Unit: III.
Event driven programming

Herramientas Avanzadas para el Desarrollo de Aplicaciones

Languages and computing systems
Universidad de Alicante

Content

Background

- In terms of structure and execution of an application it represents the opposite of what we have done so far: *sequential programming*.
- The way in which we write code and the way it is run it is determined by the *events* that occur as a consequence of the interaction with the outside world.
- We can say that it represents a new *programming paradigm*, in which everything goes around the events[1].

[1] Significant modifications in the state of a program.

Sequential vs. event driven programming I

- In sequential programming we tell the user what to do next, from the beginning until the end of the program.
- The type of code we write is such as:

```
repeat
  present_menu ();
  opt = read_option ();
  ...
  if (opt == 1) then action1 ();
  if (opt == 2) then action2 ();
  ...
until finished
```

- In event driven programming we indicate:
  - Which things -*events*- can occur?
  - What has to be done when they occur

```
are_events (ev1, ev2, ev3...);
...
when_occurs ( ev1, action1 );
when_occurs ( ev2, action2 );
...
repeat
  ...
until finished
```

Sequential vs. event driven programming II

- From this point events can occur at any time and they mark the execution of the program.
- Although it does not seem so, they cause a serious problem: the flow of the execution of the program escapes to the programmer.
- The user (as the source that generates events) takes the control of the application.
- This implies being careful with the design of the application, taking into account that the order of the code execution is not marked by the programmer, and it can be different each time.

Skeleton of an event driven application I

- At the beginning of it all the event system is initiated.
- We define the events that can occur.
- We prepare the generator(s) of these events.
- We indicate the code to be executed to give answer to a triggered event -*deferred code execution*-.
- The events are being triggered.

Skeleton of an event driven application II

- Once triggered, the events are detected by the 'dispatcher' or event planner, which is in charge of invoking the code that we have previously indicated that has to be run.
- All these is done in a continuous way until the application is finished.
- This uninterrupted execution is known as **the waiting event loop**.
- The applications with a graphical user interface follow this programming schema.
- We can see it graphically in the next diagram:

Diagram of an event driven application

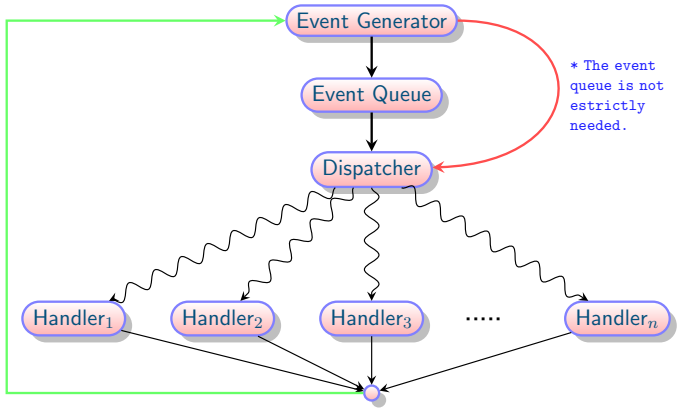

Figure: Diagram of an event driven application.

In Vala...

- The events in Vala are called **signals**.
- They are equivalent to the *events* in C# or to the *listeners* in Java.
- The *signals* are a mechanism provided by the GLib.Object class.
- If we want that a class can trigger *signals* -trigger events-, it has to be derived directly or indirectly from GLib.Object.
- A *signal* is defined as a member of a class and it seems a method without body.
- We only see its signature and name preceded by the keyword '`signal`' and the access modifier.
- We can '*connect*' to a **signal** so many methods or functions as we want to be executed when that signal is triggered.

Example of signal using 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      ...             // Signal conexion
                        // with the code to be run
14                      // We will see it in the next
                        // unit.
16
        t1.sig_1 (2);   // Signal emission
18                      // Equivalent to call: f1(2); f2(2)

20      return 0;
      }
22  }
```


Example of signal using Vala IIa

```
    class Player  {
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());
    }
```


Example of signal using 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  }
```

Example of signal using 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
```


Ejercise at class. Preparation for the deferred code execution (I)

- The deferred code execution has its origin in the concept of Callback .
- In 'C' language, a Callback is a pointer to a function.
- We are going to start preparing at class an exercise that we will do in the lab session of the unit of *deferred code execution*.

Exercise at class. Preparation for the deferred code execution (II)

- In the 'C' standard library (`#include <stdlib.h>`) we have the function "`qsort - sorts a vector`". The prototype of it is:

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

- Identify each of its parameters. Try to understand what '`compar`' is. Propose a possible value for that parameter.
- When you have done all this study, try to create a minim executable program that allows you to check how it works. '`qsort`'.

Deferred code execution. Basics (I)

- Callbacks in C, low level, we already know them...[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");                  \--*--/
16                                       \-----Deferred execution
      for (int i = 0; i < 4; i++)
18      printf("v[%d]=%d\n", i, v[i]);
      return 0;
20  }
```

[2] Function pointers.

