Programación III

Utilización de eGTK

Universidad de Valladolid
Departamento de Informática

Versión 0.5. Sistemas. 2002. FéLIX

27/9/2004

Índice

1. Introducción

El objetivo de este documento es servir de guía para la elaboración de una primera aplicación Eiffel dotada de una interfaz de ventanas basada en la biblioteca eGTK. Por ello la descripción con detalle de las capacidades de las clases que forman eGTK queda fuera de sus objetivos. Para obtener esta información se recomienda consultar las clases que forman eGTK, su forma corta, o los ejemplos que acompañan a la biblioteca.

La biblioteca eGTK no es sino un envoltorio que permite utilizar desde Eiffel la biblioteca GTK. El proyecto eGTK está en fase de desarrollo, por lo que no implementa aún todos los widgets disponibles en GTK, sin embargo permite elaborar de forma razonable aplicaciones totalmente funcionales.

Este documento ilustra un conjunto de cuatro ejemplos que puede ser descargado desde la siguiente dirección:

`ftp://ftp.lab.fi.uva.es/pub/priii/ejemplos_gtk`

Para compilar estos ejemplos, o cualquier otra aplicación que utilice eGTK se recomienda utilizar el comando egtkbuild en lugar de compile. Este comando proporciona automáticamente los parámetros de compilación necesarios. El comando egtkbuild requiere como parámetro el nombre de la clase raíz del sistema, sin la extensión, y el ejecutable construido se llama como la misma clase raíz, también sin extensión, en lugar de a.out.

Además es preciso ampliar la ruta de búsqueda de SmallEiffel para indicar la localización de las clases que forman parte de eGTK. Para ello hay que escribir un fichero loadpath.se con la localización de las clases. Todos los directorios de ejemplos proporcionados disponen de un fichero como ese, si bien, para que la compilación sea posible, es necesario que exista una variable de entorno EGTK que apunte al directorio raíz de la instalación de eGTK.

2. La primera aplicación

La clase raíz de un sistema basado en eGTK debe realizar ciertas tareas específicas asociadas a esta biblioteca: Es necesario inicializar el sistema de ventanas y activar el bucle de tratamiento de eventos.

Por ello es habitual heredar de la clase **GTK_APPLICATION** en la construcción de la clase raíz. Con ello dispondremos de acceso a los métodos initialize_tool_kit , encargado de las tareas de inicialización, y main_loop, que lanza el bucle de comprobación de eventos.

El bucle de comprobación de eventos se encarga de detectar los eventos producidos en las ventanas, como clicks del ratón en los elementos que las

forman, y activar la ejecución del código asociado a estos eventos. El final del bucle de comprobación de eventos supone el final del sistema de ventanas, pero no necesariamente de la aplicación, que terminará cuando termine la ejecución del método de creación de la clase raíz.

Como primera aplicación basada en eGTK podemos plantearnos el típico "hola mundo", resuelto en el directorio `hola`.

Necesitamos construir una única clase, lanzadora del sistema, que además de las responsabilidades ya mencionadas deberá crear la ventana principal de la aplicación y los elementos que la constituyen.

Crearemos la ventana como instancia de **`GTK_WINDOW`**, el botón, instancia de **`GTK_BUTTON`**, etiquetado con el mensaje, y deberemos añadir el botón a la ventana. Para ello **`GTK_WINDOW`** dispone del método add_widget. Por último, es preciso que mostremos todos los widgets utilizados, enviándoles el mensaje show. Los widgets desaparecerán de pantalla si reciben el mensaje hide.

En este sistema elemental, pulsar el botón que aparece en la ventana no provoca ninguna reacción de la aplicación, y aunque eliminemos la ventana no detendremos el bucle de comprobación de eventos, por lo que la aplicación seguirá activa. Para detenerla es preciso cancelar su ejecución mediante la combinación de teclas `Control-C`.

3. Mejorar la primera aplicación

La primera de las diferencias que incorpora el segundo ejemplo, contenido en el directorio `hola2` es que el click sobre el botón de la aplicación provoca la aparición de un mensaje en el terminal. Para ello debemos utilizar una clase concreta que implemente la clase diferida **`GTK_COMMAND`**, y asociar un objeto instancia de esta clase con el evento "clicked" en el botón.

Hacer efectiva la clase **`GTK_COMMAND`** mediante **`COMANDO_BOTON`** supone definir su método diferido execute. Este es precisamente el método que será ejecutado cuando se produzca el evento a que está asociado el objeto.

