Capítulo

6

Programación con GUI

El presente capítulo se centra en el trabajo con Interfaces Gráficas de Usuario (IGU por sus siglas en español, GUI por sus siglas en inglés) utilizando ventanas y otros elementos para el diseño de las mismas, dentro del sistema operativo Windows. Para el caso particular de Java, dada su característica multiplataforma, los conceptos dados aquí son aplicables a cualquier plataforma que soporte Java.

Interfaces GUI con Ventanas

Una interfaz gráfica de usuario (GUI por sus siglas en inglés) es el conjunto de elementos gráficos que un programa, ejecutándose en un computador, utiliza para permitirle al usuario interactuar "visualmente" con él. Un programa GUI utiliza hardware que lo asiste (monitor y tarjeta de video) trabajando en "modo gráfico". Si bien el texto es, en esencia, un gráfico, las interfaces basadas exclusivamente en texto (TUI por sus siglas en inglés), con monitores trabajando en "modo texto", son diferenciadas de las anteriores.

Un programa puede trabajar con una TUI, con una GUI o con ambas, aunque en éste último caso lo que se tiene es realmente un programa trabajando en "modo gráfico" emulando el comportamiento de un "modo texto".

Casi desde sus inicios (que se remontan, por lo menos, a 1973, con la primera computadora Alto de la empresa Xerox PARC puesta en funcionamiento, o más conocida, la Star de Xerox en 1981), el concepto de ventanas formó parte del concepto de GUI, como una estrategia de organización del área gráfica.

Tipo de Interfaces GUI con Ventanas

En los programas GUI con ventanas, el usuario utiliza dispositivos como el teclado y el ratón, cuyo uso envía mensajes al computador y éstos son capturados por el sistema operativo, el cual decide a qué ventanas pertenecen dichos mensajes y los "envía" a las aplicaciones correspondientes para que éstas realicen alguna acción. Es importante tener en cuenta que este mecanismo de mensajes es utilizado por otros programas, incluyendo el propio sistema operativo, para "generar" nuevos mensajes, no relacionados con la interacción gráfica mediante las ventanas, y "enviarlos" a los programas en ejecución. Ejemplos de éstos son los mensajes de comunicación entre programas (corriendo en la misma computadora o en computadoras diferentes conectadas a una red), los mensajes de aviso de los cambios en la configuración del sistema operativo (resolución de la pantalla, idioma del teclado, fecha, cambio de usuario, etc.), los relacionados a la escasez de recursos, etc.

Este sistema de mensajería, con distintas variantes, es utilizado por los sistemas operativos con soporte GUI. En particular veremos el caso del sistema operativo Windows y cómo su sistema de mensajería es manejado desde diferentes lenguajes de programación.

En la actualidad, existen dos tipos de programas GUI con ventanas comunmente utilizados: Los programas Stand-Alone y los programas dentro de navegadores de Internet (browsers).

Programas Stand-Alone

Los programas Stand-Alone crean una o más ventanas con las que el usuario interactúa. Comunmente estos programas poseen una ventana principal y una o más ventanas especializadas en alguna labor específica. Estos programas pueden ejecutar directamente (en caso de ser programas ejecutables) o mediante un programa intérprete.

Ejemplos de estos programas son los creados con C o C++ utilizando directamente el API de Windows, así como las *Aplicaciones* de Java y las *Aplicaciones de Formularios de Ventanas* de ..NET.

Programas Basados en Web

Los programas basados en Web crean contenido para los navegadores Web clientes, utilizados por los usuarios de la Web, como por ejemplo el navegador Web "Internet Explorer" y el "Netscape". Este contenido Web puede incluir código HTML, scripts ejecutados del lado del cliente, imágenes y datos binarios. Estos programas requieren, para su ejecución, de un navegador Web que los soporte. Si bien, de primera instancia, estos programas utilizan el área de dibujo de la ventana del programa navegador, pueden crear otras ventanas.

Ejemplos de estos programas son los Applets de Java y los Web Forms de .NET.

Creación y Manejo de GUI con Ventanas

Si bien un programa basado en Web puede crear ventanas, adicionalmente a la ventana del programa navegador que lo ejecuta, y muchos de los conceptos de creación y manipulación de los elementos de una ventana se aplican casi idénticamente que en los programas basados en Web, los programas Stand-Alone con ventanas son más simples y sólo dependen de las ventanas que ellos mismos crean. Debido a ésto, toda la explicación respecto a la creación y manejo de GUI's con ventanas se realizará para programas Stand-Alone.

Creación de una Ventana

Veremos cómo se crea un programa mínimo, con una única ventana, en Java, C# y C/C++ con API de Windows.

El usar directamente el API de Windows nos ayudará a entender cómo funciona el sistema de mensajería de Windows. El siguiente código corresponde a un programa en C o C++ que crea una ventana completamente funcional.

```
#include <windows.h>
#include <stdio.h>

LRESULT CALLBACK FuncionVentana(
    HWND hWnd, // Manejador (handle) de la ventana a la que corresponde el mensaje.
    UINT uMsg,          // Identificador del mensaje.
    WPARAM wParam,      // Primer parámetro del mensaje.
    LPARAM lParam )     // Segundo parámetro del mensaje.
{
    switch( uMsg )
    {
    case WM_DESTROY:
        // La función 'PostQuitMessage' crea un mensaje WM_QUIT y lo introduce
        // en la cola de mensajes. El parámetro que se le pasa, establece el
        // valor de 'wParam' del mensaje WM_QUIT generado.
        PostQuitMessage( 0 );
```

```
            break;
        default:
            // Todos los mensajes para los que no deseo realizar ninguna
            // acción, los paso a la función 'DefWindowProc', la que realiza
            // las acciones por defecto correspondientes a cada mensaje.
            return DefWindowProc( hWnd, uMsg, wParam, lParam );
        }
        return 0;
    }

    BOOL RegistrarClaseVentana( HINSTANCE hIns )
    {
        WNDCLASS wc;
        wc.style            = CS_HREDRAW | CS_VREDRAW;
        wc.lpfnWndProc      = FuncionVentana;
        wc.cbClsExtra       = 0;
        wc.cbWndExtra       = 0;
        wc.hInstance        = hIns;
        wc.hIcon            = LoadIcon( NULL, IDI_APPLICATION );
        wc.hCursor          = LoadCursor( NULL, IDC_ARROW );
        wc.hbrBackground    = ( HBRUSH )GetStockObject( WHITE_BRUSH );
        wc.lpszMenuName     = NULL;
        wc.lpszClassName    = "Nombre_Clase_Ventana";
        return ( RegisterClass( &wc ) != 0 );
    }

    HWND CrearInstanciaVentana( HINSTANCE hIns )
    {
        HWND hWnd;
        hWnd = CreateWindow(
            "Nombre_Clase_Ventana",
            "Titulo de la Ventana",
            WS_OVERLAPPEDWINDOW,
            CW_USEDEFAULT,
            CW_USEDEFAULT,
            CW_USEDEFAULT,
            CW_USEDEFAULT,
            NULL,
            NULL,
            hIns,
            NULL
            );
        return hWnd;
    }


    int WINAPI WinMain(
        HINSTANCE hIns,     // Manejador (handle) de la instancia del programa.
        HINSTANCE hInsPrev, // Manejador (handle) de la instancia de un programa,
                    // del mismo tipo, puesto en ejecución previamente.
        LPSTR lpCmdLine,    // Puntero a una cadena (char*) conteniendo la línea de
                    // comando utilizada al correr el programa.
        int iShowCmd )      // Un entero cuyo valor indica cómo debería mostrarse
                    // inicialmente la ventana principal del programa.
    {
        // Registro una clase de ventana, en base a la cual
        // se creará una ventana.
        if( ! RegistrarClaseVentana( hIns ) )
            return 0;

        // Creo una ventana.
        HWND hWnd = CrearInstanciaVentana( hIns );
        if( hWnd == NULL )
            return 0;

        // Muestro la ventana.
        ShowWindow( hWnd, iShowCmd ); // Establece el estado de 'mostrado' de la ventana.
        UpdateWindow( hWnd );       // Genera un mensaje de pintado, el que es
                            // ejecutado por la función de ventana respectiva.

        // Se realiza un bucle donde se procesen los mensaje de la cola de mensajes.
        MSG Mensaje;
        while( GetMessage( &Mensaje, NULL, 0, 0 ) > 0 )
            DispatchMessage( &Mensaje );

        // GetMessage devuelve 0 cuando el mensaje extraído es WM_QUIT y
```

```
        // -1 cuando a ocurrido un error. Note que esto produce que el mensaje
        // WM_QUIT nunca sea procesado por la función de ventana.

        // El parámetro 'wParam' del mensaje 'WM_QUIT' corresponde al parámetro
        // pasado a la función 'PostQuitMessage'. Este valor tiene la misma utilidad
        // que el entero retornado por la función 'main' en programas en C y C++ para
        // consola.
        return Mensaje.wParam;
    }
```

En este programa existen dos funciones importantes: El punto de entrada del programa, la función "WinMain", y la función de ventana "FuncionVentana".

La función WinMain es el equivalente, para un programa en Windows, a la función "main" en programas en C y C++ en modo consola. El principal parámetro de esta función es el primero, el manejador de la instancia del programa. Dicho manejador es, en esencia, un número entero utilizado por Windows para ubicar los recursos relacionados al programa. Algunos de los recursos que un programa en Windows puede tener son: Registros de ventanas, textos, imágenes, audio, video, meta-archivos, otros manejadores (a archivos, puertos, impresoras, etc.), etc. El manejador de la instancia es utilizado como parámetro de las funciones del API de Windows donde se involucren, directa o indirectamente, los recursos de una aplicación. El segundo parámetro no es utilizado en programas de 32-bits (Windows 95 y Windows NT en adelante).

La función de ventana (que puede tener cualquier nombre pero con el mismo formato de declaración que el mostrado) es dónde se realiza toda la lógica relacionada a las acciones que una ventana toma en respuesta a los mensajes recibidos, como por ejemplo, los producidos por la interacción del usuario con la ventana. La dirección de esta función de ventana es el principal dato que forma parte del registro, con la estructura WNDCLASS, de una clase de ventana. Todos los mensajes enviados a ventanas, creados con una misma clase, son procesados por la misma función de ventana, la indicada en la estructura WNDCLASS al registrarse dicha clase de ventana.

Todo programa con ventanas en Windows tiene 3 etapas principales:

1. Registro de las clases de ventana que se utilizarán en el programa para crear ventanas.
2. Creación de la ventana principal del programa.
3. Ejecución del bucle de mensajes del programa.

Para crear una ventana se utiliza la función CreateWindow. Los parámetros de esta función indican, en este orden:

1. El nombre de la clase de ventana en base a la que se crea la nueva ventana.
2. El título a mostrarse en la barra superior de la ventana.
3. El tipo y atributos (incluyendo estilo) de la ventana.
4. La coordenada X, en píxeles, de la esquina superior izquierda en la que aparecerá la ventana dentro del área de la pantalla. La constante CW_USEDEFAULT indica que se utilice un valor por defecto.
5. La coordenada Y, en píxeles, de la esquina superior izquierda en la que aparecerá la ventana dentro del área de la pantalla. La constante CW_USEDEFAULT indica que se utilice un valor por defecto.
6. El ancho de la ventana, en píxeles. La constante CW_USEDEFAULT indica que se utilice un valor por defecto.

7. El alto de la ventana, en píxeles. La constante CW_USEDEFAULT indica que se utilice un valor por defecto.

8. El manejador de una ventana padre. Si se pasa NULL (una constante igual a cero) se indica que la ventana no tiene padre.

9. Un manejador de menú o un identificador de ventana (dependiendo de los valores pasados en el tercer parámetro). Si se pasa NULL se indica que no se utilizará esta característica.

10. El manejador de la instancia del programa.

11. Un puntero genérico (void*) a cualquier información extra que el usuario quiera utilizar. Si se pasa NULL, no se utiliza esta característica.

Creada la ventana principal del programa, se ingresa al bucle de procesamiento de mensajes. Dicho bucle realiza, repetitivamente, las siguientes acciones:

1. Retirar un mensaje de la cola de mensajes, mediante la función GetMessage.

2. Mandar a llamar a la función de ventana correspondiente, mediante la función DispatchMessage.

La estructura MSG almacena los mismos datos que recibe una función de ventana como parámetros. La función GetMessage llena esta estructura con la información del mensaje retirado de la cola de mensajes del programa. La función DispatchMessage utiliza esta información para averiguar a qué ventana corresponde el mensaje, cuál es la clase de ventana de dicha ventana, cuál es la función de ventana registrada con esa clase y, finalmente, llama a dicha función de ventana pasándole los datos almacenados en la estructura MSG.

Es importante notar el hecho de que, si bien se dice en la literatura que la función de ventana es llamada por Windows, no es estrictamente así. Tanto DispatchMessage como otras funciones del API de Windows son las que realmente se encargan de llamar a las funciones de ventana, y dichas funciones son llamadas explícitamente desde el programa. Como puede deducirse, es debido a que estas funciones forman parte del API de Windows, que se dice que es Windows quién llama a las funciones de ventana. Un programa en ejecución, en un sistema operativo multitarea como Windows, no puede llamar directamente a una función de otro programa en memoria. Este aspecto está íntimamente relacionado al tema de programación concurrente, por lo que no ahondaremos por ahora más en esto.

Otro hecho importante es el que todo programa en Windows, sea hecho en C, en Java, en C# o en cualquier otro lenguaje de programación (salvo alguna excepción) presenta, escondida o no, esta estructura de trabajo. Todo programa en Windows requiere registrar las clases de ventana que utiliza, crear dichas ventanas y tener un bucle de procesamiento de mensajes. Para ilustrar mejor ésto revisemos ejemplos equivalentes en Java y C#.

El siguiente programa es el equivalente en Java al programa anterior.

```
import javax.swing.*;
import java.awt.event.*;
class MiVentana extends JFrame {
    public MiVentana() {
        setSize(400, 400);
        setTitle("Título de la Ventana");
        setVisible(true);
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                dispose();
                System.exit(0);
            }
        });
    }
```

```
}
class Aplicacion_Minima {
    public static void main(String[] args) {
        MiVentana ventana = new MiVentana();
    }
}
```

En este código, el punto de entrada es el método estático "main", dentro del cual lo único que se realiza es la creación de un objeto de la clase MiVentana que hereda de la clase JFrame (cuyo nombre completo es javax.swing.JFrame), que es la clase base para la creación de ventanas utilizando el paquete (el término utilizado en Java para referirse a una librería) SWING, cuyas clases son accesibles mediante la sentencia "import javax.swing.*". Dentro del constructor de la clase "MiVentana" se configuran las características de la ventana y se establece "qué objeto" será al que se le llame su método "windowClosing" cuando el usuario del programa indique que desea cerrar la ventana.

Ahora bien, relacionemos todo esto con lo visto anteriormente en el programa en C o C++ con API de Windows.

