Universidad Nacional del Litoral
**Facultad de Ingeniería y Ciencias Hídricas**
Departamento de Informática

# Programación Orientada a Objetos

*Asignatura correspondiente al plan de estudios de la carrera de Ingeniería Informática*

## Anexo
## Desarrollo de un proyecto con wxWidgets

*Ing. Pablo Novara*
*Ultima revisión: 13/11/2009*

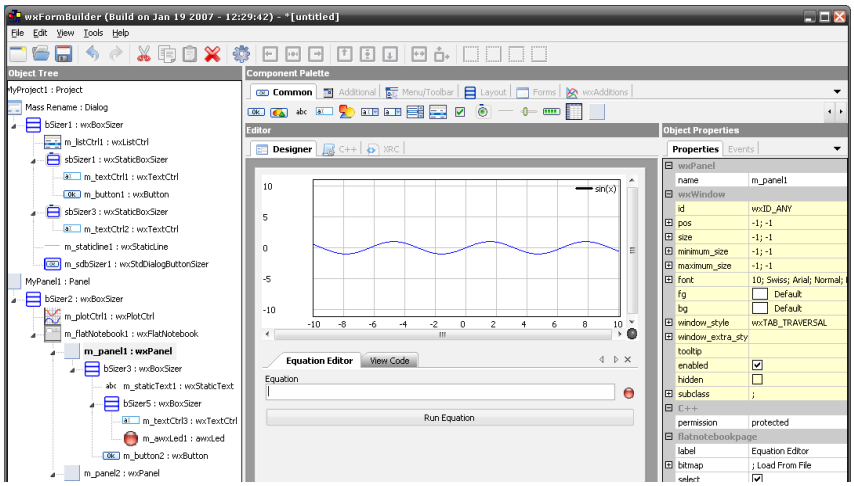
# Tutorial: Desarrollo de un proyecto con wxWidgets

En este documento se cubren los aspectos escenciales del desarrollo de un proyecto ejemplo (una agenda de teléfonos) con C++ y la biblioteca wxWidgets, abarcando tanto el diseño de las clases que modelan el problema, como el desarrollo de la interfaz visual y otras consideraciones generales.

## Introducción: ¿Qué es y cómo lo uso?

wxWidgets es una biblioteca que facilita entre otras cosas desarrollar programas con interfases de ventana. Contiene clases para construir ventanas, paneles, botones, imagenes, cuadros de texto, listas desplegables, cuadros de selección de archivos, etc. Es decir, contiene todos los elementos básicos que suelen incluir la bibliotecas de este tipo o los entornos de desarrollo visual. Como ventaja frente a otras alternativas, es libre y gratuita, orientada a objetos y multiplataforma. Como valor agregado, wxWidgets también ofrece clases para el manejo de procesos, comunicaciones, archivos y cadenas, funciones de internacionalización, etc. que van más allá de lo simplemente visual.

Dado que wxWidgets es una biblioteca, y no una IDE, ni una herramienta de diseño visual, ni nada más que mucho código, utilizarla requiere, en principio, escribir más código y sólo escribir más código. Por ejemplo, para crear una aplicación y tener una ventana, hay que codificar al menos dos clases, una que herede de la clase wxApplication y otra que herede de la clase wxFrame. La clase wxFrame, por ejemplo, tiene implementado todo lo relacionado a la visualización de la ventana y el manejo de eventos. El programador debe implementar en su clase (que hereda de wxFrame) un constructor que coloque dentro de la ventana los componentes que necesite (cree instancias de los objetos que representan los botones, cuadros de texto, etc.) y los comportamientos de la ventana para los distintos eventos (por ejemplo presionar un botón dentro de la misma) e indicar qué evento se asocia con qué método. Esto se debe hacer programando (escribiendo código) y no “dibujando” (arrastrando objetos con el mouse desde una paleta de componentes), y muchas tareas básicas resultan repetitivas. Para solucionar este problema, existe herramientas externas a la biblioteca que permiten “dibujar” las ventanas sin tener que introducir ninguna línea de código, y luego generan de forma automática los fuentes de las clases que se necesitan para obtener por resultado lo que se “dibujó”. En este tutorial se va a utilizar una de ellas: wxFormBuilder (wxfb de ahora en más).




wxfb va a generar dos archivos (un .cpp y un .h) con el código para crear las ventanas como objetos. El programador debe generar por herencia sus propias clases a partir estas en otros archivos.

Nunca se debe trabajar directamente sobre las clases generadas por wxfb porque ante cualquier cambio en el diseño de las ventanas se volveran a regenerar estos archivos y se perderan la modificaciones realizadas a los mismos. Entonces, se deben crear nuevos archivos, incluir los .h generados por wxfb y definir clases que herenden de las “dibujadas”. En estas clases sólo se deben reimplementar los métodos para los eventos que nos interesen. Estos métodos serán métodos virtuales en la clase padre (no puros, así que no es necesario redefinirlos a todos) definidos automáticamente por wxfb.

Por otro lado, como toda biblioteca, para utilizarla hay que indicarle al compilador qué archivos enlazar, donde encontrar las cabeceras y binarios, qué directivas de preprocesador predefinir, etc. Para evitar hacer todo esto manualmente, ZinjaI incluye una plantilla de proyecto con todas estas configuraciones ya especificadas para utilizar wxWidgets tanto en Windows como en GNU/Linux.

En resúmen, en este tutorial se va a utilizar ZinjaI y wxfb para construir una aplicación visual que utilice wxWidgets, de forma fácil y rápida, evitando las tareas más rutinarias, pero sin perder de vista qué cosas se están haciendo automáticamente para comprender mejor el desarrollo y funcionamiento de la aplicación. El proceso es el siguiente:

1) Programar las clases que resuelven el problema independientemente de la interfaz (probándolas en pequeños programas clientes de consola).
2) “Dibujar” las ventanas con wxfb, decir que eventos interesan y generar automáticamente el código fuente para esos diseños.
3) Heredar de las clases generadas y sobreescribir los métodos virtuales (clicks en botones, cambios en cuadros de texto, etc) invocando a las clases y métodos del paso 1.
4) Compilar, ejecutar, depurar...

**Proyecto ejemplo: Una agenda básica**

Para introducir los conceptos teóricos y prácticos básicos para usar wxWidgets, en este documento se propone presentar a modo de tutorial los pasos para el desarrollo de una agenda básica. En este ejemplo se va a utilizar ZinjaI para construir el proyecto y wxfb para dibujar las interfases visuales. Además, se va a ejemplificar primero la construcción de un conjunto de clases completas y consistentes para representar el problema, independientes de la interfaz, para luego integrar con poco esfuerzo en las ventanas desarrolladas con wxWidgets.