Deferred code execution. Basics (II). Exercise at class

- We also know that `Vala` allows us to improve the '**C**' solution: '`signal`' + '`connect`'.
- But we also know that Vala is *compatible* with '**C**' libraries...
- Please, have a look to the '**C**' compatibility library with *Posix*: vala posix , look for the reference to the function qsort .
- Look to the declaration of the type '`compar_fn_t`'.
- Exercise: Working in groups like in the previous unit, rewrite the previous code done in **C** using **Vala** and using the compatibility layer **Posix**. *15min.*

Deferred code execution. Basics (III)

- Other programming languages do not have a syntactic support for callbacks but they offer other *higher level* alternatives.
- Let us see three *different implementations* using **C++**:
  - libsigc++ : Used by the library gtkmm .
  - signal/slot : used by the library Qt .
  - boost::signals : Implementation *similar* to `libsigc++` but the preferred for being the standard support for deferred code execution in `C++` because is used in the project boost .
- In the documentation of each of these implementations of deferred code execution we have simple examples, let's see an example of each of them. *10min. each*

The HollyWood principle(I)

- It is very easy: **Don't call us... we will call you**.
- It is mainly used when working with '`frameworks`'.
- The workflow seems like:
  1. In the case of working with a `framework`[3] we implement an interface and in the most simple case we write the code to be executed later.
  2. We register the code to be executed later.
  3. We wait until we are called -to the previously registered code- when 'it is our turn'.
- The programmer does not indicate the workflow of the application, but the events triggered do it.

[3] Programación-3.

The HollyWood principle(II)

- You can check more information about it here .
- If you want to go deeper into this concept you should know that this principle is also known with other names:
  1. Inversion of control (IoC).
  2. Dependency Injection (DI).
- You can go deep into this programming techniques in other subjects such as Programación-3.

and in Vala...

- We know the essentials from the last unit: `signals` and `signal connections`:

```
    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    }
    }
```


Signals in Vala. The basics (I)

- Remember that if we want that a class can trigger signals it has to derive directly or indirectly from the class `GLib.Object`.
- A signal can have none, one or more parameters.
- The signature of the function we connect to a signal should coincide with the one of the signal...
- ...except in Vala that it allows us: Important!
  1. omitting parameters starting from the end, as we like...
  2. or providing the 'starter' of the signal as the first parameter of the `callback`.

Signals in Vala. The basics (II)

- For instance, for the signal '`Foo.some_event`' the following signals would be valid:

```
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)
```

- The names of the parameters are free.
- Specifying the source of the signal ('`source`') can help us to differenciate among connecting the same function to the same signal for different instances of the same type.

Signals and Lambda functions (λ)

- A signal can be connected to a in-line defined function, called lambda function (function-λ).
- They are also called anonymous functions because they have no name.
- We could rewrite the first example of page 68 using λ functions:

```
    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    }
    }
```

- Question: What are '`t`' and '`a`'? What do they represent?

Disconnecting signals in Vala (I)

- It can happen that from certain moment we are not interested on invoking a certain callback when the signal to which it was connected is triggered.
- To achieve this we should '*disconnect it*' from the signal. It is the inverse process to the connection.
- We can do it in several ways, the easiest is using the method '`disconnect`', with the following notation:

```
1    foo.some_event.connect (on_some_event);     // Connection...
     foo.some_event.disconnect (on_some_event); // Disconnection
```

- But what happen if we had connected the signal to a λ function?..remember that they have no name...

Disconnecting signals in Vala (II)

- The *trick* is knowing that the `connect` method returns a value of the type '*unsigned long integer*': `ulong`.
- Therefore, when we connect a λ function that we may want disconnect later, we should keep the returned value of the connection, for instance:

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


Signals with dynamic link in Vala (I)

- A signal can be redefined in a derived class ... in the same way as a normal method...
- For this purpose, it should have a *dynamic link*, this is the same as declaring it using the keyword '`virtual`'.
- When a signal is declared `virtual` it can have a implementation of a `callback` by default, besides this implementation can be redefined in derived classes, as follows:

```
     class Demo : Object {
2      public virtual signal void sig () {
         stdout.printf ("default callback\n");
4      }
     }
6
     class Sub : Demo {
8      public override void sig () {
         stdout.printf ("Substitution of the default callback\n");
10     }
     }
```


Signals with dynamic link in Vala (II)

- Can we connect several callbacks to a signal with a default `callback` assigned?.**sí**.
- IMPORTANT: They are executed **before** the default `callback`.
- There exists the possibility of connecting `callbacks` that are executed after the default `callback`, this is done with the method `connect_after`:

```
1    void main () {
       var demo = new Demo ();
3      demo.sig.connect (() => stdout.printf ("before\n"));
       demo.sig.connect_after (() => stdout.printf ("after\n"));
5      demo.sig (); // we trigger the signal
     }
7
     // Result:
9    // before
     // default callback
11   // after
```

Exercise: The domotic house

Working in groups we are going to write the code for an application that simulates -*as an example* - a domotic house. This house contains:

- Several `rooms`, each of them has...
  - `Blinds`, they can be `up` or `down`. We are interested in knowing when they go up.
  - `Doors`, they can be `open` or `closed`. We are interested in knowing when they are opened.
  - `Lights`, they can be `on` or `off`. We are interested in knowing when they are switched on.
- We have to create the classes/interfaces needed and a simple test program to create a house, with several rooms and each of them with several doors/ windows/ lights.
- Check that, indeed, when someone opens a door, switches on a light or a pulls up a blind, the domotic system warn us.