Como es de esperarse, la clase JFrame debe crear en su constructor (o en algún constructor de sus clases base) una ventana llamando a "CreateWindow" (u otra función equivalente). Como ya hemos visto, no se debe poder crear una ventana antes de registrar la clase de ventana en base a la que se creará, por lo que es de esperarse (y realmente sucede así) que el programa intérprete de Java realice este registro antes de llamar a nuestro método "main". Siguiendo el orden de ejecución de los constructores, luego de completada la ejecución del constructor de JFrame se llamará al de nuestra clase "MiVentana", donde modificamos los valores por defecto con los que se creó inicialmente la ventana (por defecto, JFrame crea una ventana en la coordenada [X,Y] = [0,0], con cero píxeles de ancho y cero de alto y con su atributo de visibilidad puesto en "no-visible"). El constructor termina indicando, mediante el método de JFrame "addWindowListener", a qué método de qué clase se llamará cuando la función de ventana, de la ventana creada por JFrame, reciba el mensaje WM_CLOSE. El procesamiento por defecto, realizado por "DefWindowProc", para este mensaje es llamar a la función de API de Windows "DestroyWindow", que es la que realmente destruye la ventana generando, de paso, el mensaje WM_DESTROY. Como puede deducirse, el método "dispose" de JFrame debería estar llamando a DestroyWindow y el método "System.exit" a PostQuitMessage.

Finalmente queda algo muy importante que no es directamente visible en nuestro código: ¿Dónde se ejecuta el bucle de procesamiento de mensajes? Para explicar ésto, debe aclararse que el programa intérprete de Java realiza algunas acciones luego de llamar a nuestro método "main", entre ellas está la de verificar si después de ejecutarse este método se creó o no alguna ventana. Si se creó entonces se entra a un bucle de procesamiento de mensajes del que, como puede esperarse, el programa no sale hasta haberse llamado al método "System.exit". Si no se creó ninguna ventana, el programa intérprete finaliza. Es por esto que la literatura sobre programación con ventanas en Java indica, sin dar mayor detalle, que si se creara alguna ventana en un programa Java Stand-Alone (a éstos se les llama "Aplicaciones Java"), debe de llamarse en algún momento al método "System.exit".

Ahora bien, el siguiente programa es el equivalente en C# del programa anterior.

```
using System;
using System.Windows.Forms;

class MiVentana : Form {
    public MiVentana() {
        Size = new System.Drawing.Size(400, 400);
        Text = "Título de la Ventana";
        Visible = true;
    }
}
class Aplicacion_Minima {
    public static void Main(string[] args) {
        MiVentana ventana = new MiVentana();
        Application.Run(ventana);
    }
}
```

En el código, el punto de entrada es el método estático "Main" (a diferencia de Java, C# ofrece varias sobrecargas posibles: Con y sin parámetros, con y sin valor de retorno), dentro del cuál se crea una objeto de la clase MiVentana que hereda de la clase Form (cuyo nombre completo es System.Windows.Forms.Form), que es la clase base para la creación de ventanas utilizando un ensamblaje (el término utilizado en C# para referirse a una librería) System.Windows.Forms, también llamado Windows Forms, cuyas clases son accesibles mediante la sentencia "using System.Windows.Forms". Dentro del constructor de la clase "MiVentana" se configuran las características de la ventana.

Nuevamente, ¿cómo se relaciona todo ésto con lo visto anteriormente en el programa en C o C++ con API de Windows?

Como es de esperarse, la clase Form actúa en forma muy similar a la clase JFrame de Java, creando una ventana utilizando funciones como CreateWindow, para luego modificar los valores por defecto de creación (a diferencia de Java, dicha ventana es por defecto visible, con una posición [X,Y] y un ancho y alto con valores por defecto) en el constructor de "MiVentana". A diferencia de Java, Form sí contiene una implementación por defecto cuando se desea cerrar la ventana, por lo que no se requiere escribir algún código al respecto. El porqué Java sí lo requiere y C# no, se debe a que Java no tiene forma de saber cuál es nuestra ventana principal, en caso hayamos creado más de una. Por el contrario, C# requiere que se lo indiquemos al llamar al método estático "Application.Run". Como puede deducirse, este método realiza el bucle de procesamiento de mensajes.

Como puede verse, no importa el lenguaje de programación utilizado, o qué tanto dicho lenguaje nos oculte la implementación interna, tras capas de abstracción (en forma de clases por ejemplo), siempre debemos tener claro que dicha implementación debe necesariamente contener los elementos mostrados en el programa en C o C++ anterior, y por tanto, debe utilizar las funciones del API de Windows que ese programa utiliza. En algunos casos, dichos lenguajes de programación ofrecen acceso a elementos de bajo nivel del API de Windows, como los manejadores de las ventanas que crean y utilizan (por ejemplo Visual Basic, a manera de propiedades de algunos de sus controles visuales), de manera que se acceda a cierta funcionalidad que sólo provee dicha API.

Elementos de una Ventana

Un conjunto de programas GUI con ventanas, que comparten el mismo conjunto de librerías para la manipulación de éstas (en Windows, estas librerías forman parte del mismo sistema operativo, y se les llama API de Windows), comparten no sólo un aspecto común, sino un conjunto de elementos gráficos con un comportamiento y uso común. A ésto se le conoce como el "Look And Feel" de dicho entorno GUI.

El hecho de tener elementos de aspecto y comportamiento común reduce la curva de aprendizaje para que un usuario, que ya aprendió a utilizar uno de estos programas, aprenda a utilizar otro que utilice la misma librería GUI.

En Windows, las ventanas tienen los siguientes elementos:

- Un borde.
- Una barra de título.
- Un ícono de sistema.
- Un conjunto de botones de sistema.
- Un menú.
- Una o más barras de herramientas.

✍ Una barra de estado.

✍ Un área de dibujo o “área cliente”.

De estos elementos, sólo el último es obligatorio, y si bien los demás son comunes, algunas ventanas pueden no tenerlos.

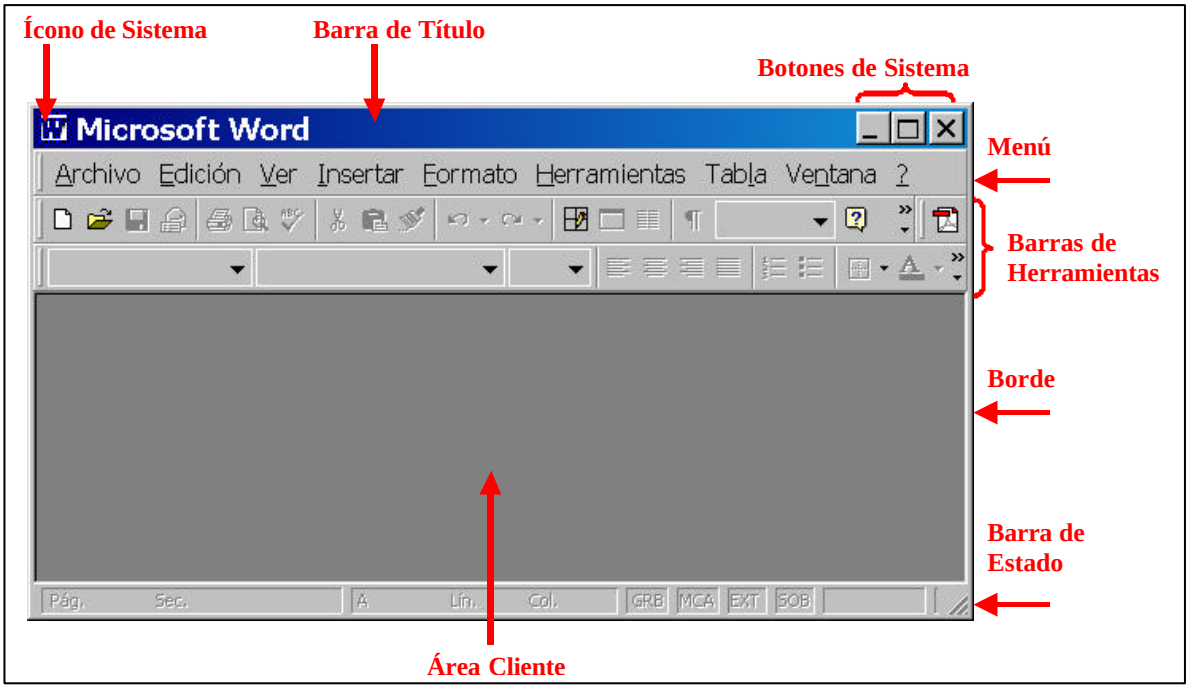

Figura 6.1. Elementos de una Ventana

Interacción con el Entorno

Cada uno de los elementos de una ventana tiene una forma de interacción con el usuario del programa al que pertenece dicha ventana. Internamente, como ya hemos visto, cada una de estas interacciones (el uso del teclado o el ratón) produce un mensaje, que es capturado por el sistema operativo, es colocado por éste en la cola de mensajes del programa correspondiente y finalmente es retirado de dicha cola por el bucle de procesamientos de mensajes de dicho programa. Ya hemos visto como el mecanismo original de procesamiento de estos mensajes, mediante una función de ventana, puede ser abstraído y ocultado al usuario mediante clases que hagan más sencillo este trabajo, como en el caso de Java y C#. En esta sección veremos en mayor detalle, cómo son estas estrategias de abstracción y qué tipos de mensajes se pueden procesar.

Manejo de Eventos con API de Windows

La estrategia de manejo de eventos del sistema operativo Windows, y por tanto del API de Windows, es mediante un sistema de mensajería, similar al sistema de mensajería de correo físico o electrónico.

Un mensaje es el conjunto de datos que el sistema operativo recolecta para un evento dado. Como ejemplo, para el evento click de un botón del ratón, el sistema operativo crea un mensaje que contiene, entre otras cosas, en qué posición de la pantalla y con cuál botón del ratón se hizo click. Dichos mensajes, como cartas de correo, son colocados en las colas de mensajes de las ventanas a las que les corresponden, como si fueran buzones para dichas cartas. Cuando el sistema operativo le da tiempo de CPU a un programa en ejecución, dicho programa verifica si hay mensajes en su cola de mensajes, como cuando

nosotros tenemos un poco de tiempo libre (o cualquier otra excusa para tomarnos un descanso) y vemos nuestro correo electrónico. Si el programa encuentra un mensaje en su cola, lo saca y lo procesa.

La figura 6.2 muestra el ciclo que siguen los eventos procesados en Windows para las aplicaciones que utilizan ventanas.

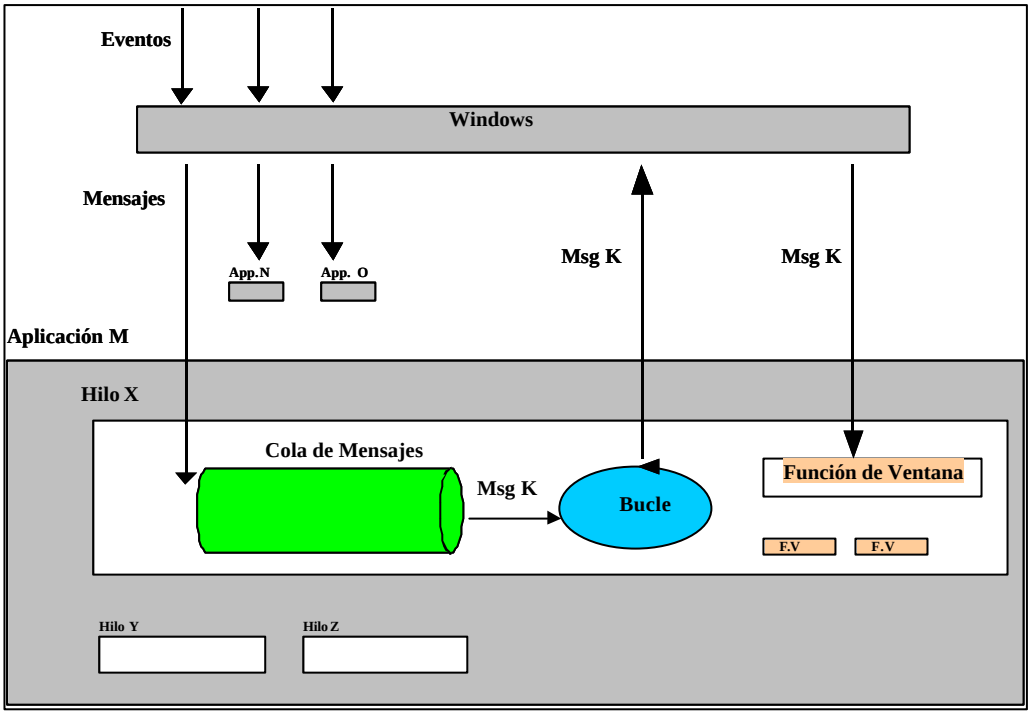

Figura 6.2. Procesamiento de un Evento

La secuencia de pasos seguida es:

- Windows detecta un evento.
- Crea el mensaje respectivo y lo envía a la aplicación involucrada.
- El bucle de procesamiento de mensajes detecta dicho mensaje y solicita a Windows que lo envíe a la ventana adecuada.
- Windows determina la ventana destinataria y averigua a qué clase pertenece.
- En base a la clase determina la función de ventana que le corresponde y envía el mensaje a dicho procedimiento.
- La función de ventana actúa según el mensaje recibido.

Para el procesamiento de los mensajes, toda ventana tiene una “función de procesamiento de mensajes” relacionada, a la que se le llama “función de ventana”. Dicha “relación” entre una ventana y su función de ventana se origina al crearse la ventana utilizando una plantilla de creación llamada “clase de ventana”, la cual debe haberse registrado previamente. Uno de los datos de dicha plantilla es la dirección de la función de ventana que deberá llamarse para procesar los mensajes de toda ventana que se cree utilizando dicha plantilla.

La función de ventana es pues, el área central de trabajo de todo programa desarrollado utilizando el API de Windows. El resto del código suele ser casi siempre el mismo. La función de ventana es la que determina cómo se comportará nuestro programa, nuestra ventana para el usuario, ante cada evento.

Existe un conjunto de mensajes estándar reconocidos por el sistema operativo y que nuestras funciones de ventana pueden manejar. Para cada uno de estos mensajes existe una constante relacionada. En el programa básico mostrado en la sección 3.1, se utilizó una de estas constantes: WM_DESTROY. Al igual que esta constante, definida dentro del archivo de cabecera Windows.h, existe una constante WM_XXX para cada mensaje reconocido por el sistema operativo. Es posible que un programador defina sus propios mensajes simplemente escogiendo un valor fuera del rango que Windows reserva para los desarrolladores de su sistema operativo. El manejo de mensajes definidos por el usuario cae fuera del alcance de esta introducción.

Cuando la función de ventana es llamada para procesar un mensaje, recibe los siguientes datos:

- El handle de la ventana a la que corresponde el mensaje, dado que, como hemos visto, una función de ventana puede utilizarse para procesar los mensajes de más de una ventana, cuando todas éstas fueron creadas utilizando la misma clase de ventana.

- La constante que identifica al mensaje.

- Dos parámetros que contienen información sobre dicho mensaje, o bien contienen direcciones de estructuras reservadas en memoria con dicha información.