La asociación del objeto a un evento se consigue enviando el mensaje add_action al objeto sobre el que queramos realizar la asociación. Los parámetros del mensaje son el nombre del evento asociado y una referencia al objeto que lo manejará, boton.add_action("clicked", maneja_boton), en nuestro caso.

Para que la aplicación termine de una forma más natural, tras la destrucción de su ventana, deberemos controlar el evento destroy de la misma. La técnica utilizada es la que acabamos de describir, si bien ahora encargaremos el manejo del evento a la clase **`EGTK_QUIT_COMMAND`** proporcionada por la

biblioteca eGTK.

El comportamiento del sistema construido de este modo es el estándar en los sistemas de ventanas convencionales: Al destruir la ventana principal, la aplicación termina sin más confirmaciones. Sin embargo, y aunque habitualmente esto no se recomienda, es posible interceptar esta destrucción. La destrucción de la ventana se produce porque tras la ejecución del método execute el evento destroy se transmite a la ventana. Para evitarlo debemos utilizar un manejador que herede de **GDK_EVENT_COMMAND** y utilizar adecuadamente la característica last_event_handled.

4. Una aplicación más compleja

Habitualmente las aplicaciones basadas en eGTK realizan algún trabajo además de mostrar ventanas y botones. Mezclar la realización de este trabajo con la gestión de las ventanas resulta poco práctico. En su lugar es preferible aislar dos subsistemas encargados de la gestión de la interfaz de usuario y la solución del problema planteado respectivamente. Estos dos subsistemas están construidos en torno a un conjunto de clases "observadoras" encargadas del interfaz, y una clase "sujeto" de la observación, que coordina las tareas relacionadas con el problema que realmente resuelve la aplicación.

Dividir de este modo el sistema facilita la construcción del mismo y su posterior mantenimiento.

El tercer ejemplo, contenido en el directorio `aplicacion` implementa un contador controlado desde una ventana. El contador se incrementa en uno con cada pulsación del botón de la ventana, y ésta muestra en todo momento el valor del contador.

Las dos clases fundamentales utilizadas en este ejemplo son **CONTADOR** y **OBSERVADOR**. La primera implementa el contador, mientras que la segunda representa la ventana que lo controla. La arquitectura básica del sistema está representada en el diagrama de clases de la figura 1.

El punto crucial de este diseño es el modo en que se establece la comunicación entre **CONTADOR** y **OBSERVADOR**. La idea es que, aunque de hecho es siempre el observador quien directa o indirectamente solicita modificaciones de su estado al contador, el observador no modifica su estado, que constituye la única información sobre el sistema que ve el usuario final, hasta que no recibe notificación de sus cambios por parte del contador.

Por otra parte, el contador no indica en esa notificación su nuevo estado al observador, sino solamente que debe actualizarse, dejando en manos del observador la decisión sobre qué datos del estado del contador necesita consultar.

Figura 1 Diagrama de clases del sistema “aplicacion”

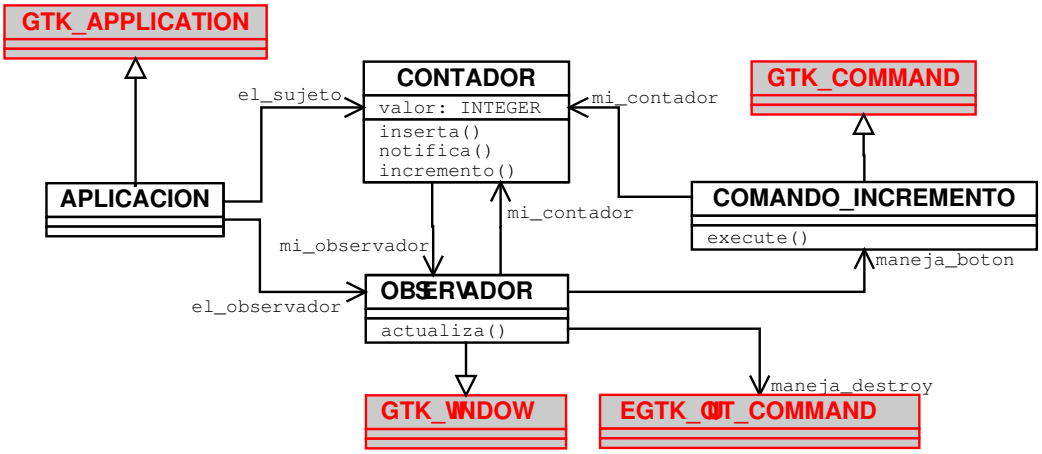
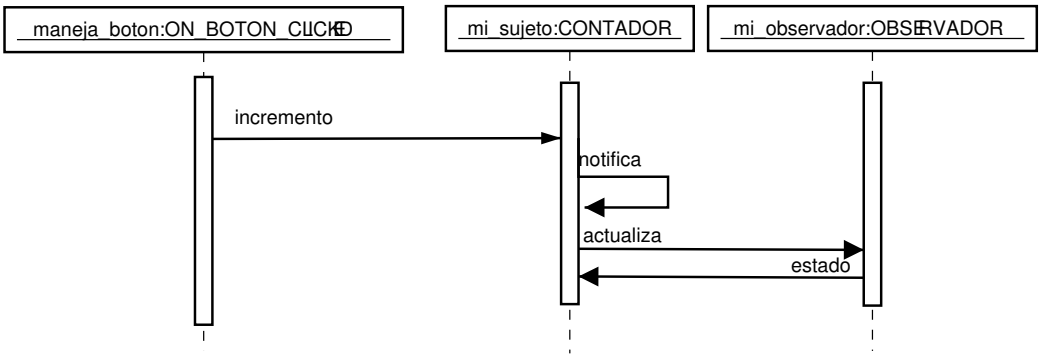


