Event-Driven User Interfaces

Os desenvolvimentos tecnológicos que permitiram a **produção em massa e adoção de ecrãs táteis** constituem pontos chave na história das interfaces de utilizador, contribuindo ainda mais para a solidificação do modelo de interação orientado por eventos. Este modelo, utilizado na maior parte das aplicações móveis modernas, permite disponibilizar um **conjunto maior de funcionalidades em simultâneo e de forma intuitiva, correspondendo ainda a uma simplificação na estrutura do programa**. Neste caso, é **o programa que espera pelo utilizador** (controlo externo), nomeadamente que este lhe forneça um dos possíveis *inputs*.

A **manipulação** destas interfaces pode ser **feita de forma direta**, i.e., para cada ou alguns dos objetos mostrados ao utilizador. Tecnologicamente, tal é conseguido através de **mecanismos de comunicação entre objetos interativos e o sistema de entrada e saída** (e.g., um ecrã tátil[1]), concretizada por eventos:

> No contexto da informática e, particularmente, programação, **um evento corresponde a uma ação ou ocorrência que um programa foi capaz de detetar e tratar**. Um evento tem algumas propriedades que, em última análise, definem o seu tratamento por parte de um programa, nomeadamente **o tipo de acontecimento** (e.g., clique de um dos botões do rato, pressionar de uma tecla), **a posição do cursor ou tecla pressionada**, o **programa ao qual se destina** o evento, etc.

A rotina principal de captação de eventos é da responsabilidade do Sistema Operativo (SO), que a executa constantemente e **coleciona os eventos numa pilha *First In First Out*, assegurando que estes são tratados pela ordem que chegaram**. Note-se que alguns eventos podem ser descartados (e.g., um clique numa área do *desktop* que não tem nenhuma ação associada). De uma maneira abstrata, o modo de funcionamento desta rotina pode ser representada como se mostra a seguir:

```
while(true)
  espera por evento
  envia evento para lista de eventos
  analisa evento e envia para programa
  (se o programa estiver a dormir, acorda-o)
```

Ao receber um evento, o programa fá-lo **chegar à rotina de tratamento** a que corresponde:

[1] Note que um ecrã tátil é, simultaneamente, um dispositivo de entrada e saída.

```
while(true)
  espera por evento
  if( existir evento )
    recebe e analisa evento
    envia evento para rotina de tratamento
  else
    sleep
```

Note que, no caso da programação orientada por eventos, um programa pode entrar no estado de *sleep* quando não existem eventos, pelo que o Sistema Operativo pode atribuir tempo de processamento a outros programas.

A programação baseada por eventos é normalmente feita recorrendo àquele que é conhecido por ***modelo de delegação de eventos***, **baseado nas entidades *Controlos*, *Consumidores* e *Interfaces*** (de programação). Os ***controlos* serão a fonte do evento** (e.g., um botão), os ***Consumidores* (*Listeners*) correspondem às rotinas de tratamento desses eventos** (i.e., consomem – no sentido de tratarem – essa ocorrências) e as ***interfaces* são a forma normalizada de comunicação entre as duas entidades** antes referidas. Aquando da implementação, **o programador tem de definir os controlos e registar o conjunto de eventos que deseja escutar** para cada um deles, bem como **implementar a interface** (de programação) do evento que pretende escutar.

O código Java seguinte mostra um exemplo simples de como o parágrafo anterior se concretiza em termos de implementação.

```
ckb1 = (Checkbox) findViewById(R.id.chk1);
ckb2 = (Checkbox) findViewById(R.id.chk2);

ckb2.setOnClickListener(new OnClickListener(){
  @override
  public void onClick( View v ){
    if( (CheckBox v).isChecked() )
      ckb1.setEnabled(false);
  }
});
```

No trecho declaram-se dois *check buttons*, que estão definidos num ficheiro XML e que são automaticamente configurados através do método `findViewByID()`. De seguida, é invocado o método `setOnClickListener` que regista um novo *listener* para o `ckb2`, redefinindo imediatamente o *handler* para o evento clique no rato. Esta redefinição é feita através de uma classe anónima e re-escrita do método (da interface `OnClickListener`) *onClick()*.

O código incluído anteriormente pode parecer confuso a partir da quarta linha mas, basicamente, o que se está a fazer a partir desse ponto é condensar as seguintes atividades:

1. Invocar o método `setOnClickListener( p )` com um parâmetro, que tem de ser da classe que implementa a interface referida a seguir;
2. Inicializar um novo objeto da classe `OnClickListener`;
3. Aproveitar para redefinir, imediatamente, o método `onClick()` da interface.

