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

Dr. Pablo Novara
Ultima revisión: 27/11/2017

Tutorial: Desarrollo de un proyecto con wxWidgets

En este documento se cubren los aspectos esenciales del desarrollo de un proyecto ejemplo 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 interfaces gráficas. Contiene clases para construir ventanas, paneles, botones, imágenes, cuadros de texto, listas desplegables, cuadros de selección doementos básicos que suelen incluir las bibliotecas de este tipo y/o los entornos de desarrollo visuales. Como ventaja frente a otras alternativas, es libre y gratuita, orientada a objetos y multiplataforma[1].

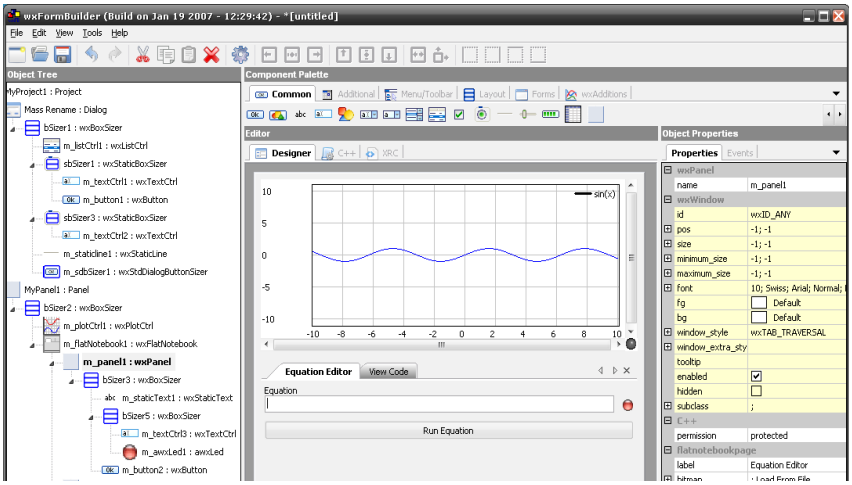
wxWidgets es una biblioteca, no un IDE, ni una herramienta de diseño visual, ni nada más que mucho código. Utilizarla, por lo tanto, requiere escribir más código: un programa cliente para las clases/funciones de la biblioteca. Por ejemplo, para crear una aplicación y tener una ventana, hay que codificar una clase que herede de la clase `wxFrame`. La clase `wxFrame` es la clase que representa una ventana "normal" vacía, y tiene implementado todo lo relacionado a la visualización de la ventana y la detección de sus **eventos**. El programador debe implementar en una clase propia, que herede de `wxFrame`, un constructor que coloque dentro de la ventana los **componentes** que necesite. Usualmente la biblioteca ofrecerá una clase para representar cada tipo de componente, y el programador los agregará a la ventana por composición. Luego deberá programar en métodos los comportamientos de la ventana para los distintos eventos e indicar qué evento se asocia con qué método.

Un **evento** es un suceso interesante, usualmente una acción del usuario, al que la aplicación puede o debe responder de alguna manera. Ejemplos de eventos son: presionar una tecla, hacer click sobre un botón, maximizar la ventana, etc.

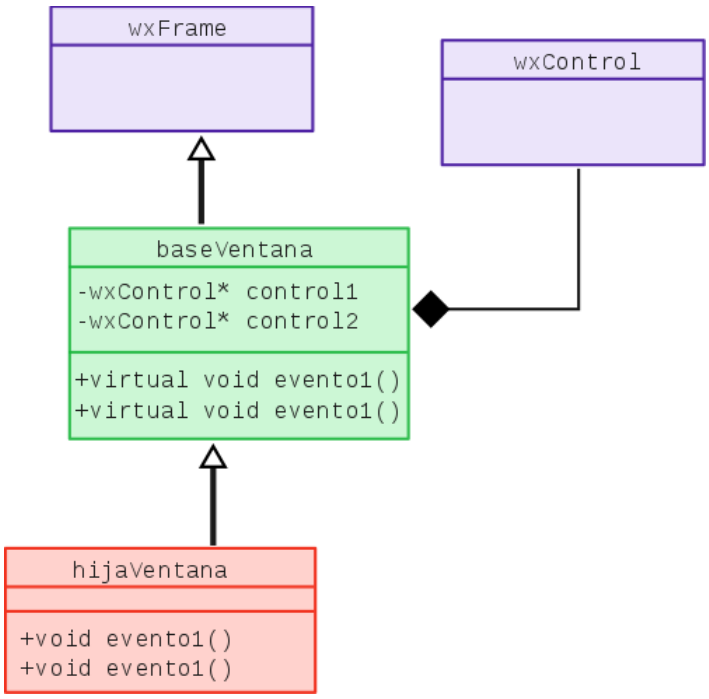
Se denomina **componente, control o widget** a un elemento visual dentro de la ventana. Ejemplos de componentes son: un botón, un cuadro de texto editable, una lista desplegables, una barra de desplazamiento, una etiqueta de texto fijo, etc.

