Tema: III.
Programació dirigida per esdeveniments i execució diferida de codi

Herramientas Avanzadas para el Desarrollo de Aplicaciones

Departament de Llenguatges i Sistemes Informàtics
Universitat d'Alacant

Contingut

Preliminars

- En termes de l'estructura i l'execució d'una aplicació representa l'oposat al que hem fet fins ara: *programació seqüencial*.
- La manera en la qual escrivim el codi i la forma en la qual s'executa aquest està determinada pels successos (*esdeveniments*) que ocorren com a conseqüència de la interacció amb el món exterior.
- Podem afirmar que representa un nou *paradigma de programació*, en el qual tot gira al voltant dels esdeveniments[1].

[1] Canvis significatius en l'estat d'un programa.

Programació seqüencial vs. dirigida per esdeveniments I

- A la programació seqüencial li diem a l'usuari el que pot fer a continuació, des del principi al final del programa.
- El tipus de codi que escrivim és com aquest:

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

- A la programació dirigida per esdeveniments indiquem:
  - Quines coses -*esdeveniments*- poden ocórrer?
  - El que cal fer quan ocorrin

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

Programació seqüencial vs. dirigida per esdeveniments II

- A partir d'aquest punt els esdeveniments poden ocórrer en qualsevol moment i marquen l'execució del programa.
- Encara que no ho sembli plantegen un problema seriós: el flux de l'execució del programa escapa al programador.
- L'usuari (com a font generadora d'esdeveniments) pren el control sobre l'aplicació.
- Això implica haver de portar cura amb el disseny de la aplicació tenint en compte que l'ordre d'execució del codi no ho marca el programador i, a més, pot ser diferent cada vegada.

Esquelet d'una aplicació dirigida per esdeveniments I

- Al principi de la mateixa duem a terme una iniciació de tot el sistema d'esdeveniments.
- Es defineixen tots els esdeveniments que poden ocórrer.
- Es prepara el generador o generadors d'aquests esdeveniments.
- S'indica què codi s'executarà en resposta a un esdeveniment produït -*execució diferida de codi*-.
- S'espera al fet que es vagin produint els esdeveniments.

Esquelet d'una aplicació dirigida per esdeveniments II

- Una vegada produïts són detectats pel 'dispatcher' o planificador d'esdeveniments, el qual s'encarrega d'invocar el codi que prèviament hem dit que havia d'executar-se.
- Tot això es realitza de forma ininterrompuda fins que finalitza l'aplicació.
- A aquesta execució ininterrompuda és al que es coneix com **el bucle d'espera d'esdeveniments**.
- Les aplicacions amb un interfície gràfic d'usuari segueixen aquest esquema de programació que acabem de comentar.
- Podem veure-ho de forma gràfica en el següent diagrama:

Diagrama de una aplicación dirigida por eventos

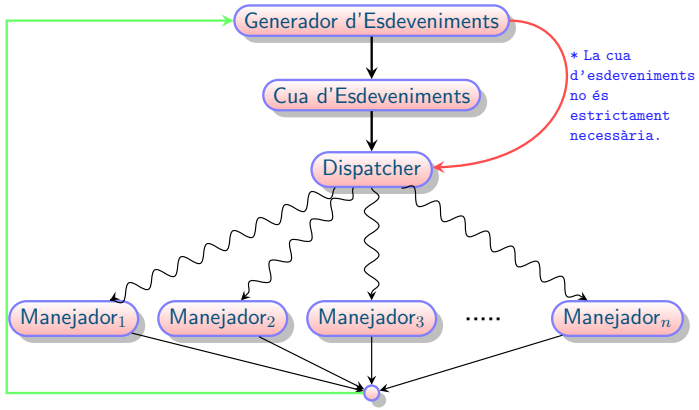

Figura: Diagrama d'una aplicació dirigida per esdeveniments.

I tot això en Vala...

- Els esdeveniments en Vala es denominen **senyals**.
- Són equivalents als *esdeveniments* en C# o als *listeners* en Java.
- Les *senyals* són un mecanisme proporcionat per la classe GLib.Object.
- Si volem que una classe nostra pugui emetre *senyals* -emetre esdeveniments-, ha de derivar directa o indirectament de GLib.Object.
- Una *senyal* es defineix com un membre d'una classe i es sembla a un mètode sense cos.
- Només veiem el seu signatura i nom precedits per la paraula reservada '`signal`' i el modificador d'accés.
- A una **senyal** li '*connectem*' tants mètodes o funcions com vulguem que s'executin quan es emeti aquest senyal.