Figura 2 Diagrama de secuencia del proceso de notificación en el sistema “aplicacion”



Esta idea, que en este caso especialmente simple puede parecer innecesariamente compleja, permite en general construir sujetos y observadores realmente independientes. El sujeto no necesita sino acceder a un método actualiza en el observador, sin necesidad de conocer cómo es el observador o qué datos necesita, mientras que el observador sólo debe conocer los aspectos del sujeto en que realmente está interesado, y se limita a transmitirlos al usuario. El proceso completo de notificación, aplicado a nuestro ejemplo, aparece ilustrado en la figura 2.

La necesidad del sujeto de comunicar al observador sus cambios de estado obliga a establecer una referencia del primero al segundo. Este, a su vez, necesita consultar su estado al sujeto, por lo que necesita de una referencia al mismo.

Los manejadores de botones, que finalmente activan las solicitudes al sujeto, necesitan también acceso al mismo, si bien esta solución podría cambiar cuando las solicitudes de actualización del observador sean más complejas y dependan de su estado. En esta situación sería preferible que el manejador delegase en el observador la realización de la solicitud de cambio.

Técnicamente esta aplicación no presenta muchas diferencias con la segunda versión del "hola mundo", aunque la construcción de la ventana se realiza ahora en la clase **OBSERVADOR**. Tan solo se han añadido algunos elementos cosméticos que facilitan el maquetado de la ventana como cajas y separadores, instancias de **GTK_BOX** y **GTK_SEPARATOR** respectivamente, y etiquetas que muestran información al usuario, instancia de **GTK_LABEL**. Hay que hacer notar que, si bien casi todos estos elementos se utilizan como entidades locales al método de construcción de la ventana, valor_mostrado debe ser atributo de la clase, puesto que permite visualizar el estado del contador, y por ello debe ser modificable a lo largo de la ejecución del sistema.

5. Complicando algo más la aplicación

El diseño utilizado en la aplicación anterior es especialmente adecuado cuando utilizamos más de un observador para informar del estado del sujeto. Para ilustrar esta situación nos planteamos una nueva versión, contenida en el directorio `aplicación2`, en que dos ventanas de tipos distintos permiten utilizar el mismo contador del ejemplo anterior. El primer tipo de ventana es como el utilizado en la versión anterior, mientras que el segundo dispondrá de botones de incremento individual y de cinco en cinco.

Los diagramas correspondientes a la nueva aplicación están recogidos en las figuras 3 y 4.

En esta aplicación hay que resaltar el hecho de que ahora el contador trabaja con dos observadores. Para simplificar la tarea de notificación es recomendable utilizar un array de observadores, pero puesto que los observadores no son homogéneos, es preciso definir una clase **OBSERVADOR_ABSTRACTO**, dotada de un método diferido actualiza. El proceso de suscripción de un observador se realiza insertando el observador en el correspondiente array de observadores abstractos. La notificación se reduce a recorrer todo el array de observadores, invocando el método actualiza de cada uno de ellos.

Por otra parte, para conseguir un comportamiento "razonable" del botón de incremento por cinco, es preciso complicar ligeramente su correspondiente manejador, **COMANDO_INCREMENTO2**.

