En esta página › Sinopsis

Tema 6: Programación dirigida por eventos.

Paradigma de programación.

- Se considera un *paradigma de programación* como un estilo o forma de escribir programas.
- Un *paradigma de programación* se caracteriza por los *conceptos empleados* y por la *manera de abstraer los elementos* utilizados en dar con la solución a un problema haciendo uso de un algoritmo.
- Esto hace que esté *limitado* por las características del lenguaje de programación empleado.
- Uno de los *paradigmas de programación* más empleados es el de la *programación orientada a objetos*, el cual lo vamos a estudiar a lo largo de la asignatura.
- Existen diversos *paradigmas de programación* y estos no tienen por qué ser excluyentes, se pueden complementar.
- Por todo ello podemos refererirnos a la *programación dirigida por eventos* (en adelante *PDE*) como un *nuevo* paradigma de programación.

Otros paradigmas de programación.

- Orientado a objetos.
- Imperativo.
- Declarativo.
  - Funcional.
  - Lógico.
- ...

aplicación.

```
-   Lo que pasa dentro de este bucle está determinado por los sucesos
    (_eventos_) que ocurren como consecuencia de la interacción con el
    mundo exterior, con el entorno en el que se ejecuta la aplicación.

-   Un _evento_ representa cualquier cambio significativo en el estado
    del programa.
```

Esquema básico de este tipo de aplicaciones I.

- Al principio de las mismas realizamos una iniciación de todo el sistema de eventos.
- Para todos los tipos de eventos que puedan ocurrir, especificamos en cuáles estamos interesados.
- Se prepara el generador o generadores de estos eventos.
- Estas tres acciones suelen estar ya implementadas en las bibliotecas destinadas a realizar *PDE*.
- Para todos los eventos en los que estemos interesados debemos decir qué código se ejecutará en respuesta a los mismos - *ejecución diferida de código* -.

Esquema básico de este tipo de aplicaciones II.

- Se espera a que se vayan produciendo los eventos.
- Una vez producidos, son detectados y tratados por el *dispatcher* o **planificador de eventos**. Este se encarga de invocar el código, que previamente hemos dicho que debía ejecutarse para cada evento (*manejador del evento*, *callback*, *slot*, etc...).
- Todo esto se realiza de forma ininterrumpida hasta que finaliza la aplicación.
- A esta *ejecución sin fin* es a lo que se conoce como **el bucle de espera de eventos**.

Esquema básico de este tipo de aplicaciones III.

- Un mismo manejador puede estar asociado a diferentes eventos.
- Un evento puede tener asociados 0 o más manejadores.
- Si asociamos más de un manejador a un evento, no debemos confiar en el orden de ejecución de los mismos.
- A la acción de asociar un manejador a un evento se la suele llamar *conexión*.

Programación secuencial vs. *PDE*.

```
// SECUENCIAL                          |  // DIRIGIDA POR EVENTOS
repetir                                |  son_eventos (ev1, ev2, ev3...);
  presentar_menu ();                   |  ...
  opc = leer_opcion ();                |  cuando_ocurra ( ev1, accion1 );
  ...                                  |  cuando_ocurra ( ev2, accion2 );
  si (opc == 1) entonces accion1 ();   |  repetir
  si (opc == 2) entonces accion2 ();   |    ...
  ...                                  |  hasta terminar
hasta terminar
```

Diagrama.