Muchas tareas básicas como éstas resultan repetitivas, tediosas y no son fáciles de realizar sólo pensando en el código (sin ayudas visuales). Para solucionar este problema, existen herramientas externas a la biblioteca que permiten "dibujar" las ventanas sin tener que introducir ninguna línea de código, y que luego generan de forma automática los fuentes de las clases que se necesitan para obtener por resultado lo que se "dibujó". Denominaremos a dicho tipo de herramientas "diseñadores", ya que permiten "diseñar" el aspecto de las ventanas. En este tutorial se va a utilizar una de ellas: wxFormBuilder (wxfb de ahora en más).

1 Como valor agregado, wxWidgets también ofrece clases para el manejo de procesos, comunicaciones, hilos, archivos, cadenas, funciones de internacionalización, etc. que van más allá de lo meramente visual y que no serán abordadas en este ejemplo.



wxfb va a generar dos archivos fuentes (un .cpp y un .h) con el código de las ventanas que se hayan "dibujado" como clases (una clase por cada ventana). El programador debe generar a partir estas y por herencia sus propias clases en otros archivos. En estas clases derivadas sólo será necesario implementar los métodos asociados a los eventos que le interesen, ya que los constructores que combinan los componentes y relacionan métodos y eventos ya habrán sido generados por el wxfb. Nunca se debe trabajar directamente sobre las clases generadas por wxfb porque ante cualquier cambio en el diseño de las ventanas se volverán a regenerar estos archivos y se perderán la modificaciones realizadas a los mismos. Además es buena práctica mantener las partes "manuales" y "automáticas" separadas. Por estos motivos, es que los métodos de las clases que genera wxfb serán virtuales.



Estructura de clases habitual: En azul: clases provistas por la biblioteca wxWidgets. En verde: clase generada por el diseñador wxfb. En rojo: la única que debe efectivamente implementar el programador, en ella se implementan los métodos que reaccionan a los eventos.

Finalmente, cabe mencionar que la implementación de los métodos que definen cómo la aplicación reacciona frente a los eventos de interés consistirá mayormente en llamadas a las clases

que modelan el problema. En este ejemplo, una clase `Agenda` ya tendrá métodos para agregar contactos, buscarlos, editarlos, validar sus datos, ordenarlos, cargarlos desde un archivo y luego guardar los cambios, etc. Todas estas operaciones son independientes de la interfaz. Por ejemplo, cómo se almacenan los datos de una persona en un archivo binario no depende de la procedencia de dichos datos (es decir, de si se cargan por consola o mediante una ventana). En consecuencia, no debería haber código relacionada a la lectura/escritura de archivos directamente dentro de un método asociado a un evento. Esto estará implementado en la clase `Agenda`, y el método del evento se limitará a invocar a los métodos de dicha clase. Por ello, será importante diseñar y verificar correctamente el conjunto de clases que representan el problema antes de pasar a la interfaz visual.

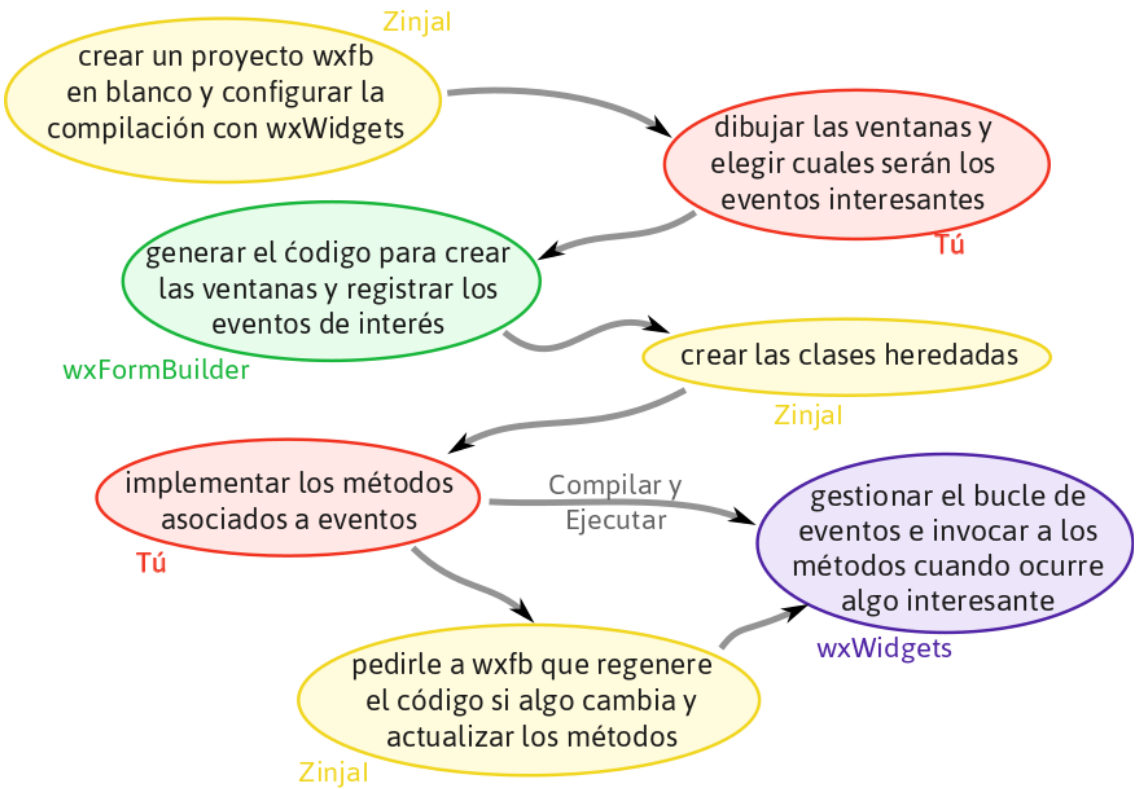
En resumen, en este tutorial se va a utilizar el IDE ZinjaI y el diseñador wxFormBuilder para construir una aplicación que utilice la biblioteca wxWidgets, de forma fácil y rápida, evitando las tareas más rutinarias, pero sin perder de vista qué cosas se están haciendo automáticamente para comprender mejor el desarrollo y funcionamiento de la aplicación. En lineas generales, los pasos del proceso son los siguientes:

