# Tema: III.
# Programación dirigida por eventos y ejecución diferida de código

Herramientas Avanzadas para el Desarrollo de Aplicaciones

Departamento de Lenguajes y Sistemas Informáticos
Universidad de Alicante





## Contenido





## Preliminares

- En términos de la estructura y la ejecución de una aplicación representa lo opuesto a lo que hemos hecho hasta ahora: *programación secuencial*.
- La manera en la que escribimos el código y la forma en la que se ejecuta éste está determinada por los sucesos (*eventos*) que ocurren como consecuencia de la interacción con el mundo exterior.
- Podemos afirmar que representa un nuevo *paradigma de programación*, en el que todo gira alrededor de los eventos[1].

[1] Cambios significativos en el estado de un programa.



## Programación secuencial vs. dirigida por eventos I

- En la programación secuencial le decimos al usuario lo que puede hacer a continuación, desde el principio al final del programa.
- El tipo de código que escribimos es como éste:

```
1  repetir
     presentar_menu ();
3    opc = leer_opcion ();
     ...
5    si (opc == 1) entonces accion1 ();
     si (opc == 2) entonces accion2 ();
7    ...
   hasta terminar
```

- En la programación dirigida por eventos indicamos:
  - ¿Qué cosas -*eventos*- pueden ocurrir?
  - Lo que hay que hacer cuando ocurran

```
   son_eventos (ev1, ev2, ev3...);
2  ...
   cuando_ocurra ( ev1, accion1 );
4  cuando_ocurra ( ev2, accion2 );
   ...
6  repetir
     ...
8  hasta terminar
```

## Programación secuencial vs. dirigida por eventos II

- A partir de este punto los eventos pueden ocurrir en cualquier momento y marcan la ejecución del programa.
- Aunque no lo parezca plantean un problema serio: el flujo de la ejecución del programa escapa al programador.
- El usuario (como fuente generadora de eventos) toma el control sobre la aplicación.
- Esto implica tener que llevar cuidado con el diseño de la aplicación teniendo en cuenta que el orden de ejecución del código no lo marca el programador y, además, puede ser distinto cada vez.



## Esqueleto de una aplicación dirigida por eventos I

- Al principio de la misma llevamos a cabo una iniciación de todo el sistema de eventos.
- Se definen todos los eventos que pueden ocurrir.
- Se prepara el generador o generadores de estos eventos.
- Se indica qué código se ejecutará en respuesta a un evento producido -*ejecución diferida de código*-.
- Se espera a que se vayan produciendo los eventos.



## Esqueleto de una aplicación dirigida por eventos II

- Una vez producidos son detectados por el 'dispatcher' o planificador de eventos, el cual se encarga de invocar el código que previamente hemos dicho que debía ejecutarse.
- Todo esto se realiza de forma ininterrumpida hasta que finaliza la aplicación.
- A esta ejecución ininterrumpida es a lo que se conoce como **el bucle de espera de eventos**.
- Las aplicaciones con un interfaz gráfico de usuario siguen este esquema de programación que acabamos de comentar.
- Podemos verlo de forma gráfica en el siguiente diagrama:



## Diagrama de una aplicación dirigida por eventos

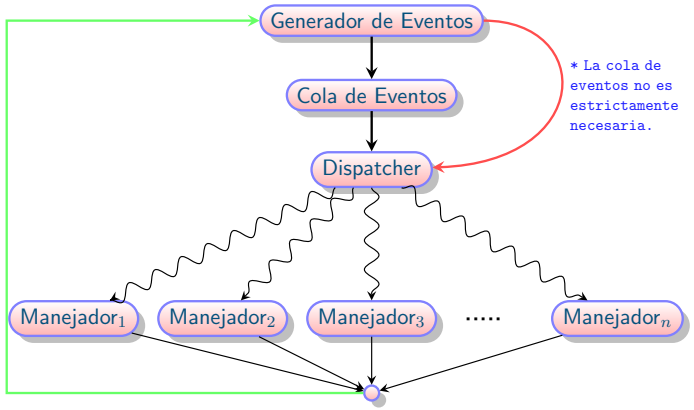


Figura: Diagrama de una aplicación dirigida por eventos.