A classe resultante é **conhecida**, no seio do jargão Java, como **classe anónima**. A definição de classes desta forma **é útil em várias situações**, nomeadamente **quando não faz sentido um determinado objeto existir quando o que o invoca não existe**, ou quando se quer **definir parte do comportamento da classe junto do local onde é declarada**.

Note-se que esta forma de programar e construir interfaces de utilizador é **extremamente útil para ambientes gráficos** já que, por um lado, **permite alguma abstração ao programador**, que **define apenas o que deve ser feito para os eventos que lhe interessam** e, por outro, permite **aproveitar o espaço gráfico da aplicação** para transmitir as suas funcionalidades de forma efetiva. As *frameworks* ou **bibliotecas existentes fornecem já a maior parte das classes e métodos necessários**, ficando a faltar a implementação de alguns desses métodos. A **captura e reencaminhamento dos eventos é feito de forma transparente para o programador**, que não precisa de se preocupar com esses detalhes.

É claro que a programação orientada por eventos assenta em vários **pressupostos**. Um dos mais importantes é que **a interface gráfica é composta por vários objetos interativos e elementos cuja localização é conhecida**, e que podem ser **estruturados hierarquicamente em árvore** (e.g., no universo Android é mencionada uma *User Interface Layout*[2], enquanto que no universo iOS o mesmo conceito é designado por *View Hierarchy*[3]). A estrutura hierarquica surge natural à abordagem, já que cada objeto interativo irá pertencer a determinada aplicação, que por sua vez corre dentro de uma janela, etc. A profundidade desta hierarquia não costuma ser superior a 5 ou 6 níveis, caso contrário também se torna difícil de gerir e desenhar, para além de **afetar negativamente a performance da mesma**.

[2] `http://developer.android.com/guide/topics/ui/overview.html`

[3] Mais info em `https://developer.apple.com/library/content/documentation/WindowsViews/Conceptual/ViewPG_iPhoneOS/WindowsandViews/WindowsandViews.html`

A imagem anterior ilustra uma hierarquia de objetos interativos para uma aplicação móvel Android™. A decomposição apresentada pode não ser a que foi realmente usada na aplicação, mas será uma aproximação igualmente possível. A hierarquia tem 4 níveis, sendo que na raiz está o ecrã, o maior contentor possível para os diversos objetos. No primeiro nível está a barra de notificações e a atividade em exibição da aplicação Android™. Por sua vez, a atividade é composta por uma barra de título, uma grelha e dois botões. A grelha e a barra de título são dois contentores, que contêm etiquetas de texto e botões.

Apesar da ilustração ser feita para uma aplicação Android™, **a explicação seria semelhante** para a esmagadora maioria das aplicações com interface gráficas, nomeadamente aquelas implementadas para Microsoft Windows, Linux (ambientes gráficos como o Gnome), iOS ou MAC OS.

Na documentação da especialidade, **os objetos interativos** são sobretudo conhecidos pela designação inglesa ***widgets* ou controlos**. Por serem usados para criar interfaces para humanos, estes objetos têm **uma representação gráfica semelhante ao que conhecemos de outros dispositivos usados no quotidiano**, nomeadamente **botões de pressão ou de on/off**, que mudam de aparência aquando da interação (e.g., ao premir um botão no ecrã, a imagem muda para simular essa ação). **Estes objetos são reutilizáveis e disponibilizados**, tipicamente, **numa biblioteca ou *framework*** preparada para o efeito. No caso do Android™, por exemplo, o conjunto de classes que disponibilizam as objetos interativos estão em `android.widgets`[4] e `android.view`[5][6] (entre outras), enquanto que nas versões recentes do iOS é a *framework* UIKit[7] que as providencia.

O **comportamento** das aplicações cuja programação é orientada por eventos **é, portanto, inteiramente definido pelas rotinas de tratamento desses eventos**. Programaticamente, são essas rotinas que definem o fluxo, a lógica e as funcionalidades das aplicações. O desenvolvimento da aplicação pode, por isso, **começar pelo planeamento da interface de utilizador e evoluir para a implementação das rotinas de tratamento**. Se revisitar o código Java incluído anteriormente, irá reparar que, no trecho, começa-se por definir os objetos interativos para depois definir o comportamento.

