Programación Orientada a Eventos

Werner Ezequiel Maximiliano
eze210 <at> gmail.com

Facultad de Ingeniería
Universidad de Buenos Aires

Contenidos

Subsection 1

Algunos Paradigmas de Programación

Paradigmas de Programación

1. Programación Secuencial o Imperativa: Se basa en definir la secuencia de pasos que debe seguir la ejecución del programa.
2. Programación Orientada a Objetos: Se abstraen las entidades del problema, su comportamiento, su estado, y sus interacciones.
3. Programación Funcional: Las funciones de este paradigma son predicados matemáticos.
4. **Programación Orientada a Eventos**: Se definen *eventos* y *cómo manejarlos*.

Hoy vamos a introducir este último.

Subsection 1

Concepto

¿En qué consiste este paradigma?

1. Toda **acción** que ejecute el sistema será en respuesta a los **sucesos** que acontezcan.
2. A esos sucesos los llamaremos **eventos**.
3. Lo que debemos programar son las acciones para atender o responder a los eventos: los **handlers**.

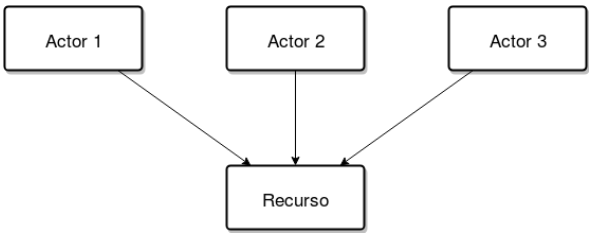
Subsection 2

Eventos

¿De dónde vienen los eventos?

1. Acciones del Usuario:
   1.1 Click en un botón.
   1.2 Movimiento del puntero del mouse por encima de algún widget.
   1.3 Key-Down.
   1.4 Key-Up.
2. Basados en tiempo:
   2.1 Se alcanza una fecha u horario.
   2.2 Se vence un timeout.
3. Generados por otros eventos:
   3.1 En nuestro código fuente podemos disparar eventos.
4. Definidos por el usuario.
5. Sucesos del entorno.

¿De dónde vienen los eventos?



- **Como tenemos múltiples fuentes de eventos, tenemos ¡muchos! problemas de concurrencia**

Subsection 3

Cola de Eventos (Event Queue)

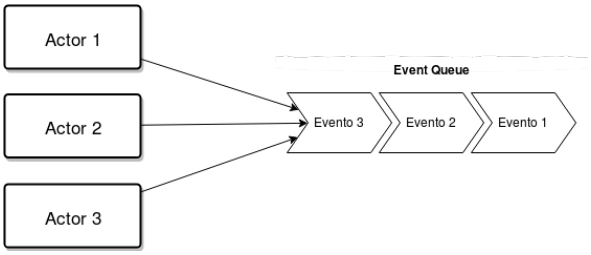
¿Cómo atenderlos a todos?

- Antes en el curso vimos varias técnicas para evitar problemas de concurrencia, y destacamos dos:
  1. Mutex para encerrar las critical sections.
  2. Colas bloqueantes.
- Esas técnicas tienen en común que serializaban cosas.
  1. Usando mutex podemos ver una serialización de critical sections (se hace primero la CS que pidió el mutex primero, después otra, etc).
  2. Usando colas bloqueantes, son los datos los que quedan serializados por la naturaleza FIFO de la estructura.

¿Cómo atenderlos a todos?

- Podemos modelizar a los eventos como estructuras de datos.
- Los eventos son producidos por múltiples actores (como los que mencionamos antes).
- Y luego son agregados a una cola para que alguien los atienda.

Event Queue



- A esa cola la llamaremos **cola de eventos** (**event queue**).
- Protegiendo la cola de eventos ya no tenemos problemas de concurrencia. ¿Por qué?

Subsection 4

Bucle de Eventos (Event Loop)

¿Quién lee los eventos de la cola?

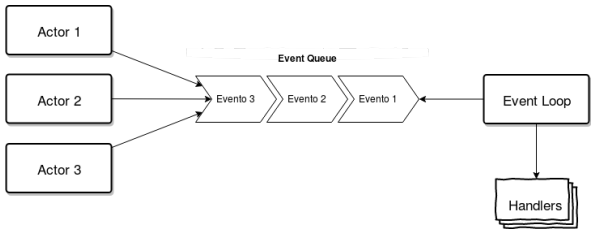
Un pseudocódigo para manejar eventos

```
1 while se debe continuar:
2   evento := obtener el siguiente elemento de la cola de eventos
3
4   if evento == salir:
5     se debe continuar := false
6
7   else if existe manejador para evento:
8     ejecutar manejador
```

- A este bucle se lo llama **event loop** y es un patrón muy usado en aplicaciones con *Graphical User Interfaces (GUIs)*.
- Es similar al **game loop** que usan los juegos para simular el paso del tiempo.

Event Loop

Actualizando nuestro diagrama...



- A esa cola la llamaremos **cola de eventos** (**event queue**).
- Protegiendo la cola de eventos ya no tenemos problemas de concurrencia. ¿Por qué?

Subsection 5

Manejadores (Handlers)

Handlers

- Los **handlers** son secciones de código que saben cómo responder a la aparición de un Evento.
- Pueden requerir cierta información sobre el Evento (en qué coordenadas de la pantalla se hizo un click, cuánto duró la pulsación de un botón, alguna información que decida el usuario).
- Como los va a disparar el event loop, se van a ejecutar de manera secuencial:
  - No van a tener problemas de concurrencia entre ellos.
  - Si uno tarda mucho, va a retrasar a todos los que vengan después.

Handlers en aplicaciones con GUI

- **En aplicaciones con GUI tenemos que programar handlers cortos, y que den feedback al usuario.**
- Si el usuario pide algún procesamiento largo conviene pasarle la tarea a algún thread (ya sea lanzándolo o no), decirle al usuario que se está procesando su pedido, y terminar el handler para que el event loop siga adelante.
- En muchos frameworks gráficos, el event loop corre en el hilo principal.
  - Algunos nos abstraen de programarlo (como GTK).
  - En otros lo tenemos que programar nosotros (como SDL).

Conclusiones

- La programación orientada a eventos nos evita los problemas de concurrencia entre los handlers.
- Es un paradigma muy usado para interactuar con el usuario, ya que es él quién decide el flujo del programa.
- Cuando hay GUI, como los handlers se ejecutan secuencialmente conviene programarlos cortos y delegar las tareas largas en otro hilo.
- Es casi esencial el multithreading cuando tenemos GUIs, y el loop de eventos suele correr en el hilo principal.

Temas (muy) relacionados

- Muchos frameworks trabajan con event queues y nos abstraen de su uso.
  1. En C/C++, podemos mencionar a QT y GTK+ (o gtkmm), que usan una event queue para manejar las interacciones con el usuario. Veremos una de estas opciones la clase que viene.
  2. En otros lenguajes, NodeJS basa TODO en una event queue.
- Este patrón es muy parecido al Observer (pero con un poco menos de acoplamiento).
- En abstracto, el concepto es el mismo que el de una *cola de mensajes*, o un *publish/subscribe*, pero el término que usamos depende del contexto.
- Varios lenguajes de programación, como Go implementan **channels** (nombre en Go) nativos, que son en esencia event queues o message queues.