## Y todo esto en Vala...

- Los eventos en Vala se denominan **señales**.
- Son equivalentes a los *eventos* en C# o a los *listeners* en Java.
- Las *señales* son un mecanismo proporcionado por la clase GLib.Object.
- Si queremos que una clase nuestra pueda emitir *señales* -emitir eventos-, debe derivar directa o indirectamente de GLib.Object.
- Una *señal* se define como un miembro de una clase y se parece a un método sin cuerpo.
- Sólo vemos su signatura y nombre precedidos por la palabra reservada '`signal`' y el modificador de acceso.
- A una **señal** le '*conectamos*' tantos métodos o funciones como queramos que se ejecuten cuando se emita esa señal.



## Ejemplo de señal en Vala I

```
    void f1 (int n) { ... }
2   void f2 (int n) { ... }

4   public class Test : GLib.Object {
      public signal void sig_1(int a); // <=== SIGNAL
6
      public static int main(string[] args) {
8       Test t1 = new Test();

10      t1.sig_1.connect (f1);
        t1.sig_1.connect (f2);
12      ...             // Conexion de la senyal
                        // con el codigo a ejecutar
14
        t1.sig_1 (2);   // Emision de la senyal
16                      // Equivale a llamar a: f1(2); f2(2)

18      return 0;
      }
20  }
```



## Ejemplo de señal en Vala IIa

```
    class Player : GLib.Object {
2     public Player.with_name (string n) {
        name = n;
4       tries = 0;
        new_try.connect (record_try);
6     }

8     public signal void new_try ();

10    public int get_tries () { return tries; }
      public void make_try () { tries++; new_try(); }
12    private void record_try () {
        stdout.printf ("the player <%s> tried one more time\n",
14                     name);
      }
16    public string get_name () { return name; }
      //-- Datos ----------------------------------------------
18    public int tries;
      private string name = "";
20  }

22  void player_try (Player p) {
      stdout.printf ("The player [%s] tried one more time\n",
24                   p.get_name());
    }
```



## Ejemplo de señal en Vala IIb

```
1   void main () {
      const int max_tries = 2;
3     var juan = new Player.with_name ("Juan");
      var pedro = new Player.with_name ("Pedro");
5
      juan.new_try.connect (player_try);
7     pedro.new_try.connect (player_try);

9     for (int i = 0; i < max_tries; i++) {
        juan.make_try ();
11      pedro.make_try ();
      }
13  }
```

## Ejemplo de señal en Vala IIc

```
the player <Juan> tried one more time
The player [Juan] tried one more time
the player <Pedro> tried one more time
The player [Pedro] tried one more time
the player <Juan> tried one more time
The player [Juan] tried one more time
the player <Pedro> tried one more time
The player [Pedro] tried one more time
```



## Ejercicio en clase. Preparación para la ejecución diferida de código (I)

- La ejecución diferida de código tiene sus orígenes en el concepto de Callback .
- En Lenguaje 'C' un Callback no es más que un puntero a una función.
- Vamos a comenzar a preparar en clase un ejercicio que haremos en las prácticas relacionadas con la *ejecución diferida de código*.



## Ejercicio en clase. Preparación para la ejecución diferidad de código (II)

- En la biblioteca estándar de 'C' (`#include <stdlib.h>`) disponemos de la función "`qsort - ordena un vector`". El prototipo de la misma es:

```
1    void qsort(void  *base, size_t nmemb, size_t size,
                int(*compar)(const void *, const void *));
```

- Identifica cada uno de sus parámetros. Trata de entender qué es '`compar`'. Propón un posible valor para ese parámetro.
- Cuando tengas hecho todo este estudio trata de crear un programa ejecutable mínimo que te sirva para comprobar cómo funciona '`qsort`'.



## Ejecución diferida de código. Preliminares (I)

- La constituyen los llamados *Callbacks* en C, son elementos de bajo nivel, ya los conocemos[2]...

```
   int int_cmp (const void* pe1, const void* pe2) {
2    int e1 = *((const int*) pe1);
     int e2 = *((const int*) pe2);
4
     return e1-e2;
6  }

8  int main () {
     int v[4] = {4,0,-1,7};
10
     for (int i = 0; i < 4; i++)
12     printf("v[%d]=%d\n", i, v[i]);

14   qsort (v, 4, sizeof(int), int_cmp);
     printf("\n");                \--v--/
16                                   \-----Ejecucion diferida
     for (int i = 0; i < 4; i++)
18     printf("v[%d]=%d\n", i, v[i]);
     return 0;
20 }
```

[2] Punteros a funciones.