Existem normalmente **dois tipos genéricos** de elementos úteis para o desenho de interfaces de utilizador orientadas por eventos. Os **objetos interativos de entrada ou saída** e **os contentores**. Os **contentores são elementos que podem conter outros contentores ou objetos interativos** como botões ou caixas de texto. Os contentores **permitem organizar o aspeto gráfico da aplicação** (e.g., oferecendo funcionalidades de alinhamento vertical ou horizontal, posicionamento através de deslocamento, etc.).

Os eventos gerados dependem normalmente do contexto do objeto interativo e aqueles que vão ser efetivamente **tratados dependem dos objetivos da aplicação**. Por exemplo, uma aplicação que pretenda imitar

[4] `http://developer.android.com/reference/android/widget/package-summary.html`

[5] `http://developer.android.com/reference/android/view/package-summary.html`

[6] Na verdade, as classes de `android.widgets` são sub-classes das classes de `android.view`.

[7] Mais em `https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIKit_Framework/index.html`

um piano pode produzir sons logo que um botão é premido, enquanto que uma que pretenda imitar uma guitarra só produz o som após o botão ser libertado. Alguns dos eventos mais comuns (bem como os seus contextos) usados para construir aplicações móveis podem ser enumerados como se segue:

1. **Botões** de pressão com rato, toque ou caneta, podem gerar eventos de clicados, premidos ou libertados e duplamente clicados;
2. **Teclado** pode gerar eventos de tecla premida ou libertada;
3. **Movimentação do rato ou cursor** pode gerar eventos de entrada e saída de regiões;
4. **Os controlos das janelas ou da aplicação** podem gerar eventos de fecho, minimização, redimensionamento, etc.

É comum que os **objetos interativos tenham acesso aos métodos de outros objetos no mesmo nível hierárquico** ou acima. No exemplo com código Java incluído anteriormente, um dos objetos interativos podia alterar as propriedades do outro invocando o método `setEnabled()`. A **comunicação entre objetos interativos** pode, assim, ser feita de **3 formas diferentes**:

1. Através da **manipulação direta de propriedades de outros objetos** (exemplo anterior);
2. **Avisando o elemento imediatamente acima na hierarquia** acerca das alterações a fazer, sendo que este terá acesso a outros elementos do mesmo grau hierárquico;
3. **Gerando outros eventos** (pseudo-eventos, já que não são exatamente gerados por humanos), que são capturados e tratados pelas rotinas dos consumidores registadas pelo objeto interativo destino.

Ainda tomando o código incluído antes como referência, deve ser notado que **o *handler*** (a rotina de tratamento) **recebe o próprio objeto interativo como parâmetro de entrada**. O facto é que quando um evento é gerado, este segue para a pilha de eventos até ser brevemente processado e despachado para o *consumidor* registado por esse objeto, onde poderá dar jeito verificar o estado da fonte do evento.

> O **objeto interativo onde o evento é gerado** é designado **por fonte**.

Para terminar, **interessa compreender bem o fluxo de execução de uma aplicação orientada por eventos**. Para isso, considere o trecho de código seguinte:

```
import android.view.*;
import android.widgets.*;
...
btn = (Button) findViewById(R.id.btn);

btn.setOnClickListener(new OnClickListener(){
  @override
  public void onClick( View v ){
    Context context = getApplicationContext();
    Toast toast = Toast.makeText(context, "Clicou no
        botao!",Toast.LENGTH_SHORT);
    toast.show();
  }
});
...
```

O código anterior implementa uma pequena parte de uma aplicação Android™ em que é **definido um botão e registado um novo *consumidor* para o evento *clique***. Sempre que um utilizador clica no botão (com o rato ou com o teclado):

1. **É capturado** um evento do tipo *onClick* no botão `btn`;
2. O evento do tipo *onClick* **é enviado** para o *consumidor* que lhe corresponde;
3. **É executado** o código que está definido no objeto `OnClickListener`, dentro do objeto `btn`, no método `onClick`. No caso específico apresentado, seria mostrada uma mensagem numa *widget* `Toast` flutuante com o texto `Clicou no botao!` durante 2 segundos (`Toast.LENGTH_SHORT`).

2 Modelo-Visão-Controlador

Model-View-Controller

2.1 Introdução

Introdução

Como implicitamente sugerido em cima, é útil pensar numa aplicação em termos de eventos e de interface de utilizador, tal como é útil poder separar a semântica da aplicação da sua apresentação.