Entre los archivos que acompañan al tutorial se encuentra el proyecto completo compilado para ver su funcionamiento (agenda.exe en Windows, agenda.bin en GNU/Linux).

**Ideas del paradigma: Objetos y eventos**

Como ya se dijo, wxWidgets está programada utilizando el paradigma de la orientación a objetos, por lo que es en realidad un gran conjunto de clases. Pero, como la gran mayoría de las bibliotecas de componentes visuales, el comportamiento de la aplicación está guiado por eventos.

Para mencionar ejemplos claros, los objetos representan ventanas, botones, cuadros de texto, imágenes, etiquetas, listas, grillas, barras de progreso, barras de desplazamiento, etc; mientras que ejemplos de eventos pueden ser: hacer click en un botón, modificar el contenido de un cuadro de texto, cerrar una ventana, mover el ratón, seleccionar un elemento de una lista, etc.

Un programa que utiliza esta combinación de paradigmas usualmente comienza realizando todas las inicializaciones necesarias (carga bases de datos, lee configuraciones, etc), crea una ventana inicial y luego le cede el control a la biblioteca. Ésta se encargará de devolverle el control al programador cuando ocurra un evento. Es decir, la biblioteca estará esperando alguna acción de el usuario. Si el usuario hace click en un botón, la biblioteca invocará el método de la ventana que esté asociado con ese botón. Allí el programador escribe el código que constituye la reacción de la aplicación a ese botón (por ejemplo, abrir otra ventana y mostrar un registro) y luego devuelve el control a la biblioteca hasta que ocurra otro evento. Esto significa que la biblioteca es la que gestiona el “bucle de eventos” y el programador sólo debe preocuparse por reaccionar ante los mismos, pero no por averiguar cuales, cuando y cómo ocurren.

**Instalando las herramientas: Manos a la obra**

En este tutorial vamos a utilizar ZinjaI, wxWidgets y wxfb. Las direcciones donde conseguir estos proyectos son: http://zinjai.sourceforge.net, http://www.wxwidgets.org y http://wxformbuilder.org respectivamente.

Si utiliza Microsoft Windows, no necesita descargar e instalar wxWidgets ya que la misma se encuentra dentro de ZinjaI (pero debe tener cuidado de seleccionarla durante la instalación de ZinjaI, ya que por defecto no se instala). wxfb sí debe ser descargado e instalado por separado.

Si utiliza GNU/Linux, debe descargar las tres cosas por separado. Conviene descargar ZinjaI desde la página, pero utilizar el gestor de paquetes de su distribución (aptitude, synaptic, yum, yast, etc) para los otros dos.

Si luego de instalar wxfb, ZinjaI no lo encuentra, debe configurar la ruta de instalación en la pestaña “Rutas 2” del cuadro de Preferencias (Archivo->Preferencias). Sin embargo, ésto no debería ocurrir en la mayoría de los casos si la instalación es estándar.

Hay un detalle importante a tener en cuenta: se pueden descargar/compilar dos versiones de wxWidgets: ansi y unicode. La versión ansi utiliza las cadenas de caracteres estándar, mientras que la versión unicode utiliza una codificación especial donde cada caracter ocupa dos bytes (esto se hace para permitir una gran variedad de acentos y caracteres extranjeros). Si se utiliza la versión unicode, no se puede convertir fácilmente entre las cadenas wxString y las cadenas de c u objetos string (verá en otros tutoriales macros como _T, o clases como wxConv), por lo que se recomienda para empezar instalar la versión ansi. Los desarrolladores de wxWidgets ya estan trabajando para que en las próximas versiones (3.0 en adelante) esta distinción ya no exista.

**Desarrollo del código base: Cómo no pensar en la interfaz**

Es una buena práctica desarrollar de forma lo más independiente de la interfaz posible la lógica del problema a resolver. Es decir, diseñar e implementar el conjunto de clases que modelan el problema a resolver por el software sin estar condicionados por el tipo de interfaz de usuario o por una biblioteca en particular; las mismas clases deben poder utililzarse desde un cliente de consola, desde un programa gráfico, o en un servidor web. Una vez desarrollado este código base, en los eventos de la interfaz gráfica sólo deberemos completar unas pocas líneas invocando a las clases y métodos desarrollados previamente.

Esta sección explica el diseño de las clases del ejemplo. Si bien es recomendable leer el ejemplo completo, si sólo le interesa aprender a integrar la interfaz visual, puede saltearla por completo.

La aplicación que queremos desarrollar tiene por finalidad almacenar datos (nombre, dirección, email, teléfono, etc.) sobre un grupo de personas. El usuario debe poder buscar fácilmente la información, cargar nuevas personas, editar la información de una persona, etc. Para ello, comenzaremos por plantear dos clases básicas:

- Persona: representa una persona, tiene sus datos, y métodos que se encargan de cargarlos, validarlos y devolverlos cuando se los pide una rutina cliente.
- Agenda: se encarga de manejar una colección de personas, actúa como base de datos o contenedor para los objetos de tipo Persona. Tiene métodos para agregar, buscar, modificar y quitar personas, y puede manejar tambien la escritura y lectura desde un archivo.

Definimos que para cada persona se guardarán los siguientes atributos: nombre, apellido, telefono, dirección, localidad, email y fecha de nacimiento. La mayoría de los campos son cadenas de texto. Aquí se plantea una disyuntiva importante: Para que el objeto pueda guardarse directamente en un archivo binario no debe contener atributos dinámicos (como la clase string, porque se guardan los punteros y no los contenidos), pero para que el objeto sea facil de manipular conviene que los atributos sean instancias de la clase string. La solución que aquí se propone consiste en usar una estructura auxiliar. Esta estructura se usa como intermediaria entre la clase persona y el archivo: contiene cadenas de caracteres al estilo c para poder leer y escribir sin problemas. Agregaremos en la clase persona métodos para realizar la lectura y escritura que utilicen esta estructura temporal y realice la conversión adecuada.