El hacer que una ventana reconozca y reaccione a un nuevo mensaje suele consistir en agregar el "case" (al "switch" principal de la función de ventana) para la constante de dicho mensaje con el código que realice el comportamiento deseado. El siguiente código de una función de ventana muestra el manejo de un mensaje correspondiente al ratón y al teclado. El resto del programa no varía respecto al ejemplo anterior.

```
LRESULT CALLBACK FuncionVentana(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    int PosX, PosY;
    char Mensaje[100];
    int CtrlPres, ShiftPres;
    int CodigoTecla, EsTeclaExtendida;

    switch( uMsg )
    {
    case WM_LBUTTONUP:
        PosX = LOWORD(lParam);  // el word menos significativo
        PosY = HIWORD(lParam);  // el word más significativo
        CtrlPres = wParam & MK_CONTROL;
        ShiftPres = wParam & MK_SHIFT;
        sprintf(Mensaje, "X=%d, Y=%d, CtrlPres=%d, ShiftPres=%d",
            PosX, PosY, CtrlPres, ShiftPres);
        MessageBox(hWnd, Mensaje, "Posición del mouse", MB_OK);
        break;
    case WM_KEYUP:
        CodigoTecla = (int)wParam;
        EsTeclaExtendida = ( lParam & ( 1 << 24 ) ) != 0;
        if( EsTeclaExtendida == 1 )
            sprintf(Mensaje, "CodigoTecla=%d (extendida)", CodigoTecla);
        else
            if( ( '0' <= CodigoTecla && CodigoTecla <= '9' ) ||
                ( 'A' <= CodigoTecla && CodigoTecla <= 'Z' ) )
                sprintf(Mensaje, "CodigoTecla=%d (%c)", CodigoTecla,
                    (char)CodigoTecla);
            else
                sprintf(Mensaje, "CodigoTecla=%d", CodigoTecla);
        MessageBox(hWnd, Mensaje, "Tecla presionada", MB_OK);
        break;
    case WM_DESTROY:
        PostQuitMessage( 0 );
        break;
```

```
        default:
            return DefWindowProc( hWnd, uMsg, wParam, lParam );
        }
        return 0;
    }
```

La constante WM_LBUTTONUP corresponde al mensaje producido por el evento de soltar (UP) el botón izquierdo (LBUTTON) del ratón. Para este mensaje, el parámetro wParam se comporta como un bit-flag con información como "estaba la tecla CTRL presionada cuando se produjo el evento". La información de wParam se extrae utilizando constantes definidas en Windows.h, utilizando operaciones booleanas a nivel de bits. El parámetro lParam contiene la posición del ratón al ocurrir el evento. Dicha posición es relativa a la esquina superior izquierda del área cliente de ventana. Los dos bytes menos significativos de lParam contienen la posición X del ratón; los dos bytes más significativos la posición Y, ambas coordenadas medidas en píxeles. Hay 35 mensajes relacionados con el ratón, de los cuales los más comunmente procesados son:

```
WM_LBUTTONDOWN    // Se presionó el botón izquierdo del ratón
WM_LBUTTONUP      // Se soltó el botón izquierdo del ratón
WM_LBUTTONDBLCLK // Se presionó dos veces seguidas el botón izquierdo del ratón
WM_MBUTTONDOWN    // Similar a los anteriores pero para el botón del medio (M)
WM_MBUTTONUP      // ...
WM_MBUTTONDBLCLK // ...
WM_RBUTTONDOWN    // Similar a los anteriores pero para el botón derecho (R)
WM_RBUTTONUP      // ...
WM_RBUTTONDBLCLK // ...
WM_MOUSEACTIVATE // Se pres. un botón del ratón estando sobre una ventana inactiva
WM_MOUSEHOVER     // Como consecuencia a una llamada a la función TrackMouseEvent.
WM_MOUSELEAVE     // Similar al anterior (ver documentación de dicha función)
WM_MOUSEMOVE      // El ratón se mueve sobre una ventana
WM_MOUSEWHEEL     // La rueda del ratón (si la hay) se ha rotado
```

La constante WM_KEYUP corresponde al mensaje producido por el evento de soltar (UP) una tecla del teclado (KEY). Para este mensaje, el parámetro wParam contiene el código de la tecla presionada. Sólo los códigos correspondientes a los números (0x30 al 0x39, '0' al '9') y las letras mayúsculas (0x41 a 0x5A, 'A' al 'Z') coinciden con la tabla ASCII. Toda tecla del teclado tiene un código distinto, existiendo constantes en Windows.h para cada una de ellas. Como ejemplo, podríamos reconocer si la tecla presionada fue F1 utilizando:

```
if ( wParam == VK_F1 ) { ... }
```

El parámetro lParam se comporta como un bit-flag con información como "es una tecla correspondiente al conjunto de teclas extendidas". Hay 15 mensajes relacionados con el teclado, de los cuales los más comúnmente procesados son:

```
WM_KEYDOWN
WM_KEYUP
WM_CHAR
WM_DEADCHAR
```

Manejo de Eventos con Interfaces en Java

En Java, la estrategia de manejo de los mensajes del sistema operativo, corresponde a un patrón de diseño de software conocido como "Patrón Observador".

Bajo este patrón, existen dos objetos: El observador y el sujeto. El sujeto es una fuente de eventos (que pueden corresponder a mensajes del sistema operativo u otros producidos internamente por el programa) susceptibles de ser observados por objetos que cumplen con las características requeridas para ser observadores de dichos eventos.

Para que el objeto observador pueda realizar su trabajo, debe "registrarse" en el sujeto, esto es, notificarle de alguna forma su interés de observar ciertos eventos suyos. De esta forma, cuando el sujeto detecta que ha ocurrido un evento, notifica este hecho a todos los objetos observadores que se registraron para dicho evento. El siguiente programa muestra la aplicación de este patrón de diseño:

```
import java.util.Vector;

class Observador {
    public void notificar(String infoDelEvento) {
        System.out.println("Sucedió el siguiente evento: " + infoDelEvento);
    }
}

class Sujeto {
    private Vector listaObservadores = new Vector();
    public void registrarObservador(Observador ob) {
        listaObservadores.add(ob);
    }
    public void simularEvento(String infoDelEvento) {
        for(int i = 0; i < listaObservadores.size(); i++) {
            Observador ob = (Observador)listaObservadores.get(i);
            ob.notificar(infoDelEvento);
        }
    }
}

class PatronObservador {
    public static void main(String args[]) {
        Observador ob = new Observador();
        Sujeto suj = new Sujeto();
        suj.registrarObservador(ob);
        suj.simularEvento("Evento1");
        suj.simularEvento("Evento2");
    }
}
```

El programa crea un objeto observador y un sujeto observable, para luego simular que dos eventos ocurren. Note que el proceso de registro consiste simplemente en agregar la referencia al objeto observador a una lista del sujeto. Note además que al ocurrir el evento simulado, lo que hace el sujeto es recorrer la lista de referencias a objetos que se registraron como observadores, llamando a un método para cada uno de estos. Ésta es la forma en que el sujeto notifica al observador, llamando a un método de este último. Finalmente note que el método "notificar" de la clase observador es el acuerdo entre ambas partes, sujeto y observador, para que la notificación sea posible, es decir, todo observador de un objeto de mi clase Sujeto debe ser un objeto de la clase Observador o bien heredar de él, de forma que se garantice que dicho método existe.

El ejemplo anterior tiene un problema: Sólo podemos hacer que los objetos de una ClaseX observen los eventos de mi clase Sujeto, si mi ClaseX hereda de Observador. Esto es indeseable si pensamos que lenguajes como Java y C# no soportan herencia múltiple y, muy probablemente, deseáramos que un objeto, cuya clase padre no puedo modificar, escuche los eventos de otro. La solución a éste es aislar los métodos que forman parte del acuerdo en una interfaz. El siguiente programa modifica el anterior de forma que se utilice una interfaz en lugar de una clase:

```
import java.util.Vector;

interface IObservador {
    void notificar(String infoDelEvento);
}

class Sujeto {
    private Vector listaObservadores = new Vector();
    public void registrarObservador(IObservador ob) {
        listaObservadores.add(ob);
    }
    public void simularEvento(String infoDelEvento) {
        for(int i = 0; i < listaObservadores.size(); i++) {
            IObservador ob = (IObservador)listaObservadores.get(i);
            ob.notificar(infoDelEvento);
        }
    }
}

class ObservadorTipo1 implements IObservador {
    public void notificar(String infoDelEvento) {
```

```
        System.out.println("ObsevadorTipo1: Sucedió el siguiente evento: "+
infoDelEvento);
    }
}

class ObservadorTipo2 implements IObservador {
    public void notificar(String infoDelEvento) {
        System.out.println("ObsevadorTipo2: Sucedió el siguiente evento: "+
infoDelEvento);
    }
}

class PatronObservador2 {
    public static void main(String args[]) {
        ObservadorTipo1 ob1 = new ObservadorTipo1();
        ObservadorTipo2 ob2 = new ObservadorTipo2();
        Sujeto suj = new Sujeto();
        suj.registrarObservador(ob1);
        suj.registrarObservador(ob2);
        suj.simularEvento("Evento1");
        suj.simularEvento("Evento2");
    }
}
```

En el ejemplo anterior se tienen dos objetos, cada uno de una clase distinta pero que implementan la interfaz IObservador, que se registran para escuchar los eventos de un tercer objeto, uno de la clase Sujeto. Note que la implementación de la clase Sujeto se basa en la interfaz IObservador y no en una clase. De esta forma se permite que los objetos de cualquier clase que implementen la interfaz IObservador sean utilizados como observadores de un objeto de la clase Sujeto. Ésta última es la estrategia que utiliza Java para sus eventos en ventanas.

En Java, el sujeto es un objeto ventana, una instancia de cualquier clase que herede de JFrame. Esta clase se comunica con una función de ventana internamente, la que procesa un subconjunto de todos los mensajes que se pueden generar con una ventana, llamando a los métodos adecuados de los objetos observadores registrados para dicho objeto ventana. Para esto, JFrame contiene listas de observadores, como datos miembros, para los distintos grupos de mensajes: Una lista para los mensajes de manipulación de la ventana, otra para los mensajes del ratón, otra para los mensajes del teclado, etc. De esta manera, cuando ocurre un evento, el mensaje es tratado por la función de ventana de JFrame, la que a su vez llama al método adecuado de cada objeto observador registrado para dicho mensaje.

En Java, las interfaces como IObservador en nuestro ejemplo, se llaman Listeners, y existe una definida para cada grupo de mensajes. Toda clase cuyos objetos se desea que puedan escuchar un evento de una ventana, debe de implementar la interfaz Listener adecuada. Veamos cómo ésto se refleja, en el caso de los mensajes del ratón y del teclado, en el siguiente código.

```
import javax.swing.*;
import java.awt.event.*;

class MiVentana extends JFrame {
    public MiVentana() {
        setSize(400, 400);
        setTitle("Titulo de la Ventana");
        setVisible(true);
        addWindowListener(new MiObservadorVentana(this));
        addKeyListener(new MiObservadorTeclado());
    }
}

class MiObservadorVentana implements WindowListener {
    MiVentana refVentana;
    public MiObservadorVentana(MiVentana refVentana) {
        this.refVentana = refVentana;
    }
    public void windowActivated(WindowEvent e) {
    }
    public void windowClosed(WindowEvent e) {
    }
    public void windowClosing(WindowEvent e) {
        refVentana.dispose();
```

```
            System.exit(0);
        }
        public void windowDeactivated(WindowEvent e) {
        }
        public void windowDeiconified(WindowEvent e) {
        }
        public void windowIconified(WindowEvent e) {
        }
        public void windowOpened(WindowEvent e) {
        }
    }

    class MiObservadorTeclado implements KeyListener {
        public void keyTyped(KeyEvent e) {
            displayInfo(e, "KEY TYPED: ");
        }
        public void keyPressed(KeyEvent e) {
            displayInfo(e, "KEY PRESSED: ");
        }
        public void keyReleased(KeyEvent e) {
            displayInfo(e, "KEY RELEASED: ");
        }

        private void displayInfo(KeyEvent e, String s){
            String charString, keyCodeString, modString, tmpString;

            char c = e.getKeyChar();
            int keyCode = e.getKeyCode();
            int modifiers = e.getModifiers();

            if (Character.isISOControl(c)) {
                charString = "key character = "
                           + "(an unprintable control character)";
            } else {
                charString = "key character = '"
                           + c + "'";
            }

            keyCodeString = "key code = " + keyCode
                            + " ("
                            + KeyEvent.getKeyText(keyCode)
                            + ")";

            modString = "modifiers = " + modifiers;
            tmpString = KeyEvent.getKeyModifiersText(modifiers);
            if (tmpString.length() > 0) {
                modString += " (" + tmpString + ")";
            } else {
                modString += " (no modifiers)";
            }

            System.out.println(s + "\n"
                               + "    " + charString + "\n"
                               + "    " + keyCodeString + "\n"
                               + "    " + modString);
        }
    }


    class EventosDeVentanas  {
        public static void main(String args[]) {
            MiVentana ventana = new MiVentana();
        }
    }
```

En el código anterior el sujeto observable es el objeto de clase MiVentana creado dentro del método main, y los observadores son dos: Un objeto de la clase MiObservadorVentana, implementando la interfaz WindowListener, y un objeto de la clase MiObservadorTeclado, implementando la interfaz KeyListener. Ambos observadores serán notificados de los eventos de la ventana y del teclado, respectivamente, que es capaz de detectar y/o producir el sujeto. Note además que el sujeto, conceptualmente y en la práctica, no requiere saber sobre la implementación interna de los objetos observadores, lo único que le concierne es que dichos objetos poseen los métodos adecuados para, mediante éstos, poderles notificar que ocurrió un evento del tipo para el que se registraron. Es por ello

que Java utiliza interfaces para definir dicho contrato, los métodos que el objeto observador debe implementar y el objeto observable debe llamar. Note también que los métodos de registro siguen un formato común y forman parte de la interfaz que ofrece el sujeto, en este caso, los métodos addWindowListener y addKeyListener. Como es de esperarse, estos métodos reciben como parámetros referencias a objetos que implementen las interfaces respectivas.

La implementación de la interfaz de un evento particular requiere ser completa, si no lo fuera, la clase sería abstracta y no podríamos pasarle un objeto instanciado de dicha clase al método de registro respectivo. Sin embargo, es posible que no se requiera utilizar todos los métodos de la interfaz para un programa en particular, como es el caso, en el código anterior, de la clase "MiObservadorVentana". Debido a esto, muchas interfaces relacionadas a eventos en Java tienen una clase que las implementa, a la que se le llama "Adaptador". La siguiente clase es el adaptador de la interfaz WindowsListener (definido en el mismo paquete java.awt.event con dicha interfaz):

```
public abstract class WindowAdapter implements EventListener, WindowFocusListener,
WindowListener, WindowStateListener {
    // métodos de la interfaz WindowListener
    public void windowActivated(WindowEvent e) {
    }
    public void windowClosed(WindowEvent e) {
    }
    public void windowClosing(WindowEvent e) {
    }
    public void windowDeactivated(WindowEvent e) {
    }
    public void windowDeiconified(WindowEvent e) {
    }
    public void windowIconified(WindowEvent e) {
    }
    public void windowOpened(WindowEvent e) {
    }

    // métodos de las demás interfaces
    ...
}
```

Los adaptadores le dan una implementación vacía a las interfaces que implementan y son declarados como abstractos únicamente porque se espera que sirvan como clases base para otras clases que sobrescriban los métodos de las interfaces que requieran. Un ejemplo del uso de un adaptador puede verse en el primer código de ejemplo de Java, en la sección 3.1. Creación de una Ventana. En dicho código se utiliza el adaptador WindowAdapter como base de una clase anidada anónima.

Es importante señalar que no existe ningún impedimento para que el sujeto sea a su vez observador de otros sujetos o de sí mismo (como en el ejemplo de la sección 3.1), que un mismo observador pueda observar varios sujetos a la vez (del mismo tipo o de diferente tipo, de uno o más sujetos) y que un mismo evento sea observado por muchos observadores. Para este último caso podríamos, en el ejemplo anterior, haber registrado otros objetos para los mismos eventos (llamando más de una vez a addWindowListener y addKeyListener respectivamente) de manera que cuando dichos eventos ocurran, el sujeto, uno de la clase "MiVentana", llamará en secuencia a los métodos correspondientes de todas los objetos registrados para dicho evento, en el orden en que se registraron.