> De um modo simplista e geral, a **arquitetura Modelo-Visão-Controlador** (da designação inglesa *Model-View-Controller* (**MVC**)) é **um modelo de arquitetura para desenvolvimento de software que separa a lógica da aplicação, a representação da informação e a interação do utilizador** através da definição de 3 componentes: **o modelo, o controlador e a visão** (discutidos melhor em baixo).

Esta arquitetura de desenvolvimento de software foi criada por Trygve Reenskaug **nos anos 70**, enquanto este trabalhava com *Smalltalk* para a empresa Xerox Parc. **O objetivo** de estruturar a aplicação de acordo com o MVC é o de **favorecer a sua escalabilidade e manutenção**, bem como **a portabilidade e a reutilização de código**. A estrutura definida **permite também um maior grau de abstração** por parte de cada um dos elementos da equipa de implementação e desenho. A MVC é **muito**

popular na indústria de desenvolvimento de aplicações Web e móveis, sendo largamente sugerida por gigantes na área. Por exemplo, a Microsoft fornece a *framework* ASP.NET MVC dentro da sua solução .NET[8], enquanto que Apple enfatiza o beneficio da sua utilização na implementação de aplicações Cocoa[9].

2.2 Modelo
Model

O ***Modelo* é o componente central do MVC**. **Encapsula o estado interno** e consiste na **implementação dos objetos, métodos ou rotinas que capturam o comportamento base da aplicação**. É o modelo que processa os dados ou muda o seu estado. Nas especificações mais conservadores, **só pode interagir com o componente controlador**, sendo **completamente independente da interface** do utilizador.

Numa implementação de uma aplicação móvel, pode-se pensar no *Modelo* como o conjunto de classes que concretizam os seus objetivos principais. Fazendo uso do que é discutido antes nesta aula, é no *Modelo* que concetualmente devem estar **as implementações dos métodos que, de forma concisa, segura e uniformizada** alteram os dados/informação interna da aplicação. De uma forma geral, é o modelo que contém as classes e métodos para **aceder e manipular bases de dados** ou memória persistente, bem como toda a informação que pode ser exibida ao utilizador através de uma visão. No MVC, **um utilizador nunca interage diretamente com o *Modelo***.

A implementação das classes pertencentes ao *Modelo* da aplicação devem ser **o menos específicas possível em relação à representação dos dados** ou interação com os mesmos, para favorecer a portabilidade do código. A sua **interface de programação deve, contudo, ser clara e consistente** ao longo do período de vida da aplicação (já que isso garante que o controlador pode comunicar corretamente com o modelo).

2.3 Visão
View

No MVC, **uma *Visão* consiste numa possível representação dos dados da aplicação**. É o componente *Visão* que se **definem as Interfaces de Utilizador**. Note-se que podem existir **várias representações para o mesmo conjunto de dados, daí a importância em separar este componente da lógica aplicacional**. Por um lado, permite que a equipa de desenvolvimento seja repartida pelas várias áreas, por outro, permite que a **interface do utilizador possa ser desenvolvida em paralelo e melhorada ao longo do tempo**.

[8]Ver `http://msdn.microsoft.com/en-us/library/dd381412%28v=vs.108%29.aspx`.

[9]Ver `https://developer.apple.com/library/content/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html`.

A separação entre *Modelo* e *Visão* é cabal em aplicações Web. Por exemplo, é comum a implementação destas aplicações utilizando a combinação de tecnologias *HyperText Markup Language* (HTML), *Cascade Style-Sheet* (CSS) e *Hypertext PreProcessor* (PHP) ou JavaScript. Neste caso, a especificação da apresentação da informação é sobretudo definida nos ficheiros HTML e CSS, enquanto que a lógica aplicacional estará em ficheiros PHP. Este facto permite que as aplicações Web possam aparecer de forma diferente para diferentes utilizadores (e.g., o Gmail no *browser*), à custa apenas de um estilo CSS diferente.

No caso das aplicações móveis, essa separação também acontece de forma vincada. Na **plataforma Android™**, por exemplo, é permitido que **a interface do utilizador seja definida em ficheiros XML** completamente independentes do código da aplicação escrito em Java. Estes ficheiros estão normalmente na diretoria `app/src/main/res/` (de *resources*) e o seu nome termina tipicamente com a palavra *layout*. O código da aplicação (que concretiza o *Modelo* e o *Controlador*) encontra-se na pasta `app/src/main/java/`.