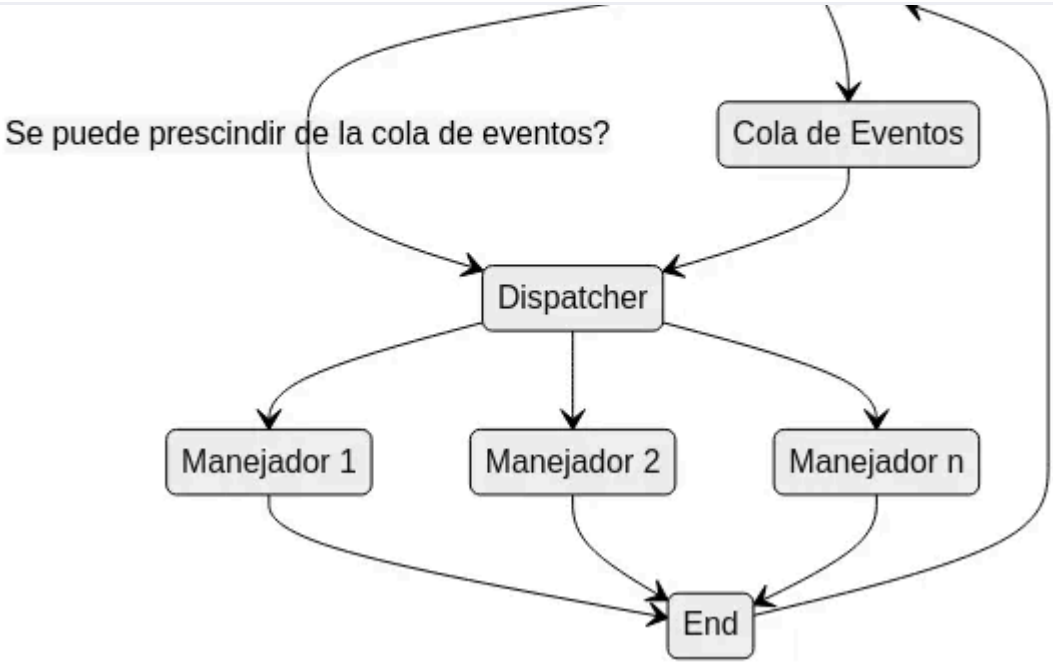

Figura 1: Esquema básico de una aplicación bajo el paradigma de la *PDE*.

PDE y C++.

- C++ **no** dispone de ningún mecanismo en el lenguaje para realizar este tipo de programación.
- Debemos recurrir a soluciones externas para ello.
- Estas *soluciones* pueden ser de distinto tipo:
    - Usando un preprocesador especial que añade esta característica nueva al lenguaje. Es el caso de la biblioteca Qt. El preprocesador empleado se llama MOC y la parte de *PDE* que implementa es la que denomina signal/slot.

      Qt emplea el término *señal* (**signal**) en lugar de *evento* y el de **slot** en lugar de *callback*.
- Mediante el uso de una biblioteca que implemente de algún modo el paradigma de la *PDE*. Existen varias de estas bibliotecas:
    1. Libsigc++, es una adaptación a C++ de la versión de *C* que hay en GLib.
    2. GLib (signals)

```
#include <QObject>

class Counter : public QObject {
    Q_OBJECT

public:
    Counter() { m_value = 0; }

    int value() const { return m_value; }

public slots:
    void setValue(int value);

signals:
    void valueChanged(int newValue);

private:
    int m_value;
};
```

```
void Counter::setValue(int value) {
   if (value != m_value) {
       m_value = value;
       emit valueChanged(value);
   }
}

Counter a, b;
QObject::connect(&a, &Counter::valueChanged,
                 &b, &Counter::setValue);

a.setValue(12);     // a.value() == 12, b.value() == 12
b.setValue(48);     // a.value() == 12, b.value() == 48
```

Un ejemplo sencillo con Libsigc++.

```
class AlienDetector {
public:
  AlienDetector();

  void run();

  sigc::signal<void> signal_detected;
};

AlienDetector::AlienDetector() {}

void AlienDetector::run() {
  sleep(3); // wait for aliens
  signal_detected.emit();
}
```

```
using namespace std;

void warn_people() {
  cout << "There are aliens in the carpark!" << endl;
}

int main() {
  AlienDetector mydetector;
  mydetector.signal_detected.connect( sigc::ptr_fun(warn_people) );

  mydetector.run();

  return 0;
}
```

Boost y C++.

- Boost es un conjunto de bibliotecas escritas en C++ con vistas a su portabilidad (compilador, s.o., etc...).

todas y cada una de las bibliotecas que la componen.

- El proyecto Boost es *importante* por:

  1. La altísima calidad del código fuente C++ que tienen sus bibliotecas.
  2. Algunas de estas bibliotecas pueden pasar a formar parte de la biblioteca estándar de C++ en un futuro.