Para definir concretamente la interfaz que estas clases exponen a las rutinas cliente, podemos pensar en las operaciones más comunes que quisieramos poder realizar, para identificamos un conjunto de métodos que no pueden faltar. Para esto, podemos imaginar que estamos desarrollando el programa cliente de estas clases y buscar una interfaz  para las mismas lo más cómoda posible (preguntarnos ¿como nos gustaría acceder a las funcionalidades desde el cliente?). La acciones básicas son:

- Ingresar los datos de una nueva persona: Para ingresar estos datos, vamos a necesitar crear una instancia de Persona, cargar los datos que el usuario ingrese y luego agregarlo en la colección de personas que tiene la clase Agenda. Entonces persona deberá presentar métodos para cargar los datos (y/o un constructor) y realizar las validaciones que sean necesarias. Una forma de pensar esto puede ser incorporar un método que valide todos los campos juntos y devuelva la lista errores si alguna validación falla. Además, agenda deberá tener un método para agregar este objeto en la base de datos.

```
Agenda mi_agenda;
Persona nueva_persona( “Juan” , ”Perez”, “343-40839492”,
                       “9 de Julio 2387”, “Santa Fe”,
                       “jperez@fich.unl.edu.ar”, 10, 09, 85 );
string errores =  nueva_persona.ValidarDatos();
```

```
if (errores.size()==0) {
   mi_agenda.Agregar(nueva_persona);
} else {
   cout<<"Los datos no son correctos:"<<endl;
   cout<<errores<<endl;
}
```

• Modificar los datos de una persona: Para modificar un campo de una persona sería deseable no tener que volver a cargar todos los demás, por lo que primero necesitamos una forma de recuperar todos los datos de una persona. Para esto puede ser cómodo sobrecargar el operador [], así accederíamos a una persona utilizando el objeto Agenda como si fuese simplemente un arreglo. Luego, podemos modificar el objeto persona que este operador nos devuelve, cambiando los campo que nos interse cambiar, y reemplazar en la clase agenda al registro viejo. Si el operador [] devuelve al objeto por referencia, y la clase Persona tiene correctamente definida la asignación (puede implicar la sobrecarga del =), entonces reemplazar estos datos sería tan simple como asignar un elemento a un arreglo.

```
Agenda mi_agenda;
// ...cargar personas...
Persona una_persona = agenda[4]; // datos de la quinta persona
una_persona.ModifcarTelefono("342-57643424");
una_persona.ModifcarDireccion("25 de Mayo 3697");
agenda[4] = una_persona; // actualizar el quinto registro
```

• Eliminar una persona de la agenda: Para esto basta con un método que reciba qué registro eliminar. Cuando hay que hacer referencia a un registro conviene usar algo simple y rápido como un entero con el índice, antes que el nombre de la persona o el contenido de algún otro campo.

```
mi_agenda.EliminarPersona(7); // borrar el octavo registro
```

• Guardar los datos en un archivo y volverlos a cargar más tarde: Esto debería manejarlo la clase agenda. En este ejemplo vamos a utilizar archivos binarios, y vamos a hacer que la clase agenda siempre esté asociada a un archivo. Podríamos optar por trabajar siempre sobre el archivo (buscar, modificar, etc), pero sería más comodo tener la lista de datos en un contenedor STL (vector o lista) para poder aprovechar las rutinas de STL (busqueda, ordenamiento, manejo automático de memoria, etc.). Además este enfoque es más rápido, y si la base de datos no es demasiado grande no presenta inconvenientes. Entonces, podemos cargar los datos en el constructor, y guardarlos en el destructor, o en un método para tal fin (podríamos querer guardar antes de salir por las si se corta la luz, por ejemplo). Es buena costumbre además, devolver en los métodos propensos a fallar un código que indique si se guardó con éxito (ej: true o false).

```
Agenda mi_agenda("datos.dat"); // cargar desde un archivo
// ... agregar/borrar/modificar registros ...
if (mi_agenda.Guardar()) // actualizar el archivo
   cout<<"Archivo guardado."<<endl; // si guardó bien
else
   cout<<"Error al guardar datos."<<endl; // si no guardó
```

• Recuperar la lista completa: Para recuperar la lista de datos completa podemos utilizar el operador[], y sólo necesitaríamos añadir un método para saber cuantos registros contiene la instancia de Agenda. Si queremos ordenar esta lista, podemos sobrecargar el operador de comparación de la clase Persona y aprovechar los algoritmos de ordenamiento de STL. Una alternativa es proponer varias funciones de comparación con distintos criterios de ordenamiento y utilizar la sobrecarga de la función de ordenamiento que recibe un puntero a función con el criterio de comparación (esta función reemplaza al operador <). Optaremos por esta segunda opción, encapsulándola en un método que reciba una constante indicando el criterio y realice las llamadas adecuadas:

```
Agenda mi_agenda("datos.dat"); // cargar desde un archivo
agenda.Ordenar(ORDEN_NOMBRE);  // el criterio indica que función
                               // de comparación utiliza
```

```
for (int i=0;i<agenda.CantidadDatos();i++) // mostrar todos
   cout<<agenda[i].VerNombre()<<endl;   // los nombres
```

• Realizar una búsqueda: En un caso general una búsqueda puede arrojar más de un resultado. No es bueno en un método devolver todo un vector con los resultados (es caro computacionalmente o poco prolijo desde el punto de vista del manejo de memoria). De forma similar a las búsquedas en objetos string, vamos a proponer un método que reciba qué buscar y desde dónde buscar, y será responsabilidad del cliente invocarlo más de una vez si espera más de un resultado (si devolvíamos un vector igual necesitados un bucle para recorrerlo). Por comodidad conviene que el método devuelva la posición del elemento encontrado en lugar de devolver el elemento (es más rápido y flexible), o un valor especial si no lo encuentra (por ejemplo, -1).

```
cout<<"Los que contienen MA en el nombre son:"<<endl;
int res = agenda.BuscarNombre("MA",0); // buscar el primero
while (res!=NO_SE_ENCUENTRA) {
   cout<<agenda[res].VerNombre()<<endl; // mostrar
   res = agenda.BuscarNombre("MA",res+1); // buscar desde el
                                          // siguiente
}
```