Así como un objeto se registra para escuchar un evento, también se puede desregistrar. Para ésto existen los correspondientes métodos removeXXXListener, como son removeWindowListener y removeKeyListener.

Note que todos los métodos de una interfaz Listener reciben como parámetro una referencia de una clase XXXEvent, la que encapsula los datos del mensaje y provee de una interfaz para su fácil uso. En el caso de la interfaz WindowListener es WindowEvent, en el caso de la interfaz KeyListener es KeyEvent.

La tabla 6.1 sumariza las clases involucradas en tres tipos de eventos comúnmente manejados en Java:

Tabla 6.1. Clases Relacionadas con Eventos en Java

Evento	Listener	Método de registro en JFrame	Adaptador	Parámetro de los métodos de la interfaz
Ventana	WindowListener	addWindowListener	WindowAdapter	WindowEvent
Teclado	KeyListener	addKeyListener	KeyAdapter	KeyEvent
Ratón	MouseListener	addMouseListener	MouseAdapter	MouseEvent

Manejo de Eventos con Delegados en C#

Los delegados son clases especiales en .NET que manejan internamente una referencia a un método de una clase, de manera que puede llamarla directamente. Es el equivalente a un puntero a función de C o C++, pero sin permitir un acceso directo a dicho puntero o referencia. Los delegados tienen un formato especial de declaración:

```
[Modificadores] delegate <tipo de retorno> <nombre>( [<Lista de Parámetros.>] );
```

La declaración de un delegado es muy similar a la de un método, pero con la palabra "delegate" entre los modificadores y el tipo de valor de retorno. Es importante tener en cuenta que esta declaración corresponde a un "tipo de dato", no a una variable. Dado que esta declaración corresponde a un tipo de dato más, la declaración de variables de este tipo y su inicialización siguen las mismas reglas conocidas para las clases. El siguiente ejemplo muestra la creación de un objeto delegado y su uso.

```
using System;

class OtraClase {
    public static int Metodo3(string sMensaje) {
        Console.WriteLine("    OtraClase.Metodo3 : " + sMensaje);
        return 3;
    }
    public int Metodo4(string sMensaje) {
        Console.WriteLine("    OtraClase.Metodo4 : " + sMensaje);
        return 4;
    }
}

class Principal {

    private delegate int MiDelegado(string sMensaje);
    static event MiDelegado evento;

    private static int Metodo1(string sMensaje) {
        Console.WriteLine("    Principal.Metodo1 : " + sMensaje);
        return 1;
    }
    private int Metodo2(string sMensaje) {
        Console.WriteLine("    Principal.Metodo2 : " + sMensaje);
        return 2;
    }

    public static void Main(string[] args) {
        ////////////////////////////
        // Prueba con delegados

        Console.WriteLine("Prueba con delegados");
        MiDelegado delegado;

        delegado = new MiDelegado(Metodo1);
        Console.WriteLine("Llamando al delegado ...");
        Console.WriteLine("    retorno = " + delegado("mensaje1"));

        Principal refPrincipal = new Principal();
```

```
        delegado = new MiDelegado(refPrincipal.Metodo2);
        Console.WriteLine("Llamando al delegado ...");
        Console.WriteLine("    retorno = " + delegado("mensaje2"));

        delegado = new MiDelegado(OtraClase.Metodo3);
        Console.WriteLine("Llamando al delegado ...");
        Console.WriteLine("    retorno = " + delegado("mensaje3"));

        OtraClase refOtraClase = new OtraClase();
        delegado = new MiDelegado(refOtraClase.Metodo4);
        Console.WriteLine("Llamando al delegado ...");
        Console.WriteLine("    retorno = " + delegado("mensaje4"));

        ////////////////////////////
        // Prueba con eventos
        ////////////////////////////

        Console.WriteLine("Prueba con eventos");

        evento += new MiDelegado(Metodo1);
        evento += new MiDelegado(refPrincipal.Metodo2);
        evento += new MiDelegado(OtraClase.Metodo3);
        evento += new MiDelegado(refOtraClase.Metodo4);
        Console.WriteLine("Llamando al evento ...");
        Console.WriteLine("    retorno = " + evento("mensaje del evento"));
    }
}
```

La salida de este programa es:.

```
Prueba con delegados
Llamando al delegado ...
        Principal.Metodo1 : mensaje1
        retorno = 1
Llamando al delegado ...
        Principal.Metodo2 : mensaje2
        retorno = 2
Llamando al delegado ...
        OtraClase.Metodo3 : mensaje3
        retorno = 3
Llamando al delegado ...
        OtraClase.Metodo4 : mensaje4
        retorno = 4

Prueba con eventos
Llamando al evento ...
        Principal.Metodo1 : mensaje del evento
        Principal.Metodo2 : mensaje del evento
        OtraClase.Metodo3 : mensaje del evento
        OtraClase.Metodo4 : mensaje del evento
        retorno = 4
```

La declaración de una variable tipo delegado y su inicialización es similar a la de cualquier otra clase, excepto por el parámetro de su constructor. Note que si el método pasado como parámetro no es estático, se requiere contar con una referencia a un objeto de la clase que contiene dicho método, al que se desea llamar.

El método pasado como parámetro, al crear un objeto delegado, debe tener el mismo formato de declaración del delegado. Para el ejemplo anterior, tanto los métodos llamados, como la declaración del tipo delegado, reciben como parámetro una referencia a un objeto System.String y retornan un valor System.Int32.

Luego de inicializar una variable de tipo delegado, su uso es igual al de un puntero a función de C o C++, utilizando el nombre de la variable como si se tratara del nombre de un método.

Los objetos delegados utilizados en el código anterior, sólo nos permiten llamar a un único método a la vez. Sin embargo, es posible utilizar un objeto delegado enlazado con otros objetos delegados, de forma que se pueda llamar a más de una función. A este tipo de delegado se le llama **MulticastDelegate**. El uso de este tipo de delegados cae fuera del alcance del presente curso.

Sin embargo, como puede verse en el código anterior, una forma simple de llamar a un grupo de métodos es utilizando la palabra clave event. El formato de declaración de una variable event es:

```
[Modificadores] event <Nombre del Delegado> <Nombre>;
```

Esta declaración sólo puede ir en el ámbito de una clase, no dentro de un método. Esto se debe a que, a pesar de parecerse a la declaración de una variable, al agregarle la palabra event a la declaración, la sentencia se expande al ser compilado el código, generándose la declaración de una variable delegado, del tipo puesto en la declaración, y dos métodos, add_XXX y remove_XXX (donde XXX es el nombre del tipo del delegado). Para el código anterior, esta expansión sería de la forma:

```
private static MiDelegado evento = null;
private static MiDelegado add_MiDelegado(...) {}
private static MiDelegado remove_MiDelegado(...) {}
```

Estos métodos add y remove son llamados automáticamente cuando se utilizan los operadores '+' y '-', respectivamente, con la variable evento. Debido a esto, la variable declarada con la palabra event no requiere ser inicializada.

En .NET se utilizan los conceptos de delegado y evento para manejar la respuesta a la interacción del usuario con las ventanas del programa. Para cada tipo de evento que el usuario pueda generar, existen en las clases de .NET propiedades que encapsulan variables event, así como los tipos de delegados correspondientes. El siguiente código muestra un ejemplo del uso de estos eventos y delegados:

```
using System;
// Al siguiente espacio de nombres pertenecen:
// Form, KeyPressEventHandler, KeyPressEventArgs,
// MouseEventHandler, MouseEventArgs, PaintEventHandler, PaintEventArgs.
using System.Windows.Forms;

class Ventana : Form {
    public Ventana() {
        this.Size = new System.Drawing.Size(400, 400);
        this.Text = "Título de la Ventana";
        this.Visible = true;
        this.MouseUp += new MouseEventHandler(this.Ventana_MouseUp);
        this.MouseLeave += new EventHandler(this.Ventana_MouseLeave);
        this.KeyPress += new KeyPressEventHandler(this.Ventana_KeyPress);
    }
    private void Ventana_MouseUp(object sender, MouseEventArgs e) {
        MessageBox.Show("Evento MouseUp");
    }
    private void Ventana_MouseLeave(object sender, System.EventArgs e) {
        MessageBox.Show("Evento MouseLeave");
    }
    private void Ventana_KeyPress(object sender, KeyPressEventArgs e) {
        MessageBox.Show("Evento KeyPress");
    }
}

class Eventos_Ventana {
    public static void Main(string[] args) {
        Ventana refVentana = new Ventana();
        Application.Run(refVentana);
    }
}
```

En el código anterior, se hace uso de las propiedades MouseUp, MouseLeave, KeyPress y Paint, las que nos permiten acceder a datos miembros internos declarados como event para los tipos de delegado MouseEventHandler, EventHandler, KeyPressEventHandler y PaintEventHandler respectivamente.

Es importante tener en cuenta que las variables declaradas como event son únicamente una forma de simplificar el trabajo con los delegados, cuando se desea tener la posibilidad de llamar a más de un método cuando un evento sucede.

Aunque enfocado en forma distinta a Java, el uso de delegados es realmente la implementación del mismo patrón de diseño, el patrón Observador. La única diferencia está en que, mientras en Java el objeto observador implementa una interfaz para que el sujeto observable le notifique de un evento llamando a los métodos de ésta, en .NET se utiliza un puntero a función encapsulado en una clase especial.

Tipos de Eventos

Los eventos con los que interactúa un programa son comunmente los producidos como consecuencia de un mensaje del sistema operativo, en particular los relacionados con el ratón, el teclado y con el manejo de la ventana.

Un programa también puede definir sus propios eventos o simular los ya existentes como una manera de independizar el programa de las capacidades del sistema operativo subyacente (como es el caso de Java para muchas de sus clases, del paquete Swing, con representación visual).

Un programa también puede disparar eventos al detectar que sucesos no visuales se producen en su entorno, como por ejemplo: El arribo de un paquete de datos por red, la recepción de un mensaje enviado desde otro programa, la baja del nivel de algún recurso por debajo del límite crítico (como la memoria), etc.

Gráficos en 2D

El manejo de los dispositivos gráficos en Windows se realiza mediante la librería GDI32 (Graphics Device Interface). Esta librería aísla las aplicaciones de las diferencias que existen entre los dispositivos gráficos con los que puede interactuar un computador.

El siguiente diagrama muestra el flujo de comunicación entre las aplicaciones en ejecución, la librería gráfica de Windows, las librerías por cada dispositivo y los dispositivos mismos.

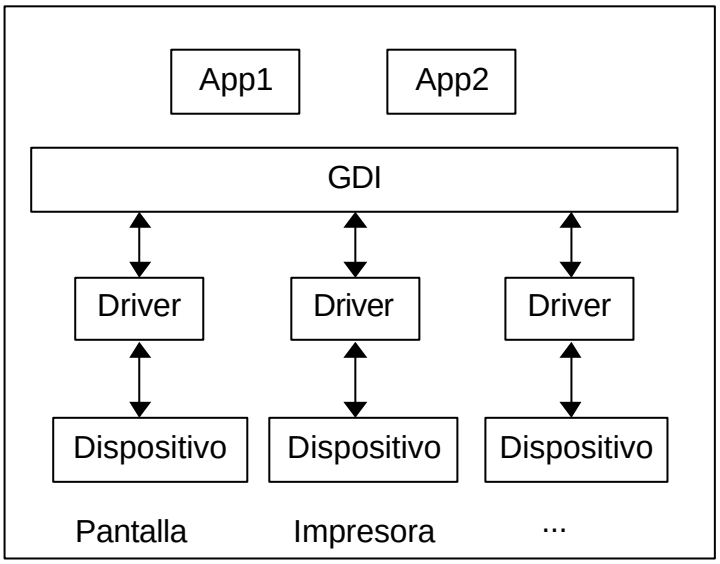

Figura 6.3. Flujo de Comunicación entre Aplicaciones y la Librería GDI

Como se puede apreciar, las aplicaciones interactúan únicamente con la librería GDI. Una vez que una aplicación le indica a GDI con qué tipo de dispositivo desea interactuar, dicha interacción se realiza en forma independiente al dispositivo elegido. Las librerías por cada dispositivo se conocen como Drivers, y deben cumplir la especificación requerida por GDI para que puedan interactuar con él. De esta forma, cada fabricante de un nuevo dispositivo, que desee que éste sea utilizado desde Windows deberá proveer un Driver que sepa cómo manejar su dispositivo y que cumpla las especificaciones de GDI.

Un punto importante es ¿qué sucede cuando una aplicación le indica a GDI con qué tipo de dispositivo desea interactuar? La GDI busca si dicho dispositivo (es decir, su driver) existe, lo carga a memoria si no estuviese ya cargado, crea una entrada en la tabla de recursos para el nuevo recurso a utilizar (el dispositivo), llena dicha entrada y retorna al programa un **handle** al nuevo dispositivo creado. Dicha entrada en la tabla de recursos contiene:

- Información sobre el dispositivo mismo, su tipo, sus capacidades, etc.
- Información sobre su estado actual, lo que puede incluir referencias a otros recursos utilizados cuando la aplicación desea interactuar con el dispositivo.

A dicha información en conjunto se le llama Contexto del Dispositivo (Device Context), por lo que el tipo de su handle relacionado es HDC (Handle Device Context).

De lo anterior se resume que, para que una aplicación pueda interactuar con un dispositivo debe de obtener un HDC adecuado para dicho dispositivo. Al igual que con otros handles, la aplicación deberá liberar dicho HDC cuando ya no requiera trabajar más con él.

Existen 5 formas de dibujo bastante comúnes (no son las únicas):

- Síncrono, donde las acciones de dibujo se realizan en cualquier lugar de la aplicación.
- Asíncrono, donde las acciones de dibujo se realizan en un lugar bien definido de la aplicación.
- Sincronizado, cuando el dibujo asíncrono se "sincroniza" con otras acciones en cualquier lugar de la aplicación.
- En Memoria, donde las acciones de dibujo se realizan en un lugar de la memoria distinta a la memoria de video.
- En Impresora, donde las acciones de dibujo se traducen en comandos enviados a una impresora.

En los tres primeros casos, los HDC que se obtendrían corresponden al área cliente de una ventana, siendo el dispositivo gráfico un monitor de computadora. En el penúltimo caso, el dispositivo gráfico es un espacio de memoria fuera de la memoria de video.

A continuación veremos cómo se realizan los distintos tipos de dibujo para los diferentes lenguajes utilizados. Es importante mantener siempre presente la idea de que, sin importar en qué lenguaje se trabaje, siempre se debe utilizar un HDC para dibujar sobre un dispositivo gráfico.

Dibujo con API de Windows

En esta sección se verá las diferentes formas de dibujo que permite la librería de Windows. Es importante tener en consideración que la implementación en Windows de las librerías de dibujo tanto en Java como C# utilizan internamente esta API para realizar su trabajo.