Boost y *PDE* I.

- Boost dispone de una biblioteca de implementación de señales (*eventos*) que que permite realizar *PDE*: **boost::signals**. Nos centraremos concretamente en su versión 2 o **boost::signals2**.

- Esta biblioteca implementa un marco de PDE basado en la emisión de señales (eventos) y la posterior ejecución del código asociado a los mismos, los llamados *slots* o *callbacks*.

- Una *señal* o *evento* puede tener asociados 0 o más *manejadores* o *callbacks*, mientras que un *callback* o *slot* puede estar asociado con más de una señal.

Boost y *PDE* II.

- A la asociación de un *slot* con una *señal* se le llama **conexión**.

- Los *manejadores* o *slots* se ejecutan cuando la *señal* se emite (el evento se ha producido).

- Un *manejador* se puede desconectar de la *señal* a la que se ha conectado previamente.

- Un *manejador* puede devolver un valor. En el caso de `boost::signals2` es un dato de tipo `boost::optional`, este actúa como un *wrapper* sobre el valor real devuelto.

boost::signals2 : ejemplos I.

```
// compilar con: g++ signals1.cc -o signals1
```

```
  std::cout << "void my_first_slot()\n";
}

class Car {
public:
  void out_of_gas_cb () { std::cout << "Gas needed!\n"; }
  static void class_method () { std::cout << "Class Method!\n"; }
};
```

```
int main() {
  boost::signals2::signal<void ()> sig;
  Car c;

  sig.connect(my_first_slot);

  ////////////////////////////////////////////////////
  // Caso especial:                                 //
  //   El callback o slot es un método de una clase. //
  ////////////////////////////////////////////////////
  sig.connect( std::bind(&Car::out_of_gas_cb, &c) );
  sig.connect( Car::class_method );

  std::cout << "Emitting the signal...\n";
  sig();
}
```

boost::signals2 : ejemplos II.

```
// compilar con: g++ signals2.cc -o signals2
#include <iostream>
#include <boost/signals2.hpp>

class Car {
public:
  Car (std::string b) { brand = b; }
```

```
private:
  std::string brand;
};
```

```
int main() {
  boost::signals2::signal<void ()> sig;
  Car a("audi");
  Car s("seat");

  //auto cna = sig.connect(boost::bind(&Car::out_of_gas, &a));
  boost::signals2::connection cna = sig.connect(std::bind (&Car::out_of_gas_cb, &a));
  boost::signals2::connection cns = sig.connect(std::bind (&Car::out_of_gas_cb, &s));

  std::cout << "Emitting the signal...\n";
  sig();

  cna.disconnect ();            // disconnect audi signal

  std::cout << "\nDisconnect & emit the signal...\n\n";
  sig();
}
```

```
//==============//
// Uso de bind: //
//==============//-------------------------------------------------------------//
// int sub(int lhs, int rhs);      // returns lhs - rhs                        //
// bind(sub, 3, 4);                // returns a function object whose          //
//                                 // operator() returns sub(3, 4)             //
// 1) bind(sub, 3, 4)();                                                       //
// 2) auto functor = bind(sub, 3, 4); // define a variable for the functor     //
//   cout << functor() << '\n';      // call the functor, returning -1.        //
//-----------------------------------------------------------------------------//
```

boost::signals2 : ejemplos III.

```
class Car {
  public:
    Car () { temp = 90; }

    void out_of_gas_cb (int dist) {
      std::cout << "Gas needed!\nDistance to next petrol station: "
                << dist << "km\n";
    }

  private:
    int temp;
};
```

```
Int main() {
    Car c;

    boost::signals2::signal<void (void)> sig;
    boost::signals2::signal<void (Car*, int)> sig2;

    sig.connect ( std::bind(&Car::out_of_gas_cb, &c, 20) );
    sig2.connect ( &Car::out_of_gas_cb );

    std::cout << "Emitting the signal...\n";
    sig();
    sig2(&c, 12);
}
```

boost::signals2 : ejemplos IV.

```
// compilar con: g++ signals4.cc -o signals4
#include <iostream>
#include <boost/signals2.hpp>

class Car {
  public:
    Car () { temp = 90; }
```

```
      return temp;
    }

  private:
    int temp;
};
```

```
int main() {
    Car c;

    boost::signals2::signal<int (void)> sig;
    sig.connect ( std::bind(&Car::out_of_gas_cb, &c, 20) );

    std::cout << "Emitting the signal...\n";
    int t = *sig();               // returns a boost::optional

    std::cout << "The temp. of car's engine is " << t << "ºC.\n";
}
```

Boost y *PDE* III.