En efecto, si nos limitamos a invocar repetidamente al incremento del sujeto, las modificaciones del aspecto de los observadores, que son tratadas

Figura 3 Diagrama de clases del sistema “aplicacion2”

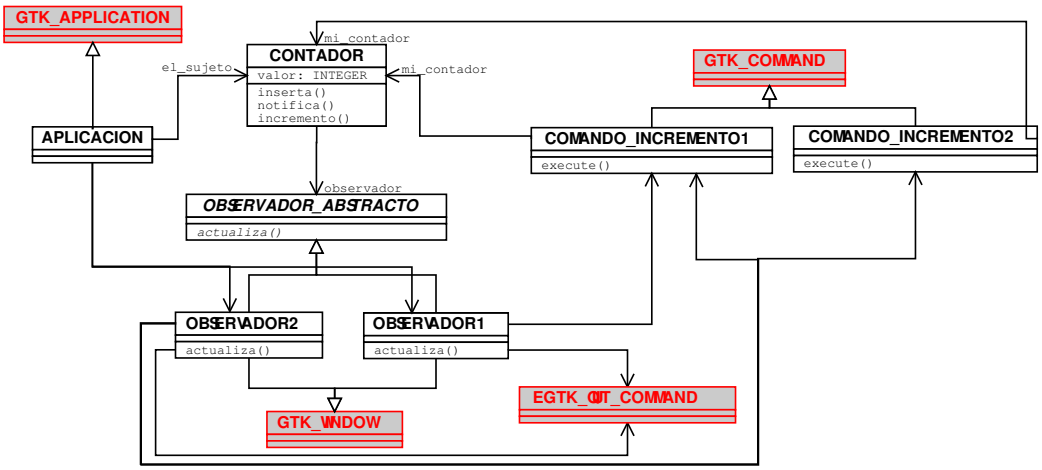
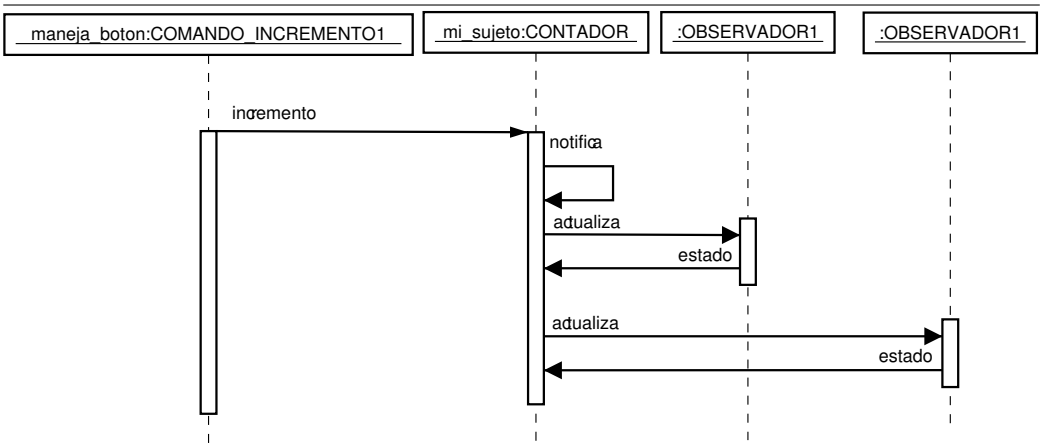


Figura 4 Diagrama de secuencia del proceso de notificación en el sistema “aplicacion2”



por el sistema también como eventos, no serán apreciables en pantalla hasta que no termine la ejecución del método execute, y por ello el incremento se presentará al usuario en un único paso, sin pasar por los estados intermedios. Para evitar ésto, el método execute debe asegurarse de que se vacía la cola de eventos a tratar cada vez que termina una solicitud de incremento al contador. Utilizamos para ello dos métodos de **GTK_MAIN**: main_loop_iteration, que trata un evento más de la cola de eventos pendientes, y events_pending que indica la presencia de eventos pendientes en dicha cola.

Para acceder a estos métodos, podemos construir la clase manejadora del evento, **COMANDO_INCREMENTO2** como heredera de **GTK_MAIN**. Se trata de lo que Meyer describe como "herencia oportunista": No es que nuestro comando sea una especialización de esta clase, sino que la herencia nos permite acceder de manera cómoda a los métodos que necesitamos utilizar.

Por último hay que reseñar una dificultad extra en la construcción de los observadores. Necesitamos que hereden simultáneamente de **OBSERVADOR_ABSTRACTO** y **GTK_WINDOW**, pero la segunda clase dispone de versiones redefinidas de dos métodos de la clase **GENERAL**, copy e is_equal. El conflicto de herencia repetida que se plantea puede ser resuelto de forma simple eliminando de **OBSERVADOR_ABSTRACTO** ambos métodos mediante la clausula undefine.