Funciones de Dibujo

A continuación se explica algunas de las funciones de dibujo de la librería GDI:

La función **"DrawText"** utiliza la fuente de letra, color de texto y color de fondo actual del DC, para dibujar un texto. Su prototipo es:

```
int DrawText(
  HDC hDC,          // handle al DC
  char* texto,    // texto a dibujar
  int longitud,     // longitud del texto
  RECT* area,     // rectángulo dentro del que se dibujará el texto
```

```
  UINT opciones    // opciones de dibujo del texto
);
```

Un ejemplo de uso sería:

```
RECT rc = {20, 50, 0, 0};
DrawText( hDC, "hola", -1, &rc, DT_NOCLIP);
```

El código anterior dibujaría la cadena "hola" sobre el DC referido con el handle "hDC". Utilizamos la opción DT_NOCLIP dado que no nos interesa restringir la salida del texto a un rectángulo específico. Por el mismo motivo sólo especificamos la posición X e Y inicial del texto en la variable "rc". La estructura RECT se define como:

```
struct RECT { long left, top, right, bottom; };
```

La función MoveToEx establece el punto inicial de dibujo de líneas sobre un DC. A partir de dicha posición se realizará dibujos de líneas hacia otras posiciones, con funciones como LineTo. Cada nueva línea dibujada actualiza la posición actual de dibujo de líneas. Los prototipos de MoveToEx y LineTo son:

```
BOOL MoveToEx(
  HDC hdc,            // handle al DC
  int Xinicial,       // coordenada-x de la nueva posición
                     // (la que se convertirá en la actual)
  int Yinicial,       // coordenada-y de la nueva posición
                     // (la que se convertirá en la actual)
  POINT* PosAntigua   // recibe los datos de la antigua posición actual
);

BOOL LineTo(
  HDC hdc,     // handle al DC
  int Xfinal,  // coordenada-x del punto final
  int Yfinal   // coordenada-y del punto final
);
```

Un ejemplo de uso sería:

```
MoveToEx( hDC, 10, 10, NULL );
LineTo( hDC, 40, 10 );
LineTo( hDC, 40, 40 );
LineTo( hDC, 10, 10 );
```

El código anterior dibujaría un triángulo con vértices (10,10), (40,10) y (40,40). La posición actual de dibujo, para subsiguientes dibujos de líneas, quedaría en la coordenada (10,10).

Es importante señalar que todas las acciones de dibujo sobre un DC se realizan en base al sistema de coordenadas del mismo. En el caso de un DC relativo al área cliente de una ventana, el origen de su sistema de coordenadas en la esquina superior izquierda del área cliente, con el eje positivo X avanzando hacia la derecha, y el eje positivo Y avanzando hacia abajo.

Dibujo Asíncrono

Las acciones de dibujo se centralizan en el procesamiento del mensaje WM_PAINT. El siguiente código muestra un esquema típico de procesamiento de este mensaje:

```
case WM_PAINT:
    hDC = BeginPaint(hWnd, &ps);
    // Aquí van las acciones de dibujo
    EndPaint(hWnd, &ps);
    break;
```

La función BeginPaint retorna un HDC adecuado para dibujar sobre el área cliente invalidada de una ventana. La función EndPaint libera el HDC obtenido. Para entender el concepto de invalidación, imagine el siguiente caso:

✍ Se tiene en un momento dado dos ventanas mostradas en pantalla. La primera oculta parte de la segunda, como se muestra en la siguiente figura:

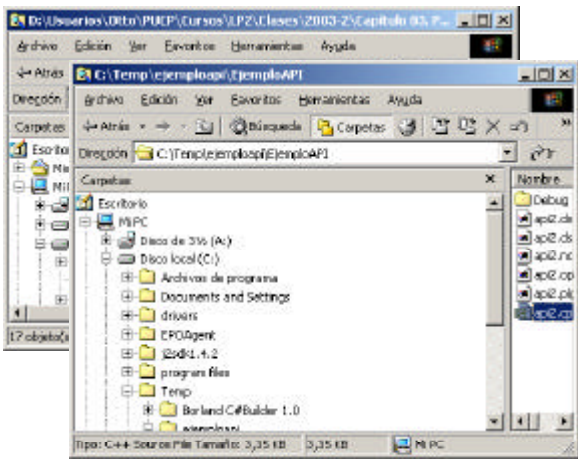

Figura 6.4. Dibujo Asíncrono: Ventana ocultando otra ventana

✍ Luego se mueve la primera ventana de forma que descubre parte o toda el área ocultada de la segunda ventana, como se muestra en la siguiente figura:

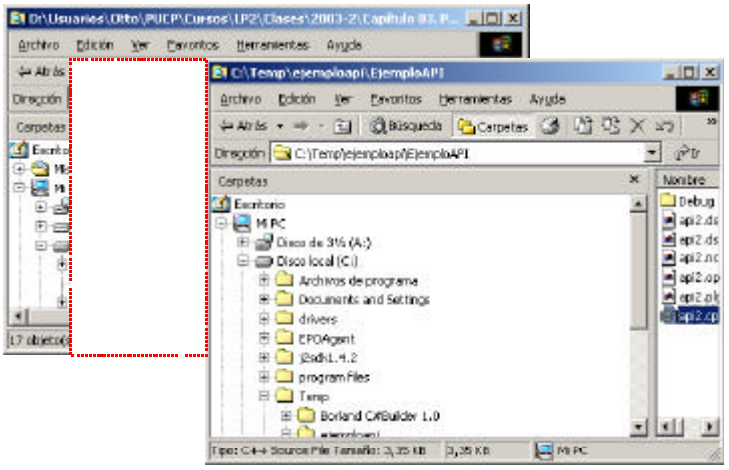

Figura 6.5. Dibujo Asíncrono: Ventana descubriendo otra ventana

✍ El dibujo actual del área descubierta ya no es válido y debe de ser redibujado por el código de la aplicación correspondiente a la segunda ventana, dado que Windows no tiene forma de saber cómo se debe dibujar coda ventana de cada aplicación, sólo las aplicaciones mismas lo saben. Sin embargo, Windows sí reconoce que esta invalidación ha ocurrido, dado que sabe dónde se encuentra cada ventana y cuál está frente a cual, por lo que genera un mensaje WM_PAINT, con la información acerca del rectángulo invalidado, y lo deposita en la cola de mensajes de la aplicación a la que le corresponde dicha ventana invalidada.

✍ Finalmente, cuando el bucle de procesamiento de mensajes correspondiente extraiga y mande a procesar dicho mensaje WM_PAINT, la función de ventana de la ventana invalidada repintará el área de dibujo inválido.

La estructura PAINTSTRUCT es llenada por BeginPaint con información acerca del área invalidada. Esta información podría ser utilizada por el programa para aumentar la eficiencia del código de dibujo, dado que podría repintar solamente el área invalidada y no repintar toda el área cliente. Uno de los datos miembros de dicha estructura es el mismo valor retornado por BeginPaint. La función EndPaint utiliza dicho dato para eliminar el DC.

Los mensajes WM_PAINT son generados en forma automática por el sistema operativo cuando éste sabe que el área cliente de una ventana requiere repintarse. Si al generarse un mensaje WM_PAINT para una ventana, ya existe en la cola de mensajes otro WM_PAINT para la misma ventana, se juntan ambos mensajes en uno solo para un área invalidada igual a la combinación de las áreas invalidadas de ambos mensajes.

También es posible generar un mensaje WM_PAINT manualmente y colocarlo en la cola de mensajes respectiva de forma que se repinte la ventana. La función que hace ésto es:

```
BOOL InvalidateRect(
  HWND hWnd,           // handle de la ventana
  CONST RECT* lpRect,  // rectángulo a invalidar
  BOOL bErase          // flag de limpiado
);
```

El segundo parámetro es el rectángulo, dentro del área cliente, que deseamos invalidar. El tercer parámetro le sirve a la función BeginPaint. Si dicho parámetro es 1, BeginPaint pinta toda el área invalidada utilizando la brocha con la que se creó la ventana (dato miembro hbrBackground de la estructura WNDCLASS) antes de finalizar y retornar el HDC (de forma que se comience con un área de dibujo limpia). Si dicho parámetro es 0, BeginPaint no realiza este limpiado.

El siguiente código muestra el uso de esta función:

```
...
case WM_LBUTTONDOWN:
    iContador++;
    InvalidateRect(hWnd, NULL, TRUE);
    break;
case WM_RBUTTONDOWN:
    for(iBucle = 0; iBucle < 10; iBucle++) {
        iContador++;
        InvalidateRect(hWnd, NULL, TRUE);
    }
    break;
case WM_PAINT:
    hDC = BeginPaint(hWnd, &ps);
    sprintf(szMensaje, "Contador=%d", iContador);
    DrawText(hDC, szMensaje, -1, &rc, DT_NOCLIP);
    EndPaint(hWnd, &ps);
    break;
...
```

El fragmento de código anterior pertenece a una función de ventana que utiliza InvalidateRect. Cuando se presiona con el botón izquierdo del ratón se modifica un contador de forma que el dibujo actual ya no es correcto y debe repintarse. Para el botón derecho se desea que se muestre, como una secuencia animada, cómo se va modificando el contador. Sin embargo, esta secuencia no se muestra, y sólo se ve el valor final del contador en la ventana. Esto se debe al hecho que InvalidateRect “no es” una llamada directa al código en WM_PAINT, sino la colocación de un mensaje WM_PAINT en la cola de mensajes, por lo que sólo al retornar del procesamiento del mensaje actual, WM_RBUTTONDOWN en este caso, el bucle de procesamiento de mensajes podrá retirar el WM_PAINT de la cola de mensajes y procesarlo.

Como puede apreciarse, el dibujo realizado en WM_PAINT es asíncrono respecto a la solicitud de dibujo realizada en WM_RBUTTONDOWN. Este tipo de dibujo es adecuado para dibujos estáticos o de fondo, pero no para secuencias de animación.

Dibujo Síncrono

Para realizar dibujos desde cualquier otro lugar fuera del procesamiento del mensaje WM_PAINT se utiliza la combinación de funciones:

```
// Obtener un DC para el área cliente de la ventana referida con el handle hWnd
HDC GetDC( HWND hWnd );
// Liberar el DC obtenido con GetDC
int ReleaseDC( HWND hWnd, HDC hDC );
```

Para obtener un HDC relativo al área cliente de una ventana se utiliza la función GetDC. Una vez que se han finalizado las acciones de dibujo sobre la ventana, se debe liberar el HDC obtenido llamando a ReleaseDC.

El siguiente segmento de un programa muestra el uso de esta técnica:

```
...
case WM_LBUTTONDOWN:
    hDC = GetDC(hWnd);
    iContador++;
    sprintf(szMensaje, "Contador=%d", iContador);
    DrawText(hDC, szMensaje, -1, &rc, DT_SINGLELINE);
    ReleaseDC(hWnd, hDC);
    break;
case WM_RBUTTONDOWN:
    hDC = GetDC(hWnd);
    for(iBucle = 0; iBucle < 10; iBucle++) {
        iContador++;
        sprintf(szMensaje, "Contador=%d", iContador);
        DrawText(hDC, szMensaje, -1, &rc, DT_SINGLELINE);
        Sleep(100);
    }
    ReleaseDC(hWnd, hDC);
    break;
...
```

A diferencia del caso anterior, el HDC creado puede utilizarse en cualquier lugar del programa. Este HDC debe ser liberado cuando ya no sea requerido.

Dibujo Sincronizado

El dibujo sincronizado permite realizar modificaciones al estado del dibujo en cualquier parte del programa, concentrando el trabajo de dibujo dentro del mensaje de pintado WM_PAINT.

El siguiente segmento de un programa muestra el uso de esta técnica:

```
...
case WM_LBUTTONDOWN:
    iContador++;
    InvalidateRect(hWnd, NULL, TRUE);
    UpdateWindow(hWnd); // acá se fuerza el procesamiento del mensaje WM_PAINT
                        // colocado en la cola de mensajes por InvalidateRect
    break;
case WM_RBUTTONDOWN:
    for(iBucle = 0; iBucle < 10; iBucle++) {
        iContador++;
        InvalidateRect(hWnd, NULL, TRUE);
        UpdateWindow(hWnd);
        Sleep(100);}
    break;
case WM_PAINT:
    hDC = BeginPaint(hWnd, &ps);
    sprintf(szMensaje, "Contador=%d", iContador);
    DrawText(hDC, szMensaje, -1, &rc, DT_SINGLELINE);
    EndPaint(hWnd, &ps);
    break;
...
```

Dibujo en Memoria

Cuando el dibujo se debe realizar en varios pasos, el realizarlo directamente sobre la ventana provoca que el usuario del programa vea todo el proceso de dibujo, produciéndose en algunos casos el parpadeo de la imagen. En estos casos es posible realizar la composición del dibujo en memoria, para luego pasar la imagen final a la ventana en una sola acción.

El siguiente segmento de un programa muestra el uso de esta técnica:

```
// Luego de creada la ventana
```

```
hBM_Fondo = (HBITMAP)LoadImage(hInsApp, "Fondo.bmp",
    IMAGE_BITMAP, 0, 0, LR_LOADFROMFILE | LR_DEFAULTSIZE);
hDC_Fondo = CreateCompatibleDC(hDC);
hBM_Original = (HBITMAP)SelectObject(hDC_Fondo, hBM_Fondo);
szMensaje = “Mensaje de Texto";
TextOut(hDC_Fondo, 20, 20, szMensaje, strlen(szMensaje));

ShowWindow( hWnd, iShowCmd );
UpdateWindow( hWnd );

// Luego del bucle de procesamiento de mensajes

SelectObject(hDC_Fondo, hBM_Original);
DeleteDC(hDC_Fondo);
DeleteObject(hBM_Fondo);

// En la función de ventana

HBITMAP hBM_Fondo = 0;
HDC hDC_Fondo = 0;

LRESULT CALLBACK FuncionVentana( ... ) {
    HDC hDC; PAINTSTRUCT ps; char * szMensaje;

    switch( uMsg ) {
    case WM_PAINT:
        hDC = BeginPaint(hWnd, &ps);
        if(hBM_Fondo != 0 && hDC_Fondo != 0)
            BitBlt(hDC, 0, 0, 800, 600, hDC_Fondo, 0, 0, SRCCOPY);
        else {
           szMensaje = "Error al cargar la imagen";
           TextOut(hDC_Fondo, 20, 20, szMensaje, strlen(szMensaje));
        }
        EndPaint(hWnd, &ps);
        break;
        ...
    }
    return 0;
}
```

Luego de creada la ventana se realiza lo siguiente:

- Crear un DC en memoria en base a otro DC, típicamente, en base al DC de una ventana.
- Crear un área de dibujo en memoria, ésto es, un BITMAP.
- Seleccionar el bitmap en el DC en memoria.
- Realizar los dibujos respectivos en el DC en memoria.

Luego del bucle de mensajes, donde la ventana ya fue destruida, se realiza lo siguiente:

- Se restaura el DC en memoria seleccionándole en bitmap con el que se creó.
- Destruir el DC de memoria.
- Destruir el bitmap.

Dado que el DC en memoria y el bitmap se utilizarán en la función de ventana, es conveniente definir sus variables como globales.

Dentro del procesamiento del mensaje WM_PAINT la secuencia de pasos suele ser la siguiente:

- Se obtiene un handle a un DC para el área cliente de la ventana: BeginPaint

- Utilizando dicho handle se llaman a las funciones de dibujo del API de Windows para dibujar sobre la ventana.

- Se libera el DC: EndPaint

Dibujo en Java

Java provee, a partir de su versión 1.2, el paquete SWING que simplifica significativamente el trabajo con ventanas. Esta librería está formada en su mayoría por componentes ligeros, de forma que se obtenga la máxima portabilidad posible de las aplicaciones con ventanas hacia las diferentes plataformas que soportan Java.

El paquete SWING se basa en el paquete AWT que fue desarrollado con las primeras versiones de Java. Para los ejemplos en las siguientes secciones, se realizará dibujo en 2D sobre la clase base para ventanas JFrame de SWING.

Dibujo Asíncrono

El dibujo en una ventana requiere únicamente sobrescribir el método "paint" de la clase "JFrame". Dentro de este método se llama a la implementación de la clase base de paint y luego se realizan acciones de dibujo utilizando la referencia al objeto Graphics recibida. La clase Graphics contiene métodos adecuados para realizar:

- Dibujo de texto.