- Si conectas varios *slots* a una misma señal y estos devuelven valores, no olvides echar un vistazo a Signal Return Values (Advanced).
- Si por algún motivo necesitas tener control sobre el orden de ejecución de los *slots* conectados a una señal, consulta la documentación sobre los llamados *grupos de llamadas*: Ordering Slot Call Groups (Intermediate).

Boost y *PDE* IV.

- ¿Cómo capturar señales enviadas por el hardware y reconvertirlas en señales de `boost::signals2` ?
- Vamos a probarlo provocando una división por 0, capturar la excepción producida y reconvertirla en una *señal* de `boost::signals2` .

P2 GIR

En esta página › Sinopsis

```
#include <cstdlib>
#include <csignal>
#include <string>

using namespace std;

class Hardware;
using HWPtr = Hardware *;

class Hardware {
public:
  Hardware(string n) {
    if (SIG_ERR == signal(SIGFPE, fpe)) {
      cerr << "failure to setup signal.";
    }
    name = n;
  }

  // Signals //-----------------------------------------------------
  boost::signals2::signal<void(HWPtr)> onDivisionByZero;

  string hw_name() { return name; }
private:
  int data;
  string name;
  // Class methods //-----------------------------------------------
  static void fpe(int n);
};

Hardware gHw("Computer CPU");

void Hardware::fpe(int n) {
  cerr << "Low level signal caught.\nCalling high-level signal.\n\n";
  gHw.data = n;
  gHw.onDivisionByZero(&gHw);
}

//-- Main program: Compilar con '-O' ---------------------------------
```

```
}

int main(int argc, char *argv[]) {
  gHw.onDivisionByZero.connect(myFPEHandler);

  auto n = 3;
  int den = 0;

  cout << "Division hecha?: r= " << r << '\n';

  auto r = n / den;
  return 0;
}
```

Boost y programación secuencial vs. *PDE*.

- Del ejemplo del *Control de Misión* compara las versiones *secuencial* y *dirigida por eventos*:

```
// SECUENCIAL
void MissionControl::follow_voyagers () {
  ulong impulses = 0;
  while ((not v1.is_outof_ss()) or
         (not v2.is_outof_ss())) {
    std::cout << "[" << ++impulses
              << "]-------Following Voyagers------------\n";

    v1.travel ();
    if (v1.is_outof_ss()) do_something_when_outof_ss(v1);

    v2.travel ();
    if (v2.is_outof_ss()) do_something_when_outof_ss(v2);
  }
}
```

```
// DIRIGIDA POR EVENTOS
void MissionControl::follow_voyagers () {
```

```
              << "]---------Following Voyagers----------\n";
    v1.travel ();
    v2.travel ();
  }
}
```

PDE y patrones de diseño. Ampliación de conocimientos.

- La PDE estudiada es una realización concreta de un patrón de diseño software. Concretamente del patrón llamado Observer.
- El libro Design Patterns: Elements of Reusable Object-Oriented Software se considera un libro básico en esta materia.
- En la página de enlaces de la web de la asignatura tienes recogido un curso en *YouTube* sobre patrones de diseño en `C++` . Aquí tienes el primero de los vídeos dedicados al patrón *Observer*.

Aclaraciones.

- **Este contenido no es la bibliografía completa de la asignatura**, por lo tanto debes estudiar, aclarar y ampliar los conceptos que en ellas encuentres empleando los enlaces web y bibliografía recomendada que puedes consultar en la página web de la ficha de la asignatura y en la web propia de la asignatura.

Espacios de nombres.

Polimorfismo. Enlace dinámico.