De esta forma, hemos presentado lo necesario para realizar las interacciones básicas con las clases propuestas. Las interfases entonces serían:

```
// Clase que representa a una persona
class Persona {

   // datos de una persona
   string nombre;
   string apellido;
   string telefono;
   string direccion;
   string localidad;
   string email;
   int dia_nac, mes_nac, anio_nac;

public:

   // construir un obejto con los datos
   Persona(string a_nombre="", string a_apellido="",
           string a_telefono="", string a_direccion="",
           string a_localidad="", string a_email="",
           int a_dia=0, int a_mes=0, int a_anio=0);

   // verificar si los datos son correctos
   string ValidarDatos();

   // obtener los datos guardados
   string VerNombre();
   string VerApellido();
   string VerDireccion();
   string VerLocalidad();
   string VerTelefono();
   string VerEmail();
   int VerDiaNac();
   int VerMesNac();
   int VerAnioNac();

   // modificar los datos
   void ModificarNombre(string a_nombre);
   void ModificarApellido(string a_apellido);
   void ModificarDireccion(string a_direccion);
   void ModificarLocalidad(string a_localidad);
```

```
   void ModificarTelefono(string a_telefono);
   void ModificarEmail(string a_email);
   void ModificarFechaNac(int a_dia, int a_mes, int a_anio);

};


// Clase que administra la lista de personas
class Agenda {

   // nombre del archivo de donde se leen y donde se escriben los datos
   string nombre_archivo;

   // contenedor STL para los datos
   vector<Persona> arreglo;

public:

   // crea el objeto y carga los datos desde un archivo
   Agenda(string archivo);

   // guarda los datos en el archivo
   bool Guardar();

   // devuelve la cantidad de registros
   int CantidadDatos();

   // agrega un registro
   void AgregarPersona(const Persona &p);

   // devuelve un registro para ver o modificar
   Persona &operator[](int i);

   // quita una persona de la base de datos
   void EliminarPersona(int i);

   // ordena el vector
   void Ordenar(CriterioOrden criterio);

   // funciones para busquedas
   int BuscarNombre(string parte_nombre, int pos_desde);
   int BuscarApellido(string parte_apellido, int pos_desde);
   int BuscarDireccion(string parte_direccion, int pos_desde);
   int BuscarTelefono(string parte_telefono, int pos_desde);
   int BuscarEmail(string parte_correo, int pos_desde);
   int BuscarCiudad(string parte_ciudad, int pos_desde);
   int BuscarNacimiento(int dia, int mes, int anio, int pos_desde);

};
```

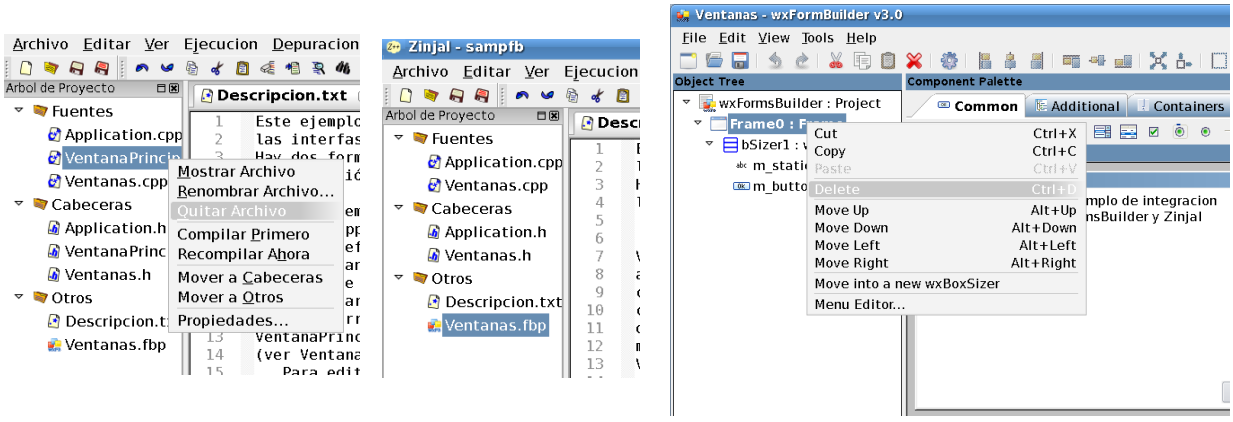
Las implementaciones completas se encuentran entre los archivos que acompañan al tutorial (ver Persona.cpp, Persona.h, Agenda.cpp y Agenda.h).

**Dibujando la interfaz de usuario: la magia de wxFormBuilder**

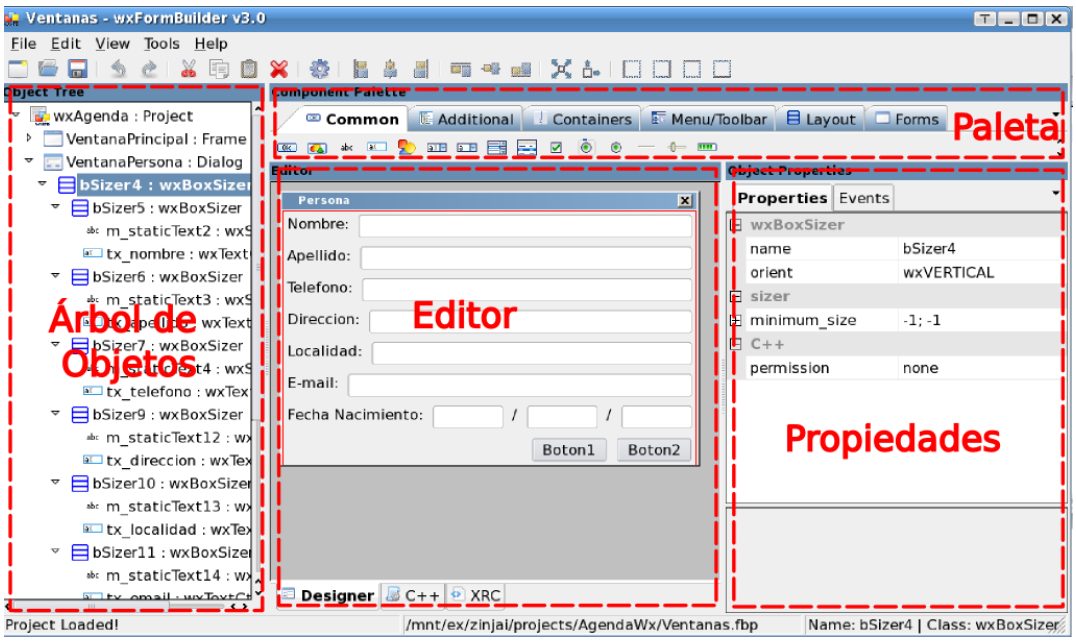
Utilizando la herramienta wxfb, vamos a poder dibujar las ventanas de forma visual casi sin requerir conocimientos de programación. Pero hay un detalle a tener en cuenta respecto a la política con se acomodan las cosas en una ventana. wxWidgets permite (al igual que muchas otras bibliotecas como GTK, QT, Swing, etc.) acomodar y dimensionar los componentes visuales automáticamente según una estructura jerárquica, a diferencia de otros diseños donde los controles están fijos en tamaño y posición. En una primera impresión, esta forma parece más dificil, pero el diseñador se acostumbra rápidamente, y se pueden mencionar varias ventajas. La principal radica en que no hay que preocuparse por cómo varían los tamaños de los elementos de una PC a otra, que pueden depender el tipo de letra, el ancho del borde, el sitema operativo, etc. ya que se calculan los tamaños automáticamente en base a las proporciones definidas y los mínimos necesarios.