- Dibujo de figuras geométricas con y sin relleno.

- Dibujo de imágenes.

El escribir instrucciones de dibujo dentro del método paint equivale a hacerlo dentro del "case WM_PAINT" de la función de ventana en API de Windows. En su implementación para Windows, es de esperarse que la clase Graphics maneje internamente un HDC, obtenido mediante una llamada a "BeginPaint". El siguiente programa muestra el dibujo asíncrono.

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

class Ventana extends JFrame {
    public Ventana() {
        setSize(400, 400);
        setTitle("Titulo de la Ventana");
        setVisible(true);
        addWindowListener(new WindowAdapter() { ... });
    }
    public void paint(Graphics g) {
        super.paint(g);
        g.drawString("Hola mundo!!!!", 100, 100);
    }
}

class Dibujo1 {
    public static void main(String args[]) {
        Ventana ventana = new Ventana();
    }
}
```

Dibujo Síncrono

Para el dibujo síncrono se obtiene un objeto Graphics utilizando el método getGraphics de JFrame. Es importante que, si dicho método es llamado muy seguido, se libere los recursos del objeto Graphics obtenido (por ejemplo, el HDC obtenido mediante un GetDC para la implementación en Windows de esta clase) llamando al método "dispose" (que es de suponer debería llamar a ReleaseDC). Éste es un

claro ejemplo donde el usuario debe preocuparse por liberar explícitamente los recursos dado que el recolector de basura puede no hacerlo a tiempo.

El siguiente programa muestra el dibujo síncrono.

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

class Ventana extends JFrame {
    public Ventana() {
        ...
        addMouseListener(new MouseAdapter() {
            public void mouseClicked(MouseEvent e) {
                Graphics g = getGraphics();
                g.drawString("HOLA", e.getX(), e.getY());
                g.dispose();
            }});
    }
}
...
```

En el programa anterior, cada vez que se realice un click con el botón del mouse, estando el mouse sobre el área cliente de la ventana, se dibujará el texto "HOLA" en dicha posición de la ventana.

Dibujo Sincronizado

Para sincronizar un dibujo se llama al método "repaint" de la clase JFrame y se concentra todo el dibujo en la sobrescritura del método "paint" de la ventana. El siguiente programa muestra el uso de "repaint".

```
import ...
class Ventana extends JFrame {
    Point clicPos;
    public Ventana() {
        ...
        addMouseListener(new MouseAdapter() {
            public void mouseClicked(MouseEvent e) {
                clicPos = e.getPoint();
                repaint();
            }
        });
    }
    public void paint(Graphics g) {
        super.paint(g);
        if(clicPos != null)
            g.drawString("Clic en " + clicPos, 100, 100);
    }
}
...
```

En el programa anterior, cada vez que se realiza un click sobre la ventana se actualiza la variable de clase "clicPos" y se invalida la ventana llamando a "repaint". Esto provoca que en algún momento más adelante se ejecute el método "paint" mostrándose un mensaje con la posición donde se realizó el click.

Dibujo en Memoria

Existen varias estrategias para realizar dibujo en memoria en Java, para cada cual un conjunto de clases adecuadas. Una de estas estrategias consiste en utilizar un objeto BufferedImage, el cual crea un espacio en la memoria sobre la cual se puede realizar un dibujo. Esta clase provee un método "getGraphics" que permite obtener un objeto Graphics adecuado para dibujar en esta memoria. Luego de compuesta la imagen en memoria, se puede utilizar el método "drawImage" del objeto Graphics en el método "paint" para dibujar dicha imagen en la ventana.

El siguiente programa muestra esta estrategia de dibujo.

```
import ...
import java.awt.image.*;
class Ventana extends JFrame implements ImageObserver {
```

```
    BufferedImage img;
    public Ventana() {
        ...
        img = new BufferedImage(100, 100, BufferedImage.TYPE_INT_RGB);
        if(img != null) {
            Graphics g = img.createGraphics();
            g.fillOval(0, 0, 100, 100);
            g.dispose();
        }
    }
    public void paint(Graphics g) {
        ...
        g.drawImage(img, 0, 0, this);
    }
}
...
```

Dibujo en C#

En .NET las clases relacionadas con el dibujo sobre ventanas se encuentran dentro del espacio de nombres System.Drawing.

Dibujo Asíncrono

Realizar un dibujo síncrono sobre una ventana consiste en agregar un nuevo delegado al evento "Paint" de la clase Form. Dicho delegado deberá hacer referencia a un método que reciba como parámetro una referencia "object" y una referencia "PaintEventArgs". Ésta última contiene las propiedades y métodos necesarios para realizar un dibujo sobre la ventana.

El siguiente programa muestra un ejemplo de este tipo de dibujo.

```
using System;
using System.Windows.Forms;
using System.Drawing;

class Ventana : Form {
    private Font drawFont = new Font("Arial", 16);
    private SolidBrush drawBrush = new SolidBrush(Color.Black);

    public Ventana() {
        this.Size = new System.Drawing.Size(400, 400);
        this.Text = "Título de la Ventana";
        this.Visible = true;
        this.Paint += new PaintEventHandler(this.Ventana_Paint);
    }
    private void Ventana_Paint(object sender, PaintEventArgs e) {
        Graphics g = e.Graphics;
        g.DrawString("hola", drawFont, drawBrush, 10, 40);
    }
}

class DibujoAsincrono {
    public static void Main(string[] args) {
        Ventana refVentana = new Ventana();
        Application.Run(refVentana);
    }
}
```

Dibujo Síncrono

Para el dibujo síncrono se utiliza el método CreateGraphics de la clase Form desde cualquier punto del programa. Este método retorna un objeto Graphics (podemos suponer que internamente llama a GetDC) con el cual se puede dibujar sobre la ventana. Cuando ya no se requiera utilizar este objeto, el programa debe llamar al método "Dispose" del mismo, de forma que se liberen los recursos reservados por éste en su creación (podemos suponer que libera el HDC interno que maneja mediante un RealeaseDC).

El siguiente programa muestra el uso del dibujo síncrono.

```
using System;
using System.Windows.Forms;
using System.Drawing;

class Ventana : Form {
    private Font drawFont = new Font("Arial", 16);
    private SolidBrush drawBrush = new SolidBrush(Color.Black);
    public Ventana() {
        this.Size = new System.Drawing.Size(400, 400);
        this.Text = "Título de la Ventana";
        this.Visible = true;
        this.MouseUp += new MouseEventHandler(this.Ventana_MouseUp);
    }
    private void Ventana_MouseUp(object sender, MouseEventArgs e) {
        Graphics g = this.CreateGraphics();
        g.DrawString("Hola", drawFont, drawBrush, e.X, e.Y);
        g.Dispose();
    }
}

class DibujoSincrono {
    public static void Main(string[] args) {
        Ventana refVentana = new Ventana();
        Application.Run(refVentana);
    }
}
```

Dibujo Sincronizado

Para sincronizar un dibujo se llama al método “Invalidate” de la clase Form y se concentra todo el dibujo en la sobrescritura del método donde se concentra el trabajo de dibujo asíncrono. El siguiente programa muestra el uso de “Invalidate”.El siguiente programa muestra el uso del dibujo sincronizado.

```
using System;
using System.Windows.Forms;
using System.Drawing;

class Ventana : Form {
    private Font drawFont = new Font("Arial", 16);
    private SolidBrush drawBrush = new SolidBrush(Color.Black);
    private Point clicPos = new Point(0, 0);
    public Ventana() {
        this.Size = new System.Drawing.Size(400, 400);
        this.Text = "Título de la Ventana";
        this.Visible = true;
        this.MouseUp += new MouseEventHandler(this.Ventana_MouseUp);
        this.Paint += new PaintEventHandler(this.Ventana_Paint);
    }
    private void Ventana_MouseUp(object sender, MouseEventArgs e) {
        clicPos = new Point(e.X, e.Y);
        Invalidate();
    }
    private void Ventana_Paint(object sender, PaintEventArgs e) {
        Graphics g = e.Graphics;
        g.DrawString("Clic en " + clicPos, drawFont, drawBrush, 10, 40);
    }
}

class DibujoSincronizado {
    public static void Main(string[] args) {
        Ventana refVentana = new Ventana();
        Application.Run(refVentana);
    }
}
```

Dibujo en Memoria

Al igual que en Java, existen muchas estrategias de dibujo en memoria. El siguiente ejemplo crea un objeto de la clase “Image” que inicialmente contiene un dibujo guardado en un archivo. Dicho objeto crea un área en memoria, inicializada con la imagen leída del archivo, a la que puede accederse mediante un objeto Graphics creado mediante el método estático “FromImage” de la misma clase Graphics.

Cuando dicho objeto Graphics ya no se requiera, el programa debe liberar sus recursos llamando al método “Dispose”. El siguiente programa muestra un ejemplo de este tipo de dibujo.

```
using System;
using System.Windows.Forms;
using System.Drawing;

class Ventana : Form {
    Image img;

    public Ventana() {
        this.Size = new System.Drawing.Size(400, 400);
        this.Text = "Título de la Ventana";
        this.Visible = true;
        this.Paint += new PaintEventHandler(Ventana_Paint);

        img = Image.FromFile("Fondo.bmp");
        Graphics g = Graphics.FromImage(img);
        Font f = this.Font;
        Brush b = Brushes.Black;
        g.DrawString("Es injusto que el coyote nunca alcance al correcaminos",
            f, b, 50, 30);
        g.Dispose();
    }
    public void Ventana_Paint(object sender, PaintEventArgs args) {
        Graphics g = args.Graphics;
        g.DrawImage(img, 0, 0);
    }
}

class DibujoEnMemoria {
    public static void Main() {
        Application.Run(new Ventana());
    }
}
```

Manejo de Elementos GUI

Las ventanas tienen un conjunto de elementos visuales que ocupan parte (o toda) su área cliente y cuyo comportamiento está bien estandarizado y es común a todos los programas que se ejecutan utilizando una misma librería gráfica. Algunos de estos elementos son: botones, cajas de texto, etiquetas, listas de selección, agrupadores, etc.

En esta sección veremos cómo se crean los elementos GUI más comunes, cómo se distribuyen en el área cliente y cómo se manejan los eventos que producen al interactuar el usuario con ellos.

Elementos GUI del API de Windows

En API de Windows, todos los elementos GUI son ventanas. Cada elemento se crea utilizando una clase de ventana preregistrada, y por consiguiente posee una función de ventana ya implementada en alguna de las librerías del API. El siguiente código muestra un ejemplo simple de creación de una ventana con una etiqueta, una caja de texto y un botón.

```
#include <windows.h>
#define ID_TEXTO 101
#define ID_BOTON 102

LRESULT CALLBACK FuncionVentana( HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam
) {
    switch( uMsg ) {
    case WM_COMMAND:
        if( LOWORD( wParam ) == ID_BOTON ) {
            char szNombre[ 100 ];
            HWND hWndBoton = ( HWND )lParam;
            GetDlgItemText(hWnd, ID_TEXTO, szNombre, 100);
            MessageBox(hWnd, szNombre, "Hola", MB_OK );
        }
        break;
```

```
    // Aquí va el resto del switch
    ...
    }
    return 0;
}

BOOL RegistrarClaseVentana( HINSTANCE hIns ) {
    ...
}

HWND CrearInstanciaVentana( HINSTANCE hIns ) {
    ...
}


int WINAPI WinMain( HINSTANCE hIns, HINSTANCE hInsPrev, LPSTR lpCmdLine, int
iShowCmd ) {
    // Creo y registro la ventana principal.
    ...

    // Creo una etiqueta, una caja de texto y un botón.
    HWND hWndLabel = CreateWindow(
        "STATIC", "Ingrese un nombre:", WS_CHILD | WS_VISIBLE, 10, 10, 150, 20,
        hWnd, NULL, hIns, NULL );
    HWND hWndTextBox = CreateWindow(
        "EDIT", "", WS_CHILD | WS_VISIBLE | WS_BORDER, 170, 10, 100, 20,
        hWnd, (HMENU)ID_TEXTO, hIns, NULL );
    HWND hWndButton = CreateWindow(
        "BUTTON", "OK", WS_CHILD | WS_VISIBLE, 10, 40, 100, 20,
        hWnd, (HMENU)ID_BOTON, hIns, NULL );

    // Muestro la ventana.
    ...

    // Se realiza un bucle donde se procesen los mensaje de la cola de mensajes.
    MSG Mensaje;
    while( GetMessage( &Mensaje, NULL, 0, 0 ) > 0 )
        if( TranslateMessage( &Mensaje ) == FALSE )
            DispatchMessage( &Mensaje );

    return 0;
}
```

A los elementos GUI del API de Windows se les llama controles. Como puede observarse, los controles no son más que ventanas cuyos nombres de clase (STATIC, EDIT y BUTTON) corresponden a clases de ventana preregistradas. Dado que estas ventanas deben dibujarse dentro del área cliente de nuestra ventana principal, tenemos que asignarles la constante de estilo WS_CHILD y el octavo parámetro debe ser el identificador de esta ventana padre. El estilo WS_VISIBLE evita que tengamos que ejecutar ShowWindow para cada una de estas ventanas hijas.

El noveno parámetro de CreateWindow puede, opcionalmente, ser un número identificador que distinga dicha ventana hija de sus ventanas hermanas (hijas de la misma ventana padre). Este parámetro es aprovechado en la función de ventana de nuestra ventana principal. En dicha función se agrega una sentencia CASE para el mensaje WM_COMMAND, el cual es generado por diferentes objetos visibles cuando el usuario interactúa con ellos. En particular, cuando presionamos el botón creado, se agrega a la cola de mensajes del programa, un mensaje WM_COMMAND con los dos bytes menos significativos del parámetro wParam iguales al identificador del botón, ID_BOTON. También utilizamos el identificador de la caja de texto, ID_TEXTO, para poder obtener el texto ingresado llamando a la función GetDlgItemText.

En general, la interacción con los controles estándar del API de Windows, así como otros objetos visuales de una ventana, generan mensajes que son enviados a la ventana padre para su procesamiento. De igual forma, dicho procesamiento suele incluir el envío de nuevos mensajes a los objetos visibles, para obtener más información o para reflejar el cambio de estado del programa (internamente, GetDlgItemText envía un mensaje directamente a la función de ventana del control hijo, de manera que ésta devuelva el texto ingresado), en otras palabras, todo se realiza enviando y recibiendo mensajes. Esto tiene la ventaja de unificar la forma de trabajo con ventanas a un modelo simple de envío-recepción de mensajes, pero con

la desventaja de limitar el procesamiento a una sola ventana (comunmente, solo la ventana padre). Esto tiene sentido bajo el enfoque de que, todos los elementos manejados son ventanas, por tanto es de esperarse que sólo la ventana padre esté interesada en los mensajes de sus hijas. El problema sucede cuando queremos encapsular la funcionalidad de una ventana a sí misma para ciertos trabajos, es decir, ¿cómo hacer para que una caja de texto maneje por sí mismo los mensajes que sólo le competen a él, y envíe a la ventana padre el resto?. Veremos que Java y C# solucionan, de diferente forma, estas carencias del enfoque del API de Windows.

Elementos GUI de Java

En Java, los elementos GUI se denominan componentes. El siguiente código muestra una ventana equivalente en Java al código anterior en API de Windows.