## Ejecución diferida de código. Preliminares (II). Ejercicio en clase

- También sabemos que `Vala` nos permite hacerlo mejor que '**C**': '`signal`' + '`connect`'.
- Pero también sabemos que es *compatible* con bibliotecas de '**C**'...
- Echa un vistazo a la biblioteca de compatibilidad con *Posix* de '**C**': vala posix , busca en ella la referencia a la función qsort .
- Fijate en la declaración del tipo '`compar_fn_t`'.
- Ejercicio: Trabajando en grupos como en el ejercicio anterior, reescribe ese ejemplo hecho en **C** ahora con **Vala** y usando la capa de compatibilidad **Posix**. *15min.*



## Ejecución diferida de código. Preliminares (III)

- Otros lenguajes de programación sin llegar a tener un soporte sintáctico para esto ofrecen alternativas de *más alto nivel*.
- Vamos a ver el caso de tres *implementaciones distintas* para el caso de **C++**:
  - libsigc++ : Empleada por la biblioteca gtkmm .
  - signal/slot : Empleada por la biblioteca Qt .
  - boost::signals : Implementación *parecida* a `libsigc++` pero candidata a convertirse en el soporte estándar de ejecución de código diferido en `C++` por su pertenencia al proyecto boost .
- En la documentación de cada una de estas tres implementaciones de ejecución de código diferido tenemos ejemplos sencillos, vamos a ver un ejemplo de cada uno de ellos. *10min. cada uno*



## El principio de HollyWood (I)

- Es muy sencillo: **No nos llame...ya le llamaremos**.
- Se emplea sobre todo cuando se trabaja con '`frameworks`'.
- El flujo de trabajo se parece a esto:
  1. En el caso de trabajar con un `framework`[3] implementamos un interfaz y en el caso más sencillo escribimos el código a ejecutar más adelante.
  2. Nos registramos... es decir, indicamos de algún modo cuál es el código a ejecutar posteriormente.
  3. Esperamos a que se llame -al código registrado previamente- cuando le 'toque': *No nos llame... ya le llamaremos*.
- El programador ya no 'dicta' el flujo de control de la aplicación, sino que son los eventos producidos los que lo hacen.

[3] Se estudian en la asignatura Programación-3.



## El principio de HollyWood (II)

- Puedes consultar más información sobre él aquí .
- Si quieres ampliar más tus conocimientos sobre él debes saber que a este principio también se le conoce por otros nombres:
  1. Inversión de control ( IoC ).
  2. Inyección de dependencias ( DI ).
- Puedes ampliar más información sobre estas técnicas de programación en asignaturas como Programación-3.

## Y todo esto en Vala...

- Conocemos lo esencial ya del principo de este tema: `señales` y `conexiones con las mismas`:

```
    public class Test : GLib.Object {
2
      public signal void sig_1(int a);
4     public void m (int a) { stdout.printf("%d\n", a); }

6     public static int main(string[] args) {
        Test t1 = new Test();
8
        t1.sig_1.connect(t1.m);
10
        t1.sig_1(5);
12
        return 0;
14    }
    }
```



## Señales en Vala. Lo básico (I)

- Recuerda que para que una clase pueda emitir señales debe derivar directa o indirectamente de la clase `GLib.Object`.
- Una señal puede tener ninguno, uno o más parámetros.
- La signatura de la función que conectamos a una señal debe coincidir con la de la propia señal...
- ...salvo que en Vala se permite: ¡Importante!
  1. que omitamos tantos parámetros desde el final como queramos...
  2. o que proporcionemos el 'originador' de la señal como primer parámetro del `callback`.



## Señales en Vala. Lo básico (II)

- Por ejemplo, para la señal '`Foo.some_event`' serían señales válidas todas estas:

```
1   public class Foo : Object {
      public signal void some_event (int x, int y, double z);
3     // ...
    }
5   ...// las siguientes funciones pueden conectarse a 'some_event'
    void on_some_event ()
7   void on_some_event (int x)
    void on_some_event (int x, int y)
9   void on_some_event (int x, int y, double z)
    void on_some_event (Foo source, int x, int y, double z)
```

- Los nombres de los parámetros son libres.
- Especificar el origen de la señal ('`source`') nos puede servir para diferenciar si conectamos la misma funcion a la misma señal para diferentes instancias del mismo tipo.