La idea es entonces utilizar unos componentes llamados sizers. Un sizer ocupa toda el area donde se lo coloca, y la divide en celdas generalmente horizontales o verticales. En cada celda podemos colocar otro sizer o un componente visual (un botón por ejemplo). Cuando colocamos algo dentro de un sizer disponemos de banderas que indican si ese algo debe estirarse, centrarse, dejar un borde, etc.

Para comenzar debemos crear un proyecto en ZinjaI (Archivo->Nuevo Proyecto) utilizando la plantilla “wxFormBuilder Project”. Esto nos crea un proyecto con algunos archivos de ejemplo. Podemos probar ejecutarlo para verificar que la biblioteca está correctamente instalada. Luego, para borrar la clase heredada de ejemplo, seleccione los archivos (VentanaPrincipal.h y VentanaPrincipal.cpp) en el árbol de proyecto y presione la tecla Suprimir, o mediante un click con el botón derecho (en el cuadro de confirmación, puede seleccionar que desea borrarlos del disco también). Los archivos que sí interesan son Ventanas.h, Ventanas.cpp y Ventanas.fbp. Si hace doble click sobre este último y wxfb está correctamente instalado, se abrirá el diseñador mostrando la ventana de ejemplo. Seleccionela desede la barra de título (o en el árbol de objetos) y pulse Ctrl+D para eliminarla.

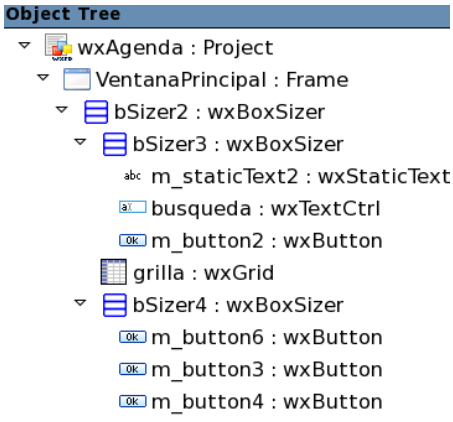



En la ventana de wxfb se distinguen las siguientes áreas:




- Árbol de objetos: muestra que componentes hay y cuál es la jerarquía.
- Paleta: contiene los elementos que podemos agregar. Cada pestaña es una categoría. Para agregar un elemento sólo hay que hacer un click. El elemento se agrega dentro o al lado de el componente que halla estado seleccionado en el árbol.
- Editor: permite visualizar el resultado y seleccionar componentes.
- Propiedades de Objeto: Permite definir propiedades, tanto visuales como de interés para el programador (por ejemplo, nombre del objeto/clase), y dispone además de una pestaña para definir los nombres de los métodos para los eventos que debe generar.

Por ejemplo, para lograr la ventana de la siguiente captura tenemos que seguir estos pasos:





1) Creamos una ventana: Seleccionar la pestaña “Forms” en la paleta y el primer boton (wxFrame). En el cuadro de propiedades cambiamos el tamaño (“width” y “height” dentro de “size”) de 500x300 a 700x500.

2) Dividimos la ventana verticalmente en tres: utilizando el primer botón (wxBoxSizer) de la pestaña “Layout” de la paleta (la cantidad no se ingresa, sino que es automática). Veremos un borde rojo en el interior de la ventana indicando el sizer seleccionado.

3) En el primer tercio volvemos a dividir en tres, pero horizontalmente (definiendo la propiedad “orient” igual a “wxHORIZONTAL”); es decir, creamos un nuevo wxBoxSizer teniendo seleccionado el anterior. Colocamos en él primero un rótulo (wxStaticText), luego un cuadro de texto

(wxTextCtrl), y luego un botón (wxButton), todos desde la pestaña “Common”. Para indicar que el cuadro de texto debe estirarse, lo seleccionamos y en el panel de propiedades definimos “Proportion” en “1”.

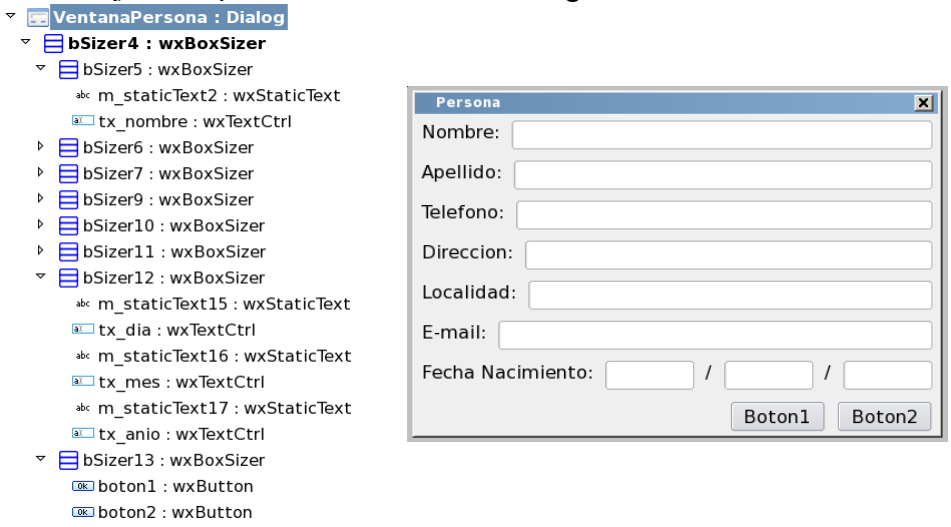
4) En el segundo lugar del sizer principal (hay que seleccionar el primer sizer en el árbol de objetos), colocamos una tabla (wxGrid, en la pestaña “Additional”). Para que la tabla ocupe todo el espacio libre, seteamos la propiedad “Proportion” del segundo sizer en “0” (el que contenia los controles para buscar). En las propiedades de la tabla, ponemos en 4 la cantidad de columnas (“cols”), en 0 la cantidad de filas (“rows”), desactivamos la posibilidad de editar los contenidos (destildar “editing”), ocultamos la columna de rótulos (“row_label_size” en “0”), y definimos los títulos de las demás columnas (“col_label_values”, haciendo click en los tres puntos suspensivos).

5) Agregamos la barra de botones inferior. Para ello colocamos otro sizer horizontal que depende del primero, y tres botones dentro de el mismo. En las propiedades del sizer, definimos “Proportion” en “0”, desactivamos “EXPAND” y activamos “wxALIGN_RIGHT” dentro de “flag”; para que el contenido se ubique a la derecha en lugar de estirarse. Para los rótulos de los botones, modificamos la propiedad “label”.