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

class VentanaDePrueba extends JFrame {
    private JTextField txt;

    public VentanaDePrueba(String titulo) {
        super(titulo);

        JLabel lbl = new JLabel("Ingrese un nombre:");
        txt = new JTextField(20);
        JButton btn = new JButton("OK");
        btn.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                JOptionPane.showMessageDialog(VentanaDePrueba.this,
                    txt.getText(), "Hola", JOptionPane.INFORMATION_MESSAGE);
            }
        });

        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        cp.add(lbl);
        cp.add(txt);
        cp.add(btn);
    }
}

public class Componentes {
    public static void main(String args[]) {
        VentanaDePrueba ventana = new VentanaDePrueba("Ventana de prueba de
componentes");
        ventana.setSize(400, 300);
        ventana.setVisible(true);
        ventana.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }});
    }
}
```

En el programa anterior, JTextField, Jlabel y JButton son componentes que representan una caja de texto, una etiqueta y un botón respectivamente, al igual que las clases de ventana EDIT, STATIC y BUTTON del API de Windows.

Los elementos GUI de Java se denominan **componentes**. Como puede observarse, los componentes en Java son clases que se agregan a una ventana para ser visualizados. Es importante en este punto hacer una distinción entre lo que son internamente estos componentes, contra lo que son los controles del API de Windows. Mientras que los controles del API de Windows son ventanas, los componentes de Java (a partir de la versión 1.2 del JDK, denominada Java 2) se dividen en dos categorías:

- **Los componentes pesados**. Su comportamiento está supeditado a las capacidades de la plataforma subyacente, en este caso particular, Windows. Estos componentes son JFrame,

JDialog y JApplet. Estos componentes crean ventanas de Windows (o utilizan directamente una ya creada) y las administran internamente, ofreciendo al programador una interfaz más amigable. En otras palabras, por ejemplo, cuando realizamos click sobre una ventana creada con un objeto que deriva de JFrame, el evento que se genera es un mensaje WM_LBUTTONDOWN, el cual es sacado de la cola de mensajes y enviado a una función de ventana que llama a un método de nuestro objeto JFrame, el cual se encarga de llamar al método respectivo de MouseListener para todos los objetos registrados con un llamado a addMouseListener. Además de lo anterior, el dibujo de la ventana, el efecto de maximizado y minimizado, la capacidad de redimensionar la ventana y todos los efectos visuales posibles, son gestionados por las funciones del API de Windows, así como las capacidades ofrecidas por la propia clase JFrame, por ejemplo, al llamar al método setVisible se estaría llamando internamente a ShowWindow del API de Windows. Debido a esta dependencia, estos componentes son denominados pesados.

- **Los componentes ligeros**. Su comportamiento está supeditado a las capacidades ofrecidas por un componente pesado, del cual heredan o dentro del cual se dibujan, y no de la plataforma subyacente, en este caso particular, Windows. Si bien los mensajes producidos por la interacción del usuario siguen siendo producidos por el sistema operativo, el manejo de éstos (en forma de eventos), el dibujo de los componentes y de sus efectos visuales relacionados, están codificados completamente en Java. Estos componentes definen además sus propios eventos. Debido a esta independencia, estos componentes son denominados ligeros. Ejemplos de estos componentes son JButton, JLabel y JTextField. Los componentes ligeros se dibujan sobre el área cliente de componentes pesados y simulan el "Look And Feel" correspondiente, es decir, no crean una ventana child. Este tipo de ventanas, child, se verá mas adelante (sección "Tipos de Ventana")

Por otro lado, la clase JFrame no administra directamente el área cliente de su ventana, sino que delega dicho trabajo a un objeto Contenedor, derivado de la clase Container. Un Contenedor es básicamente un Componente Java con la capacidad adicional de poder mostrar otros Componentes dentro de su área de dibujo. Es por esto que es necesario obtener una referencia a dicho contenedor de la ventana, llamando al método getContentPane, dado que es a dicho contenedor al que deberemos agregarle los componentes que deseamos visualizar.

Los componentes, al igual que una ventana, pueden generar mensajes como respuesta a la interacción del usuario con ellos. En el código anterior, un botón creado con la clase JButton genera el evento Action cuando el usuario, con el ratón o el teclado, presiona dicho botón. Para procesar dicho evento, definimos una clase inner anónima que implementa la interfaz ActionListener, instanciando dicha clase y pasándole la referencia a esta instancia al método addActionListener del botón.

Es interesante notar el hecho de que una clase inner anónima puede ser creada realmente a partir de una clase o de una interfaz, siempre que en éste último caso se implementen todos sus métodos, que para este caso, es uno sólo, actionPerformed. De igual manera, es interesante notar que se ha escogido configurar el objeto observador del evento de cerrado de la ventana desde el método main, no desde el constructor de la ventana. Ambos enfoques son equivalentes.

Dentro del método actionPerformed se hace uso del dato miembro "txt" de tipo JTextField para poder mostrar el mensaje respectivo mediante el método estático showMessageDialog de la clase JOptionPane. Note que el primer parámetro de este método debe ser una referencia a la ventana padre de la ventana que se mostrará, y que dicho parámetro se pasa utilizando la expresión "Ventana.this". Esto se debe a que si pasáramos únicamente this, nos estaríamos refiriendo al objeto anónimo que implementa la interfaz ActionListener.

Finalmente para este código, note que antes de agregar los componentes al content pane, se llama al método setLayout. Esto permite determinar la forma en que los componentes son distribuidos dentro del área cliente del contenedor. El manejo del diseño (layout) de una ventana se tratará más adelante.

Sumarizando las diferencias entre API de Windows y Java, mientras los objetos visuales comunes llamados controles, preimplementados en dicha librería, son básicamente ventanas y las acciones son

manejadas mediante mensajes enviados por estos controles a sus ventanas padres, en Java se trabajan con clases llamadas componentes, las cuales se agregan al contenedor ContentPane de la ventana, y sus acciones son manejadas mediante interfaces. Mientras que los controles del API de Windows tienen una posición fija respecto a la esquina superior izquierda del área cliente de su ventana padre y sus dimensiones son fijas, los componentes de Java no tienen una posición ni dimensión fija, y esto es manejado mediante objetos **Layout** que asisten en el diseño de la ventana. Mientras que en API de Windows los mensajes sólo pueden ser recibidos por ventanas y sólo una ventana, generalmente la ventana padre, es la que recibe los mensajes de los controles, en Java cualquier objeto de cualquier clase que implemente la interfaz correspondiente a un evento puede recibir la notificación del mismo.

Elementos GUI de C#

Los elementos GUI en C# se denominan controles. El siguiente código muestra una ventana equivalente en C# al código anterior en API de Windows.

```
using System;
using System.Windows.Forms;
using System.Drawing;

class Ventana : Form {
    private TextBox txt;

    public Ventana() {
        this.Text = "Prueba de Controles";
        this.Size = new Size(300, 300);

        Label lbl = new Label();
        lbl.AutoSize = true;
        lbl.Text = "Ingrese un nombre:";
        lbl.Location = new Point(10, 10);

        txt = new TextBox();
        txt.Size = new Size(100, lbl.Height);
        txt.Location = new Point(10 + lbl.Width + 10, 10);

        Button btn = new Button();
        btn.Text = "OK";
        btn.Size = new Size(100, lbl.Height + 10);
        btn.Location = new Point(10, 10 + lbl.Height + 10);
        btn.Click += new EventHandler(Btn_Click);

        this.Controls.Add(lbl);
        this.Controls.Add(txt);
        this.Controls.Add(btn);
    }

    private void Btn_Click(object sender, EventArgs args) {
        MessageBox.Show(txt.Text, "Hola");
    }
}

class Principal {
    public static void Main(string[] args) {
        Application.Run(new Ventana());
    }
}
```

A diferencia de Java, dichos controles sí se crean en base a ventanas de Windows y su posición en el área cliente de la ventana padre sí se determina explícitamente. Al igual que Java, existe un objeto que administra la colección de controles dibujados en una ventana: Controls. Al igual que Java, cualquier objeto puede ser notificado de un evento, utilizando un objeto delegado del tipo del evento y agregándolo al evento correspondiente del componente.

Manejo Asistido del Diseño

El diseño o layout de una ventana es la distribución del espacio de su área cliente entre los elementos GUI que contiene y que componen la interfaz que ofrece al usuario. El API de Windows no ofrece herramientas para asistir al programa en el manejo del diseño. Java y C# sí lo ofrecen.

En Java todo contenedor mantiene una referencia a un objeto que implemente la interfaz LayoutManager y que se encarga de determinar la posición y dimensiones de todos los componentes agregados al ContentPane del contenedor. Esto le quita la tarea al programador de especificar por código estos datos por cada componente. Algunos de los tipos de Layout predefinidos son:

- **BorderLayout:** El área del content pane se divide en 5 regiones (norte, sur, este, oeste y centro) colocando un componente agregado en cada región, por lo que sólo se permite mostrar hasta 5 componentes a la vez, independientemente de que se agreguen más.

- **FlowLayout:** Los componentes son colocados en líneas, uno después del otro, como si cada componente fuese una palabra, continuando con la siguiente línea cuando no queda espacio en la línea actual para visualizar completamente dicho componente. Las dimensiones de cada componente son obtenidas de los valores por defecto que cada uno tiene o de las especificadas por el programa, el layout no modifica estas dimensiones.

- **GridLayout:** El área del ContentPane se divide en cuadricula o grilla, donde cada celda tiene las mismas dimensiones. Los componentes son colocados dentro de estas celdas, modificándoles sus dimensiones para que la ocupen completamente.

El siguiente código muestra el uso de estos layouts en una ventana que utiliza un tipo u otro según un parámetro pasado en su constructor.

```
public Ventana(String Nombre) {
       Container cp = getContentPane();
       if(Nombre.equals("BorderLayout")) {
           cp.setLayout(new BorderLayout());
           cp.add(new JButton("CENTER"), BorderLayout.CENTER);
           cp.add(new JButton("EAST"), BorderLayout.EAST);
           cp.add(new JButton("WEST"), BorderLayout.WEST);
           cp.add(new JButton("NORTH"), BorderLayout.NORTH);
           cp.add(new JButton("SOUTH"), BorderLayout.SOUTH);
       } else if(Nombre.equals("FlowLayout")) {
           cp.setLayout(new FlowLayout(FlowLayout.CENTER));
           for(int i = 0; i < 10; i++)
               cp.add(new JButton("Boton-" + i));
       } else if(Nombre.equals("GridLayout")) {
           cp.setLayout(new GridLayout(3, 2));
           for(int i = 0; i < 10; i++)
               cp.add(new JButton("Boton-" + i));
       }
    . . .
```

En C# el manejo del diseño se realiza mediante anclas (anchors) y muelles (docks). Todo control posee las siguientes propiedades:

- Anchor: Determina a qué borde del contenedor se anclará el control. Por ejemplo, si el control se coloca inicialmente a una distancia de 100 píxeles del borde inferior de su ventana, al redimensionarse el anchor modificará automáticamente la posición del control de forma que conserve dicha distancia.

- Dock: Determina a qué borde del contenedor se adosará el control. Por ejemplo, si el control se adosa al borde izquierdo de su ventana, su ancho inicial se conserva pero su alto se modifica de forma que coincida con el alto del área cliente de su ventana. Su posición también se modifica de forma que su esquina superior izquierda coincida con la del área cliente de su ventana.

- DockPadding: Se establece en el contenedor, por ejemplo, una clase que hereda de Form. Determina la distancia a la que los componentes, adosados a sus bordes, estarán de los mismos.

El siguiente código muestra el uso de estas propiedades sobre un botón que es agregado a una ventana.

```
Boton = new Button();
Boton.Text = "Boton1";
Boton.Dock = DockStyle.Top;
Controls.Add( Boton );

Boton = new Button();
Boton.Text = "Boton2";
Boton.Location = new System.Drawing.Point(100, 100);
Boton.Size = new System.Drawing.Size(200, 50);
Boton.Anchor = AnchorStyles.Bottom | AnchorStyles.Right;
Controls.Add( Boton );
```

Tipos de Ventana

En un sistema gráfico con ventanas, dichas ventanas pueden estar relacionadas. Estas relaciones determinan los tipos de ventanas que se pueden crear.

En Windows, existen dos tipos de relaciones entre ventanas:

- **La relación de pertenencia.** Cuando dos ventanas tienen esta relación, una ventana (llamada owned) le pertenece a la otra ventana (llamada owner), lo que significa que:
  - La ventana owned siempre se dibujará sobre su ventana owner. A la ubicación de una ventana con respecto a otra en un eje imaginario Z que sale de la pantalla del computador, se le conoce como orden-z.
  - La ventana owned es minimizada y restaurada cuando su ventana owner es minimizada y restaurada.
  - La ventana owned es destruida cuando su ventana owner es destruida.
- **La relación padre-hijo.** Cuando dos ventanas tienen esta relación, una ventana (llamada hija) se dibujará dentro del área cliente de otra (llamada padre).

La relación de pertenencia se establece con el octavo parámetro de la función CreateWindow, hWndParent. La relación padre-hijo se establece con el tercer parámetrode la función CreateWindow, escogiendo como bit-flag WS_CHILD o WS_POPUP.

Una ventana popup tiene como área de dibujo el escritorio de Windows (Windows Desktop) y puede tener un botón relacionado en la barra de tareas (Windows TaskBar). Como contraparte, una ventana Child tiene como área de dibujo el área cliente de otra ventana, la que puede ser de tipo Popup o Child, y no puede tener un botón relacionado en la barra de tareas.

Las ventanas popup pueden o no tener una ventana owner. Cuando no la tienen se les llama OVERLAPPED. Un ejemplo de una ventana OVERLAPPED es la ventana principal de todo programa con ventanas de Windows. Las ventanas child siempre tienen una ventana owner.

En resumen, las ventanas popup pueden ser owner o owned, mientras que las ventanas child son siempre owned.

Una ventana que contiene una o más ventanas child del mismo tipo, cada una con su propia barra de título y botones de sistema, se le conoce como ventana **MDI** (Multiple Document Interface), donde cada ventana hija suele ser utilizada para manipular un documento. Ejemplos de estas ventanas son los

programas editores de texto como Word. Las ventanas no-MDI son conocidas como ventanas SDI (Single Document Interface).

A las ventanas cuyos elementos permiten mostrar e ingresar información, ésto es, establecer un diálogo con el usuario se les conoce como Cajas De Dialogo. Las cajas de diálogo no son estrictamente un tipo de ventana, su tipo real es Popup, más bien su concepto corresponde a "una forma de manejo" de una ventana. Existen dos formas de mostrar una caja de diálogo: Modal y amodalmente. Una caja de diálogo modal "detiene", por así decirlo, la ejecución del código desde donde se le muestra, una caja de diálogo amodal no. Por ello las cajas de diálogo modales son adecuadas cuando, dentro de un bloque de instrucciones, se requiere pedir al usuario que ingrese alguna información necesaria para seguir con la ejecución del algoritmo implementado por el bloque. Un ejemplo son las ventanas mostradas por los programas al momento de imprimir. En estos casos "no es conveniente" que el usuario del programa pueda interactuar con la ventana principal de forma que modifique los datos que se imprimirán mientras se están imprimiendo, por lo que resulta imprescindible que el procesamiento de los eventos de la ventana principal sea bloqueado. Las cajas de diálogo amodales son adecuadas para mostrar e ingresar información mientras se sigue interactuando con otra ventana, comunmente la ventana principal del programa. Un ejemplo son las barras de herramientas de algunos programas gráficos como CorelDraw, el editor ortográfico de Word, etc.

El API de Windows implementa un conjunto de cajas de diálogo para acciones comunes como seleccionar un color, abrir un archivo, imprimir, etc. A estas cajas de diálogo se le conoce como Cajas de Diálogo Comunes.

En las siguientes secciones se detallará las capacidades del API de Windows manejado desde C/C++, de Java y de la plataforma .NET programada desde C#, para crear los diferentes tipos de ventanas.

Ventanas con API de Windows

Para crear una ventana owner se utiliza el estilo WS_POPUP (o algún estilo que lo incluya, como WS_OVERLAPPED o WS_OVERLAPPEDWINDOW) y se pasa NULL como el manejador de su ventana owner, lo que equivale a decir que no tiene ventana owner. El siguiente código crea una ventana owner:

```
hWndPopupOwner = CreateWindow(
    "ClaseVentanaPopup",
    "Título de la Ventana Owner",
    WS_POPUP | WS_CAPTION,
    100, 100, 200, 200,
    NULL, // no tiene ventana owner
    NULL, hIns, NULL
);
```

Para crear una ventana owned se utiliza el estilo WS_POPUP y se pasa un manejador válido de su ventana owner. El siguiente código crea una ventana owned:

```
hWndPopupOwned = CreateWindow(
    "ClaseVentanaPopup",
    "Título de la Ventana Popup Owned",
    WS_POPUP | WS_CAPTION,
    100, 100, 200, 200,
    hWndPopupOwner, // ventana owner
    NULL, hIns, NULL
);
```

Para crear una ventana child se utiliza el estilo WS_CHILD y se pasa un manejador válido de su ventana owner. El siguiente código crea una ventana owned:

```
hWndChild = CreateWindow(
    "ClaseVentanaHija",
    "Título de la Ventana Hija",
    WS_CHILD | WS_BORDER,
    100, 100, 200, 200,
```

```
    hWnd,   // debe tener una ventana owner
    NULL,hIns,NULL
);
```

La creación y manejo de cajas de diálogo y ventanas MDI con API de Windows va más allá de los alcances del presente documento.

Ventanas en Java

En Java cada nueva herencia de las clases base para la creación de ventanas (JFrame y JDialog) determina una nueva clase de ventana. Un programa puede definir y crear una o más ventanas, de igual o distinto tipo. Sin embargo, al crear una nueva ventana no se establece una relación de parentesco entre ellas, todas se comportan como popups owner.

El siguiente código muestra la creación de una ventana popup en Java desde la ventana principal del programa.