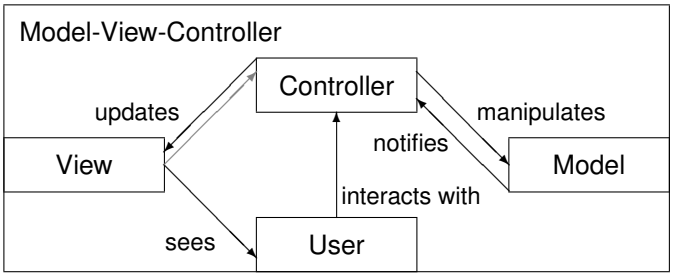
2.4 Controlador
Controller

O ***Controlador* é o intermediário entre o *Modelo*, as *Visões* e o utilizador**. Na verdade, é possível encontrar referências em que se define o *Controlador* como intermediário dos dois outros componentes e não do utilizador, sendo que este apenas interage com o componente *Visão*, onde são gerados os eventos. Contudo, apesar dos eventos poderem ser gerados sobre objetos da *Visão*, **é no *Controlador* que estão as rotinas de captura e reencaminhamento desses eventos**, pelo que também é aceite a figura estática das perspetivas. **São procedimentos do *Controlador* que atualizam os objetos da *Visão* e que despoletam mudanças no estado da aplicação através do envio de eventos para o *Modelo***. Para além disso, é da responsabilidade do *Controlador* a coordenação de tarefas de uma aplicação ou a gestão do ciclo de vida de objetos. As **rotinas de tratamento de eventos** incluem-se, portanto, no componente Controlador. Estas rotinas invocam métodos do Modelo, que são os que melhor sabem manipular os dados. É boa prática separar o código ou classes que lidam com o modelo (e.g., manipulam bases de dados) do código dos controladores (e.g., podem-se colocar a(s) classe(s) de manipulação de bases de dados num ficheiro, e definir a interação com o utilizador noutros).

2.5 Comunicação entre Componentes
Communication Between Components

É possível encontrar diferentes diagramas de comunicação entre os 3 componentes e o utilizador na literatura e *online*. Alguns desses diagramas são diferentes ao ponto de se contradizerem. O diagrama seguinte é, por isso,

uma das abordagens mais consensuais à forma de comunicação entre os 3 componentes descritos antes e o utilizador (note que, na figura, as setas $\rightarrow$ denotam o sentido das comunicações entre componentes). O diagrama tem o componente ***Controlador* ao centro**,



evidenciando o seu papel na comunicação. O **utilizador interage com a aplicação através de uma ou mais *Visões* e do *Controlador***. As *Visões* mostram a informação do *Modelo* ao utilizador, que interage com a interface para manipular essa informação. Essa **interação gera eventos**, capturados pelo *Controlador*, que os processa e envia para as rotinas de tratamento definidas no *Modelo*. O ***Controlador* pode imediatamente atualizar uma ou mais *Visões* aquando da receção de um evento**, ou esperar pelos seus efeitos no *Modelo*. Caso o estado seja alterado, é gerada uma notificação para o *Controlador*, que este usa para despoletar a atualização de uma ou mais *Visões*.

Note que, visto **não existir comunicação direta entre o *Modelo* e as *Visões***, qualquer alteração que precise ser refletida na interface tem de ser comunicada ao *Controlador*. A **notificação contém os dados do *Modelo* necessários à alteração**. A expressão *manipulates*, que dita o fluxo de informação entre o *Controlador* e o *Modelo*, também pode significar a transmissão de dados entre os dois componentes. Como exemplo, considere que um utilizador acabou de inserir uma expressão a ser pesquisada na base de dados, pressionando *Enter* de seguida. O *Controlador* captura o evento de tecla premida e o texto inserido, enviando-o para o *Modelo*. A rotina de processamento (definida no modelo) acede à base de dados, faz a pesquisa, e retorna o novo estado ao *Controlador*, que atualiza a *Visão* respetiva, chegando o resultado aos olhos do utilizador.

O exemplo incluído no último parágrafo pode levar à discussão da **possível existência de comunicação na direção de *Visões* para o *Controlador***, já que o evento que inicialmente despoletava a rotina de pesquisa na base de dados era o clique na tecla *Enter*, e não a inserção do texto. Neste caso, o *Modelo* teria de notificar o *Controlador* que havia informação em falta, e este teria de ir buscar essa informação ao objeto respetivo na *Visão* (daí a seta em cinza claro na figura).

Nota: o conteúdo exposto na aula e aqui contido não é (nem deve ser considerado) suficiente para total entendimento do conteúdo programático desta unidade curricular e deve ser complementado com algum empenho e investigação pessoal.