Finalmente, definimos los nombres de los controles que nos interesa referenciar desde nuestro programa (la ventana como “VentanaPrincipal” (nombre de clase), la grilla como “grilla” (nombre de objeto) y el cuadro de texto como “busqueda”) utilizando la propiedad “name”.

Si necesitara mover un componente, debe hacerlo arrastrándolo desde el arbol de objetos. Para borrarlo, también puede seleccionarlo allí y hacer click con el botón derecho del mouse. Finalmente, el árbol debe quedar como en la imágen anterior.

Se deja como ejercicio para el lector construir la segunda ventana:




Esta ventana no será un instancia de la clase wxFrame, sino de la clase wxDialog (el tercer botón de la pestaña “Forms” en la paleta). Éste es un tipo de ventana que se usa para cuadros de diálogo emergente que dependen de una ventana principal (entre otras ventajas, permiten mostrarse de forma modal, más adelante se discutirá qué significa). Observamos que al crearla no se dibuja más que la barra de títulos. Esto es porque el tamaño del diálogo se ajusta al contenido, pero inicialmete está vacio. Esta ventana se va a utilizar para mostrar los datos completos de una persona, agregar un persona nueva, o modificar una existente. De la misma clase se pueden heredar las tres ventanas de igual apariencia pero comportamiento diferente. Como ayuda, se aclara conviene utilizar un wxBoxSizer vertical, y luego dentro de este varios sizers horizontales (uso por linea de elementos). Si bien hay sizers más avanzados, al principio es más facil lograr el efecto deseado combinando objetos wxBoxSizer verticales y horizontales alternadamente. Finalmente, no olvide colocarle el nombre

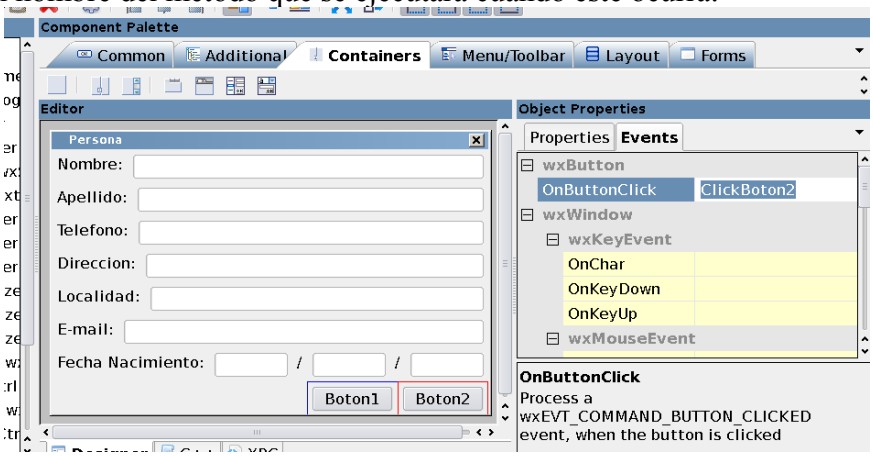
adecuado a los componentes para luego referenciarlos desde el código de los eventos (los cuadros de texto como “tx_<atributo>” y los botones como “boton1” y “boton2”).

**Integrando los dos mundos: Cómo conectar las clases con los eventos**

A esta altura tenemos diseñadas las dos ventanas y las dos clases (Persona y Agenda) con sus funciones y estructuras auxiliares. Para unir las dos partes debemos:

1) Definir que eventos interesan
2) Generar el código de las ventanas desde wxfb
3) Generar las clases heredadas de las ventanas en ZinjaI
4) Codificar los eventos usando las clases ya desarrolladas
5) Compilar, ejecutar, depurar...

Para definir los eventos de interés, debemos ir al panel de propiedades y seleccionar la pestaña "Events". En ella aparecerán los posibles eventos del objeto seleccionado. El valor que le asignemos a cada evento es el nombre del método que se ejecutará cuando éste ocurra.



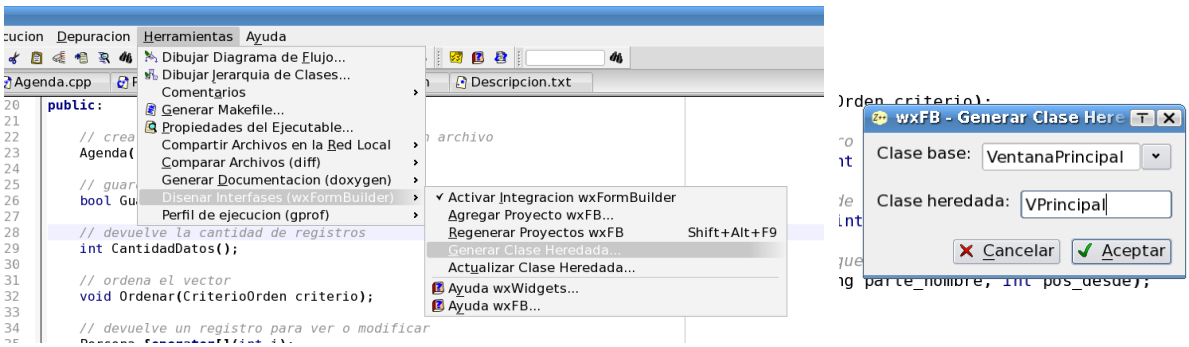

En la ventana principal los eventos que interesan son:

- Click en algún botón: Al seleccionar un botón el primer evento de la lista es "OnButtonClick", allí ingresamos:
- Para el botón Buscar: "ClickBuscar"
- Para el botón Agregar: "ClickAgregar"
- Para el botón Editar: "ClickEditar"
- Para el botón Eliminar: "ClickEliminar"
- Click en el título de alguna columna de la tabla: Vamos a permitir reordenar la lista por diferentes columna mediante esta acción. El evento es "OnGridLabelRightClick" y al método lo vamos a llamar "ClickGrilla"

En la ventana de la Persona, los eventos que interesan son los clicks en los dos botones; llamaremos a los métodos "ClickBotón1" y "ClickBotón2".

Una vez definidos los eventos, guardamos el archivo (menú File->Save) y generamos el código fuente (menú File->Generate Code, o el botón con el engranaje de la barra de herramientas). Si cerramos wxfb y volvemos a ZinjaI, podemos ver cómo los archivos Ventanas.h y Ventanas.cpp ahora tienen el código que genera la interfaz que diseñamos. Cabe aclarar que si modifican algo en wxfb y guardan los cambios, pero no regeneran el código, ZinjaI lo hará automáticamente antes de compilar el proyecto.