```
class VentanaPopup extends JFrame { ... }
class VentanaPrincipal extends JFrame {
    public VentanaPrincipal() {
        JButton boton = new JButton("Crear Ventana");
        boton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
             VentanaPopup vp = new VentanaPopup();
           }
        }); ...
    } ...
}
```

Java soporta la creación de aplicaciones MDI. La ventana MDI es llamada “backing window” y consiste en una ventana popup con un “Content Pane” del tipo “JDesktopPane”. Las “pseudo-ventanas child” son implementadas con la clase “JInternalFrame”, la que hereda de “JComponent” por lo que, como puede deducirse, no son realmente ventanas. El siguiente código muestra un ejemplo de este uso:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class Ventanas1 extends JFrame {
    public Ventanas1() {
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                dispose();
                System.exit(0);
            }
        });

        JDesktopPane desktop = new JDesktopPane();
        setContentPane(desktop);

        JInternalFrame child = new JInternalFrame();
        child.setSize(100, 100);
        child.setTitle("Ventana hija");
        child.setVisible(true);
        child.setResizable(true);
        child.setClosable(true);
        child.setMaximizable(true);
        child.setIconifiable(true);
        desktop.add(child);
    }

    public static void main(String args[]) {
        System.out.println("Starting Ventanas1...");
        Ventanas1 mainFrame = new Ventanas1();
        mainFrame.setSize(400, 400);
        mainFrame.setTitle("Ventanas1");
        mainFrame.setVisible(true);
    }
```

```
}
```

Las cajas de diálogo en Java se crean mediante clases que heredan de JDialog. El siguiente código muestra el uso de esta clase:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class CajaDeDialogo extends JDialog {
    JTextField texto;
    public CajaDeDialogo(JFrame padre, boolean EsModal) {
        super(padre, EsModal);
        setSize(300,100);

        texto = new JTextField(20);
        JButton boton = new JButton("OK");
        boton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                setVisible(false);
            }
        });

        Container cp = getContentPane();
        cp.setLayout(new GridLayout(3,1));
        cp.add(new JLabel("Ingrese un texto"));
        cp.add(texto);
        cp.add(boton);
    }
    public String ObtenerResultado() {
        return texto.getText();
    }
}

class Ventanas2 extends JFrame {
    JLabel etiqueta;

    public Ventanas2() {
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                dispose();
                System.exit(0);
            }
        });

        etiqueta = new JLabel("Resultado = ...");
        JButton boton1 = new JButton("Mostrar como modal");
        boton1.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                CajaDeDialogo dialogo = new CajaDeDialogo(Ventanas2.this, true);
                dialogo.setVisible(true);
                etiqueta.setText("Resultado = " + dialogo.ObtenerResultado());
            }
        });
        JButton boton2 = new JButton("Mostrar como amodal");
        boton2.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                CajaDeDialogo dialogo = new CajaDeDialogo(Ventanas2.this, false);
                dialogo.setVisible(true);
                etiqueta.setText("Resultado = " + dialogo.ObtenerResultado());
            }
        });

        Container cp = getContentPane();
        cp.setLayout(new GridLayout(3,1));
        cp.add(boton1);
        cp.add(boton2);
        cp.add(etiqueta);
    }

    public static void main(String args[]) {
        System.out.println("Starting Ventanas2...");
        Ventanas2 mainFrame = new Ventanas2();
```

```
        mainFrame.setSize(300, 200);
        mainFrame.setTitle("Ventanas2");
        mainFrame.setVisible(true);
    }
}
```

El programa anterior muestra la diferencia entre utilizar una caja de diálogo modal y una amodal. También muestra una forma de pasar datos desde la caja de diálogo y la ventana que la crea.

Las cajas de diálogo de Java también son llamadas "ventanas secundarias", mientras que las pseudo-ventanas hijas creadas con JInternalFrame son llamadas "ventanas primarias".

Adicionalmente Java provee la clase JOptionPane, la que permite crear cajas de diálogo con funcionalidad común, como por ejemplo, cajas de diálogo con un texto como mensaje y botones YES, NO y CANCEL.

Ventanas en C#

Al igual que en Java, cada nueva herencia de las clases base para la creación de ventanas, Form, determina una nueva clase de ventana. Un programa puede definir y crear una o más ventanas, de igual o distinto tipo. A diferencia de Java, se pueden crear ventanas popups owner y owned.

El siguiente código muestra la creación de dos ventanas popup, una owner y la otra owned.

```
class VentanaPopup : Form {
    public VentanaPopup( Form OwnerForm ) {
        ...
        this.Owner = OwnerForm;
        this.Visible = true;
    }
}
class VentanaPrincipal : Form {
    public VentanaPrincipal() {
        ...
        VentanaPopup vp1 = new VentanaPopup( null );
        VentanaPopup vp2 = new VentanaPopup( this );
    }
    ...
}
```

C# maneja ventajas child sólo como ventanas hijas de una ventanas MDI. La ventana MDI consiste en una ventana con la propiedad IsMDIContainer puesta en "true". Para que una ventana sea child de otra, se establece su propiedad MdiParent con la referencia de una ventana MDI. El siguiente código muestra un ejemplo de este uso:

```
using System;
using System.Windows.Forms;

class VentanaHija : Form {
    public VentanaHija(Form padre) {
        this.SuspendLayout();
        this.Text = "Ventana Hija";
        this.Location = new System.Drawing.Point(10,10);
        this.Size = new System.Drawing.Size(100, 100);
        this.MdiParent = padre;
        this.Visible = true;
        this.ResumeLayout(false);
    }
}

class VentanaPrincipal : Form
{
    public VentanaPrincipal()
    {
        InitializeComponent();
        this.IsMdiContainer = true;
        VentanaHija hija = new VentanaHija(this);
```

```
    }

    void InitializeComponent()
    {
        this.SuspendLayout();
        this.Name = "MainForm";
        this.Text = "Esta es la Ventana Principal";
        this.Size = new System.Drawing.Size(300, 300);
        this.ResumeLayout(false);
    }

    public static void Main(string[] args)
    {
        Application.Run(new VentanaPrincipal());
    }
}
```

Las cajas de diálogo en C# son ventanas con la propiedad FormBorderStyle puesta a FixedDialog. El siguiente código muestra el uso de una caja de diálogo.

```
using System;
using System.Windows.Forms;

class CajaDeDialogo : Form
{
    private TextBox texto;
    public CajaDeDialogo() {
        this.Text = "Caja de Dialogo";
        this.FormBorderStyle = FormBorderStyle.FixedDialog;

        Label etiqueta = new Label();
        etiqueta.Text = "Ingrese un texto";
        etiqueta.Location = new System.Drawing.Point(24, 16);

        texto = new TextBox();
        texto.Size = new System.Drawing.Size(128, 32);
        texto.Location = new System.Drawing.Point(24, 64);

        Button boton = new Button();
        boton.Text = "OK";
        boton.Size = new System.Drawing.Size(128, 32);
        boton.Location = new System.Drawing.Point(24, 112);
        boton.Click += new System.EventHandler(this.buttonClick);

        Controls.Add(etiqueta);
        Controls.Add(texto);
        Controls.Add(boton);
    }
    void buttonClick(object sender, System.EventArgs e)
    {
        //Visible = false;
        Close();
    }
    public string ObtenerResultado() {
        return texto.Text;
    }
}

class VentanaPrincipal : Form
{
    private System.Windows.Forms.Label label;
    private System.Windows.Forms.Button button2;
    private System.Windows.Forms.Button button;
    public VentanaPrincipal()
    {
        InitializeComponent();
    }

    void buttonClick(object sender, System.EventArgs e)
    {
        // caso de una caja de dialogo modal
        CajaDeDialogo dialogo = new CajaDeDialogo();
        dialogo.ShowDialog(this);
        // dado que la ejecución de la instrucción anterior se detiene hasta que la
        // caja de diálogo se cierre, es posible recuperar el valor ingresado
```

```
            label.Text = "Resultado = " + dialogo.ObtenerResultado();
        }

        void button2Click(object sender, System.EventArgs e)
        {
            // caso de una caja de dialogo amodal
            CajaDeDialogo dialogo = new CajaDeDialogo();
            dialogo.Show();
            // dado que la ejecución de la instrucción anterior "NO" se detiene hasta que
            // la caja de diálogo se cierre, el valor recuperado sera el valor que
            // inicialmente tiene la caja de texto de dicha caja de dialogo al crearse,
            // esto es, una cadena vacia
            label.Text = "Resultado = " + dialogo.ObtenerResultado();
        }

        void InitializeComponent() {
            this.button = new System.Windows.Forms.Button();
            this.button2 = new System.Windows.Forms.Button();
            this.label = new System.Windows.Forms.Label();
            this.SuspendLayout();
            //
            // Primer botón
            //
            this.button.Location = new System.Drawing.Point(24, 16);
            this.button.Name = "button";
            this.button.Size = new System.Drawing.Size(128, 32);
            this.button.TabIndex = 0;
            this.button.Text = "Mostrar Como Modal";
            this.button.Click += new System.EventHandler(this.buttonClick);
            //
            // Segundo botón
            //
            this.button2.Location = new System.Drawing.Point(24, 64);
            this.button2.Name = "button2";
            this.button2.Size = new System.Drawing.Size(128, 32);
            this.button2.TabIndex = 1;
            this.button2.Text = "Mostrar Como Amodal";
            this.button2.Click += new System.EventHandler(this.button2Click);
            //
            // Etiqueta
            //
            this.label.Location = new System.Drawing.Point(24, 112);
            this.label.Name = "label";
            this.label.Size = new System.Drawing.Size(128, 24);
            this.label.TabIndex = 2;
            this.label.Text = "Resultado = ...";
            //
            // Agrego los controles
            //
            this.ClientSize = new System.Drawing.Size(248, 165);
            this.Controls.AddRange(new System.Windows.Forms.Control[] {
                        this.label,
                        this.button2,
                        this.button});
            this.Text = "Prueba con Cajas de Diálogo";
            this.ResumeLayout(false);
        }

        public static void Main(string[] args)
        {
            Application.Run(new VentanaPrincipal());
        }
    }
```

El programa anterior muestra la diferencia entre utilizar una caja de diálogo modal, con ShowDialog, y una amodal, con Show. También muestra una forma de pasar datos desde la caja de diálogo y la ventana que la crea.

Dado que es común validar la forma en que fue respondida una caja de diálogo modal, se provee la propiedad DialogResult, cuyo tipo es el enumerado DialogResult con los siguientes valores: Abort, Cancel, Ignore, No, None, OK, Retry, Yes. Esta propiedad se establece automáticamente en algunos casos (cuando se cierra la ventana, se establece a Cancel) o manualmente desde eventos programados.

Notas sobre Localización de Archivos

Los programas en ejecución (llamados procesos) tienen siempre un *directorio de trabajo*. Este directorio sirve para poder ubicar, de forma relativa, otros archivos (por ejemplo, para abrir dichos archivos). De esta forma un proceso no requiere utilizar siempre el directorio absoluto para acceder a archivos que se encuentran en su mismo directorio de trabajo o en algún otro directorio cercano a éste, como sus subdirectorios.

Por defecto, un proceso obtiene su directorio de trabajo heredándolo de su proceso padre (el que lo arrancó), a menos que este último indique explicitamente otro directorio. Por ejemplo, la siguiente línea de comando corresponde a una ventana de comandos (o shell) desde donde se arranca un programa Abc:

```
c:\prueba> c:\temp\Abc.exe ?
```

Note que el programa se encuentra en el directorio "c:\temp" mientras que el directorio de trabajo del shell es "c:\prueba". Luego, el nuevo proceso Abc creado tendrá como directorio de trabajo "c:\prueba", pues lo hereda del shell. Si se desea arrancar un programa con un directorio de trabajo distinto al del shell, se puede utilizar el comando start:

```
c:\prueba> start /Dd:\otroDirectorio c:\temp\Abc.exe ?
```

Es fácil ver cual es el directorio de trabajo actual del shell (en Windows lo indica el mismo prompt, en Linux se puede consultar con un comando, por ejemplo pwd) y cambiarlo (utilizando un comando como cd). De igual forma, utilizando las llamadas a las funciones adecuadas del API del sistema operativo, cualquier proceso puede modificar su directorio de trabajo durante su ejecución.

Para programas diferentes a los shells, desde donde también es posible arrancar otros programas, el directorio de trabajo actual puede no ser claro, por lo que heredarlo puede no ser lo que el usuario espera. Por ejemplo, al arrancar un programa desde el explorador de Windows haciendo un doble-click sobre el nombre del archivo ejecutable, el usuario espera que se inicie dicho programa teniendo como directorio de trabajo inicial el mismo directorio donde se encuentra el archivo ejecutable, independientemente de cual sea actualmente el directorio de trabajo del explorador de Windows. Este comportamiento puede modificarse creando accesos directos a dichos programas ejecutables y configurándolos para especificar un directorio distinto como directorio de trabajo inicial.

No todos los archivos a los que un proceso requiere acceder se ubican utilizando el directorio de trabajo como referencia. Por ejemplo, en el caso de las librerías, se suelen utilizar variables de entorno para definir conjuntos de directorios de búsqueda. Por ejemplo, para las DLL nativas de Windows, se utiliza la variable de entorno PATH, mientras el compilador e intérprete de Java utilizan la variable de entorno CLASSPATH.