## Señales y funciones Lambda ($\lambda$)

- A una señal podemos conectarle una función definida en línea, una función lambda (función-$\lambda$).
- También se las llama funciones anónimas ya que no tienen nombre.
- El primer ejemplo que hemos visto en la página 68 podría reescribirse así usando funciones-$\lambda$:

```
    public class Test : GLib.Object {
2     public signal void sig_1(int a);
      public static int main(string[] args) {
4       Test t1 = new Test();

6       t1.sig_1.connect( (t, a) => {stdout.printf("%d\n", a);} );
        t1.sig_1(5);
8
        return 0;
10    }
    }
```

- Pregunta: ¿Qué son '`t`' y '`a`'? ¿Qué representa cada uno?

## Desconexión de señales en Vala (I)

- Es posible que a partir de un momento no nos interese que un determinado callback se invoque cuando se emita la señal a la que estaba conectado.
- Para conseguirlo debemos '*desconectarlo*' de la señal. Es el proceso inverso a la conexión.
- Podemos hacerlo de varias maneras, el caso más sencillo es usar el método '`disconnect`', el cual usa esta notación:

```
1   foo.some_event.connect (on_some_event);     // Conexion...
    foo.some_event.disconnect (on_some_event); // Desconexion
```

- Pero ¿qué ocurre si lo que habíamos conectado era una función-$\lambda$?..recuerda que no tienen nombre...



## Desconexión de señales en Vala (II)

- El *truco* está en saber que el método `connect` devuelve un valor de tipo '*entero largo sin signo*': `ulong`.
- Por tanto, cuando conectamos una función-$\lambda$ que pensemos desconectar, debemos guardar el valor que devuelve la conexión, por ejemplo:

```
    ulong sig_id = foo.some_event.connect (() => { /* ... */ });
2   foo.some_event.disconnect (sig_id);
```



## Señales con enlace dinámico en Vala (I)

- Una señal puede ser redefinida en una clase derivada...como un método normal y corriente...
- Para ello debe tener *enlace dinámico*, o lo que es lo mismo, ser declara '`virtual`'.
- Cuando una señal se declara `virtual` puede tener una implementación de un `callback` por defecto, además de que ésta implementación puede ser redefinida en clases derivadas, así:

```
    class Demo : Object {
2     public virtual signal void sig () {
        stdout.printf ("callback por defecto\n");
4     }
    }
6
    class Sub : Demo {
8     public override void sig () {
        stdout.printf ("Reemplazo del callback por defecto\n");
10    }
    }
```



## Señales con enlace dinámico en Vala (II)

- A una señal con un `callback` por defecto ¿se le pueden conectar más callbacks?..**sí**.
- IMPORTANTE: Se ejecutan **antes** que el `callback` por defecto.
- Existe la posibilidad de conectar `callbacks` que se ejecutan después del `callback` por defecto, eso se hace con el método `connect_after`:

```
1   void main () {
      var demo = new Demo ();
3     demo.sig.connect (() => stdout.printf ("antes\n"));
      demo.sig.connect_after (() => stdout.printf ("despues\n"));
5     demo.sig (); // emitimos la senyal
    }
7
    // Resultado:
9   // antes
    // callback por defecto
11  // despues
```

# Ejercicio: La vivienda domótica

Trabajando en grupos vamos a escribir el código para una aplicación que simule -*a modo de ejemplo*- una vivienda domótica. Esta vivienda o *casa* consta de:

- Varias `habitaciones`, cada una de las cuales dispone de...
  - `Persianas`, pueden estar `subidas` o `bajadas`. Estamos interesados en saber cuando se suben.
  - `Puertas`, pueden estar `abiertas` o `cerradas`. Estamos interesados en saber cuando se abren.
  - `Luces`, pueden estar `encendidas` o `apagadas`. Estamos interesados en saber cuando se encienden.
- Crea las clases/interfaces que consideres necesarios y un sencillo programa de prueba que cree una casa, con varias habitaciones y cada una de ellas con varias puertas/ventanas/luces.
- Comprueba que, efectivamente, cuando alguien abre una puerta, enciende una luz o sube una persiana, el sistema domótico nos avisa de ello.