Ahora que disponemos de el código base de las ventanas tenemos que generar las clases heredadas. ZinjaI puede realizar este proceso rápidamente. Para ello hay que elegir la opción "Generar Clase Hereda" del submenú "Diseñar Interfases" del menú "Herramientas".




Con este mecanismo debemos generar tres clases:

- VPrincipal, que hereda de VentanaPrincipal
- VAgregar, que hereda de VentanaPersona
- VEditar, que hereda de VentanaPersona

Una vez generadas las clases, debemos completar el proceso de inicialización. Cuando la aplicación se carga debe leer la base de datos (crear una instancia de la clase Agenda) y debe crear la ventana principal (una instancia de VPrincipal). Suele haber cierta confusión cuando se necesita que una única instancia de una clase sea visible desde toda la aplicación. En nuestro caso debe haber una única instancia de la clase Agenda para que todos los datos se manejen sobre el mismo vector. Una forma de conseguir esto es declarar una variable global de tipo Agenda (en realidad, conviene usar un puntero, para poder invocar al constructor cuando sea oportuno). El problema es que si el puntero se declara en un archivo .cpp, entonces los demás no pueden verlo, pero si se declara en un archivo .h, entonces cuando se incluya varias veces desde distintos .cpp tendremos varias veces la misma variable y el programa no enlazará (no puede haber dos variables globales con el mismo nombre). La solución es utilizar la palabra clave "extern": cuando colocamos en el código "`Agenda *mi_agenda;`" estamos declarando y definiendo la variable mi_agenda (puntero a una instancia del objeto Agenda); es decir, estamos avisando que existe y reservando memoria para la misma. Cuando anteponemos extern, sólo estamos avisando que existe, pero no reservando memoria. Entonces puede aparecer muchas veces la línea "`extern Agenda *mi_agenda;`", pero debe apercer una y sólo una vez la línea "`Agenda *mi_agenda;`" en un .cpp. Así se reserva memoria una sóla vez, pero se puede utilizar desde muchos lugares. El paso lógico entonces es declarar la variable con la palabra "extern" en el archivo "Agenda.h", y sin la palabra "extern" en el archivo "Agenda.cpp".

Volvientdo a la inicialización de la aplicación, ésta se lleva a cabo en el método OnInit de la clase wxApplication (el proyecto de ZinjaI ya cuenta con una herencia de esta clase llamada mxApplication). Allí debemos inicializar la instancia de Agenda y crear la primer ventana. Para lo primero hacemos simplemente "`mi_agenda = new Agenda("datos.dat");`" (notar que al poner el nombre del archivo sin su ruta/directorio, éste se busca en la carpeta del proyecto o del ejecutable); mientras que para crear la instancia de la ventana hacemos "`new VPrincipal(NULL);`" (no hace falta asignarla a una variable, pero debe ser creada con new para que no se pierda al terminar el método OnInit). Cada ventana recibe en su constructor al menos un parámetro, que indica cual es la ventana que la precede (en este caso NULL por ser la primera). Al salir de este método (OnInit) se le pasa el control automáticamente al bucle de eventos de wxWidgets. Hay que destacar que en una aplicación wxWidgets el programador no define una función main, ya que esta está contenida en la biblioteca y se encarga de crear la instancia de wxApplication y llamar al método OnInit.

A partir de aquí, sólo resta programar los eventos. Notar que para la carga de una ventana no hay un evento en particular porque se puede utilizar su constructor. Éste es el caso de la ventana principal, que debería transferir los datos del objeto mi_agenda a la grilla. Entonces, el constructor de Vprincipal.cpp sería:

```
VPrincipal::VPrincipal(wxWindow *parent) : VentanaPrincipal(parent) {
   int c_pers = mi_agenda->CantidadDatos(); // cantidad de personas
```

```
   grilla->AppendRows(c_pers); // agregar tantas filas como registros
   for (int i=0;i<c_pers;i++) CargarFila(i);// cargar todos los datos
   grilla->SetSelectionMode(wxGrid::wxGridSelectRows);
   Show(); // mostrar la ventana
}

void VPrincipal::CargarFila(int i) {
   Persona &p=(*mi_agenda)[i];
   grilla->SetCellValue(i,0,(p.VerApellido()+", "
                            +p.VerNombre()).c_str());
   grilla->SetCellValue(i,1,p.VerDireccion().c_str());
   grilla->SetCellValue(i,2,p.VerTelefono().c_str());
   grilla->SetCellValue(i,3,p.VerEmail().c_str());
}
```

Con el métodos AppendRows de grilla midificamos la cantidad de renglones para que coincida con la cantidad de registros. Luego, con SetCellValue cargamos los datos en cada celda de la tabla. El tipo de dato que se le debe entregar a la celda es un objeto wxString (cadena propia de wxWidget), por esto convertimos las cadenas tipos string a arreglos de caracteres con el método c_str; entonces, ya que hay un constructor de wxString que recibe un arreglo de caracteres estilo c. Finalmente, el método SetSelectionMode permite configurar una propiedad que no aparece en la paleta de wxfb, que fuerza al control a seleccionar toda una fila y no una celda individual al hacer click (para conocer todas los métodos y propiedades de un objeto consulte la documentación de la clase en la referencia oficial de wxWidgets; en ZinjaI, menu Herrmientas->Diseñar Interfases->Ayuda wxWidgets).

Para poder agregar registros debemos definir el comportamiento rotulado “Agregar”. Este botón debería crear una instancia de la clase VAgregar, esperar a que el usuario complete los datos, y luego actualizar la lista. Hay dos formas de hacerlo. La actualización de la grilla la puede hacer el evento del botón agregar si espera a que se cierre la ventana de VAgregar, o la puede hacer la ventana de VAgregar en su evento de cierre o de click en el botón guardar. Para evitar tener que pasarle a la segunda ventana punteros a objetos de la primera, se utiliza el primer enfoque.