Exemple de senyal en Vala I

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

public class Test : GLib.Object {
  public signal void sig_1(int a); // <=== SIGNAL

  public static int main(string[] args) {
    Test t1 = new Test();

    t1.sig_1.connect (f1);
    t1.sig_1.connect (f2);
    ...             // Conexion de la senyal
                    // amb el codigo a executar
                    // Ho veurem en el proximo
                    // tema.

    t1.sig_1 (2);   // Emision de la senyal
                    // Equival a cridar a: f1(2); f2(2)

    return 0;
  }
}
```


Exemple de senyal en Vala IIa

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

  public signal void new_try ();

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

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


Exemple de senyal en Vala IIb

```
void main () {
  const int max_tries = 2;
  var juan = new Player.with_name ("Juan");
  var pedro = new Player.with_name ("Pedro");

  juan.new_try.connect (player_try);
  pedro.new_try.connect (player_try);

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

Exemple de senyal 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
```


Exercici en classe. Preparació per a l'execució diferida de codi (I)

- L'execució diferida de codi té els seus orígens en el concepte de Callback .
- En Llenguatge 'C' un Callback no és més que un punter a una funció.
- Anem a començar a preparar en classe un exercici que farem en les pràctiques relacionades amb la *execució diferida de código*.

Exercici en classe. Preparació per a l'execució diferidad de codi (II)

- A la biblioteca estàndard de 'C' (`#include <stdlib.h>`) disposem de la funció "`qsort - ordena un vector`". El prototip de la mateixa és:

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

- Identifica cadascun dels seus paràmetres. Tracta d'entendre què és '`compar`'. Proposa un possible valor per a aquest paràmetre.
- Quan tinguis fet tot aquest estudi tracta de crear un programa executable mínim que et serveixi per comprovar com funciona '`qsort`'.

Execució diferida de codi. Preliminars (I)

- La constitueixen els anomenats *Callbacks* en C, són elements de baix nivell, ja els coneixem[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] Capdavanters a funcions.

Execució diferida de codi. Preliminars (II). Exercici en classe

- També sabem que `Vala` ens permet fer-ho millor que '**C**': '`signal`' + '`connect`'.
- Però també sabem que és *compatible* amb biblioteques de '**C**'...
- Fa un cop d'ull a la biblioteca de compatibilitat amb *Posix* de '**C**': vala posix , cerca en ella la referència a la funció qsort .
- Fijate a la declaració del tipus '`compar_fn_t`'.
- Exercici: Treballant en grups com en el exercici anterior, reescriu aquest exemple fet en **C** ara amb **Vala** i usant la capa de compatibilitat **Posix**. *15min.*

Execució diferida de codi. Preliminars (III)

- Altres llenguatges de programació sense arribar a tenir un suport sintàctic para això ofereixen alternatives de *més alt nivell*.
- Anem a veure el cas de tres *implementacions diferents* per al cas de **C++**:
  - libsigc++ : Empleada por la biblioteca gtkmm .
  - signal/slot : Emprada per la biblioteca Qt .
  - boost::signals : Implementació *semblant* a `libsigc++` però candidata a convertir-se en el suport estàndard d'execució de codi diferit en `C++` per la seva pertinença al projecte boost .
- En la documentació de cadascuna d'aquestes tres implementacions d'execució de codi diferit tenim exemples senzills, anem a veure un exemple de cadascun de ells. *10min. cadascun*

El principi d'Hollywood (I)

- És molt senzill: **No ens cridi...ja li cridarem**.
- S'empra sobretot quan es treballa amb '`frameworks`'.
- El flux de treball s'assembla a això:
  1. En el cas de treballar amb un `framework`[3] implementem un interfície i en el cas més senzill escrivim el codi a executar més endavant.
  2. Ens registrem... és a dir, indiquem d'alguna manera quin és el codi a executar posteriorment.
  3. Esperem al fet que es cridi -al codi registrat prèviament- quan li 'toqui': *No ens cridi...ja li cridarem*.
- El programador ja no 'dicta' el flux de control de la aplicació, sinó que són els esdeveniments produïts els que ho fan.

[3] Es estudien en l'assignatura Programació-3.

El principi d'Hollywood (II)

- Pots consultar més informació sobre ell aquí .
- Si vols ampliar més els teus coneixements sobre ell has de saber que a aquest principi també se li coneix per altres noms:
  1. Inversió de control (IoC).
  2. Injecció de dependències (DI).
- Pots ampliar més informació sobre aquestes tècniques de programació en assignatures com a Programació-3.

I tot això en Vala...

- Coneixem l'essencial ja del principo d'aquest tema: `senyals` i `connexions amb les mateixes`:

```
public class Test : GLib.Object {

  public signal void sig_1(int a);
  public void m (int a) { stdout.printf("%d\n", a); }

  public static int main(string[] args) {
    Test t1 = new Test();

    t1.sig_1.connect(t1.m);

    t1.sig_1(5);

    return 0;
  }
}
```


Senyals en Vala. El bàsic (I)

- Recorda que perquè una classe pugui emetre senyals deu derivar directa o indirectament de la classe `GLib.Object`.
- Un senyal pot tenir cap, un o més paràmetres.
- La signatura de la funció que connectem a un senyal deu coincidir amb la del propi senyal...
- ...tret que en Vala es permet: Important!
  1. que ometem tants paràmetres des del final com vulguem...
  2. o que proporcionem el 'originador' del senyal com a primer paràmetre de el `callback`.

Senyals en Vala. El bàsic (II)

- Per exemple, per al senyal '`Foo.some_event`' serien senyals vàlids totes aquestes:

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

- Els noms dels paràmetres són lliures.
- Especificar l'origen del senyal ('`source`') ens pot servir per diferenciar si connectem la mateixa funcion a la mateix senyal per a diferents instàncies del mateix tipus.

Senyals i funcionis Lambda (λ)

- A un senyal podem connectar-li una funció definida en línia, una funció lambda (funció-λ).
- També les hi crida funcions anònimes ja que no tenen nombre.
- El primer exemple que hem vist a la pàgina 68 podria reescriure's així usant funciones-λ:

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

    t1.sig_1.connect( (t, a) => {stdout.printf("%d\n", a);} );
    t1.sig_1(5);

    return 0;
  }
}
```

- Pregunta: ¿Qué son '`t`' y '`a`'? Què representa cadascun?

Desconnexió de senyals en Vala (I)

- És possible que a partir d'un moment no ens interessi que un determinat callback s'invoqui quan s'emeti el senyal a la qual estava connectat.
- Per aconseguir-ho hem de '*desconnectar-ho*' de la senyal. És el procés invers a la connexió.
- Podem fer-ho de diverses maneres, el cas més senzill és usar el mètode '`disconnect`', el qual usa aquesta notació:

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

- Però què ocorre si el que havíem connectat era una funció-λ?..recorda que no tenen nombre...

Desconnexió de senyals en Vala (II)

- El *truc* està a saber que el mètode `connect` retorna un valor de tipus '*enter llarg sense signe*': `ulong`.
- Per tant, quan connectem una funció-λ que pensem desconnectar, hem de guardar el valor que retorna la connexió, per exemple:

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


Senyals amb enllaç dinàmic en Vala (I)

- Un senyal pot ser redefinida en una classe derivada...com un mètode normal i corrent...
- Para això ha de tenir *enllaç dinàmic*, o el que és el mateix, ser declara '`virtual`'.
- Quan un senyal es declara `virtual` pugues tenir una implementació d'un `callback` per defecte, a més de que aquesta implementació pot ser redefinida en classes derivades, així:

```
     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     }
     }
```


Senyals amb enllaç dinàmic en Vala (II)

- A un senyal amb un `callback` per defecte se li poden connectar més callbacks?..**sí**.
- IMPORTANT: S'executen **abans** que el `callback` per defecte.
- Existeix la possibilitat de connectar `callbacks` que es executen després de el `callback` per defecte, això es fa amb el mètode `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
     // Resultat:
9    // abans
     // callback per defecte
11   // despues
```

Exercici: L'habitatge domòtica

Treballant en grups anem a escriure el codi per a una aplicació que simuli -*a manera d'exemple*- un habitatge domòtica. Aquest habitatge o *casa* consta de:

- Diverses `habitacions`, cadascuna de les quals disposa de...
  - `Persianes`, poden estar `pujades` o `baixades`. Estem interessats a saber quan es pugen.
  - `Portes`, poden estar `obertes` o `tancades`. Estem interessats a saber quan s'obren.
  - `Llums`, poden estar `enceses` o `apagades`. Estem interessats a saber quan s'encenen.
- Crea les classes/interfícies que consideris necessaris i un senzill programa de prova que creu una casa, amb vàries habitacions i cadascuna d'elles amb vàries portes/finestres/llueixes.
- Comprova que, efectivament, quan algú obre una porta, encén una llum o puja una persiana, el sistema domótico ens avisa d'això.