Cuando se muestra una ventana, hay dos formas de hacerlo; modal o no modal. Si se muestra como modal, la primer ventana no hace nada (no responde a eventos ni avanza en la ejecución del evento que llamó a la seguna) hasta que la segunda no se cierre; si no es modal, ambas ventanas continúan independientemente. El caso que queremos es el primero. Además, cuando se cierra la ventana modal, puede devolver un valor entero en la llamada a ShowModal para indicar qué sucedió, por lo que el código para el evento ClickAgregar será simplemente:

```
void VPrincipal::ClickAgregar( wxCommandEvent& event ) {
   VAgregar *nueva_ventana = new VAgregar(this); // crear la ventana
   if (nueva_ventana->ShowModal()==1) { // mostrar y esperar
      grilla->AppendRows(1); // agregar el lugar en la grilla
      CargarFila(mi_agenda->CantidadDatos()-1); // copiar en la grilla
   }
}
```

y dejamos la tarea de actualizar la base de datos (objeto mi_agenda) al botón aceptar de la ventana VAgregar. Éste código, considerando lo que desarrollamos en la clase persona para la validación, puede ser:

```
void VAgregar::ClickBoton2( wxCommandEvent& event ) {
   long dia,mes,anio; // convertir cadenas a numeros
   tx_dia->GetValue().ToLong(&dia);
   tx_mes->GetValue().ToLong(&mes);
   tx_anio->GetValue().ToLong(&anio);
   Persona p( // crear la instancia de persona
      tx_nombre->GetValue().c_str(),
      tx_apellido->GetValue().c_str(),
      tx_telefono->GetValue().c_str(),
```

```
            tx_direccion->GetValue().c_str(),
            tx_localidad->GetValue().c_str(),
            tx_email->GetValue().c_str(),
            dia,mes,anio);
         string errores = p.ValidarDatos();
         if (errores.size()) // ver si no pasa la validacion
            wxMessageBox(errores); // mostrar errores
         else {
            mi_agenda->AgregarPersona(p);
            mi_agenda->Guardar(); // actualizar el archivo
            EndModal(1); // cerrar indicando que se agrego
         }
      }
```

Notar que podemos utilizar la función wxMessageBox en cualquier momento para mostrar un mensaje al usuario o hacer una pregunta de tipo si-o-no/aceptar-cancelar.

Suponiendo que el Boton 2 sirve para agregar, mientras que el 1 para cancelar la operación. Esto se puede definir en el constructor:

```
VAgregar::VAgregar(wxWindow *parent) : VentanaPersona(parent) {
   SetTitle("Agregar Persona"); // titulo de la ventana
   boton1->SetLabel("Cancelar"); // rotulo boton 1
   boton2->SetLabel("Guardar"); // rotulo boton 2
}
```

Como se vió antes, la función EndModal sirve para indicar qué valor debe devolver la llamada a ShowModal. En este ejemplo, queremos que cancelar devuelva 0 indicando que no se agregó ningún registro. El código sería:

```
void VAgregar::ClickBoton1( wxCommandEvent& event ) {
   EndModal(0);
}
```

Notar que nuevamente, para convertir entre wxString (lo que devuelve GetValue) a string debemos usar como intermediarias a las cadenas tipo c (arreglos de caracteres).

Se deja como ejercicio para el lector el desarrollo de la clase VEditar y su llamada desde el método ClickEditar de VPrincipal. Esta clase debe cargar los datos de una persona en los cuadros de texto, por lo que conviene modificar el prototipo del constructor para recibir el índice del registro de esa persona, y en el mismo copiar los valores del objeto persona a los cuadros de texto. Para saber cual es el índice de registro, se puede utilizar el índice de la fila seleccionada. Un problema que va a surgir al cargar los datos en el formulario, es cómo convertir enteros a wxString. Por suerte wxString tiene numerosas sobrecargas del operador <<, por lo que se puede concatenar otros tipos de datos como si fuera un flujo c++ (`Ejemplo: wxString("El numero es: ")<<num;`).

Vamos a mostrar ahora una forma de realizar la búsqueda con el cuadro de texto de la ventana principal. Esta búsqueda se realiza cuando se hace click en el botón buscar. El mecanismo es el siguiente: el usuario ingresa parte del nombre o del apellido de la persona que busca y al hacer click en el botón "Buscar" se selecciona la próxima ocurrencia, buscando desde la selección anterior. Así, haciendo click varias veces en "Buscar", puede ir recorriendo todos los resultados. El método podría codificarse como sigue:

```
void VPrincipal::ClickBuscar( wxCommandEvent& event ) {
   int fila_actual = grilla->GetGridCursorRow();
   int res = mi_agenda->BuscarApellidoYNombre(
                              busqueda->GetValue().c_str(),fila_actual);
   if (res==NO_SE_ENCUENTRA)
      wxMessageBox("No se encontraron mas coincidencias");
   else {
      grilla->SetGridCursor(res,0); // seleccionar celda
```

```
      grilla->SelectRow(res); // marcar toda la fila
      grilla->MakeCellVisible(res,0); // asegurarse de que se ve
   }
}
```

Otra implementación interesante es la de el ordenamiento de la tabla, ya que para saber cómo ordenar hay que averiguar en qué columna se hizo click, utilizando el objeto que recibe el método del evento (wxEvent o derivado):

```
void VPrincipal::ClickGrilla( wxGridEvent& event ) {
   int columna=event.GetCol(), c_pers=mi_agenda->CantidadDatos();
   switch(columna) { // ordenar en mi_agenda
      case 0: mi_agenda->Ordenar(ORDEN_APELLIDO_Y_NOMBRE); break;
      case 1: mi_agenda->Ordenar(ORDEN_DIRECCION); break;
      case 2: mi_agenda->Ordenar(ORDEN_TELEFONO); break;
      case 3: mi_agenda->Ordenar(ORDEN_EMAIL); break;
   }
   for (int i=0;i<c_pers;i++) CargarFila(i); // actualizar vista
}
```

Finalmente, lo que resta por implementar es la eliminación de un registro. Para ello no se necesita ninguna ventana adicional, pero se puede usar la función wxMessageBox con parámetros adicionales para pedir una confirmación:

```
void VPrincipal::ClickEliminar( wxCommandEvent& event ) {
   int fila_actual = grilla->GetGridCursorRow();
   int res = wxMessageBox("Eliminar el registro?",
                          grilla->GetCellValue(fila_actual,0),wxYES_NO);
   if (res==wxYES) {
      grilla->DeleteRows(fila_actual,1);
      mi_agenda->EliminarPersona(fila_actual);
      mi_agenda->Guardar();
   }
}
```

En este punto la aplicación debería estar funcionando. Si observamos finalmente los métodos de las clases visuales podemos comprobar que ninguno requiere demasiado procesamiento, ya que son simples clientes de las clases Agenda y Persona, y han podido ser implementados con relativa facilidad.

**Más Información: Ayuda!!!**

Para más información se pueden consultar varias fuentes. Como recomendación se debe mencionar la documentación oficial de la biblioteca (para referencia, muy completa, clase por clase), los ejemplos (para esto hay que bajar la biblioteca desde su sitio oficial y buscar en el directorio samples) y las búsquedas en Google (wxWidgets se utiliza mucho en el mundo del software libre, por lo que se pueden encontrar miles de foros al respecto).