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 y elegir cuales eventos interesan (wxfb generará automáticamente el código fuente para esos diseños).

3) Crear por herencia nuevas clases a partir de las generadas por wxfb para sobrescribir los métodos virtuales asociados a eventos, invocando a las clases y funciones del paso 1.

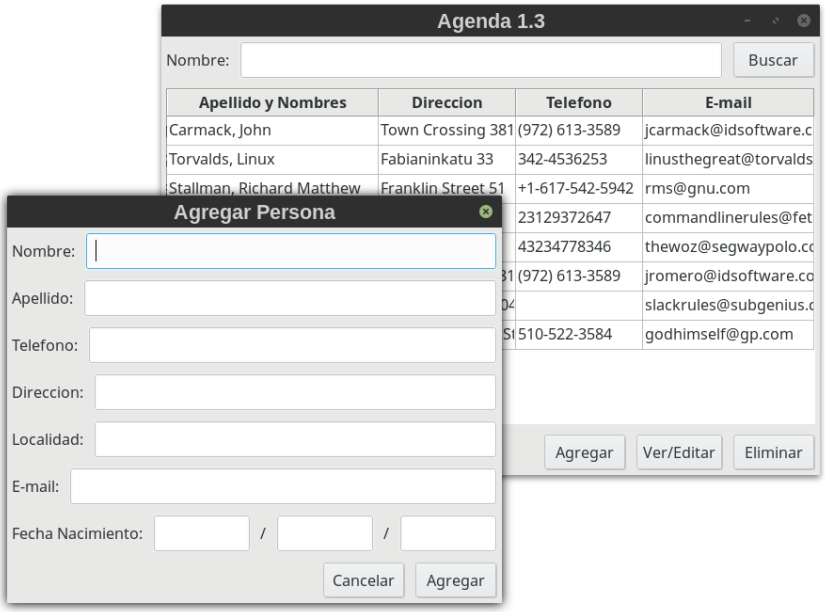
4) Compilar, ejecutar, depurar…



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. Se va a explicar primero la construcción de un conjunto de clases completas y consistentes para representar el problema, independientes de la interfaz, para luego integrarlas 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 (`wxAgenda.exe` en Windows, `wxAgenda.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.

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 de interés. Es decir, la biblioteca estará esperando alguna acción de el usuario. Si el usuario hace click en un botón, por ejemplo, la biblioteca invocará al 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, cuándo y cómo ocurren.

Se conoce como **“inversión del control”** o **“principio de Holliwood”** al hecho de que el código cliente no defina el flujo de control (el orden en que se ejecutan las acciones), sino que solo defina la respuesta a ciertos eventos. De esta forma, el orden en que se generen los eventos es lo que efectivamente determinará el orden en que se ejecuten las posibles acciones.

Instalando las herramientas: Manos a la obra

Como toda biblioteca no estándar, 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, existe un complemento para ZinjaI que incluye una plantilla de proyecto con todas estas configuraciones ya especificadas para utilizar wxWidgets ya sea en Windows, en GNU/Linux, o Mac OS. Además, ZinjaI gestiona de forma especial los archivos y las clases relacionadas a un proyecto de wxfb, automatizando algunas tareas repetitivas propias de la integración del diseñador visual en un proyecto C++.

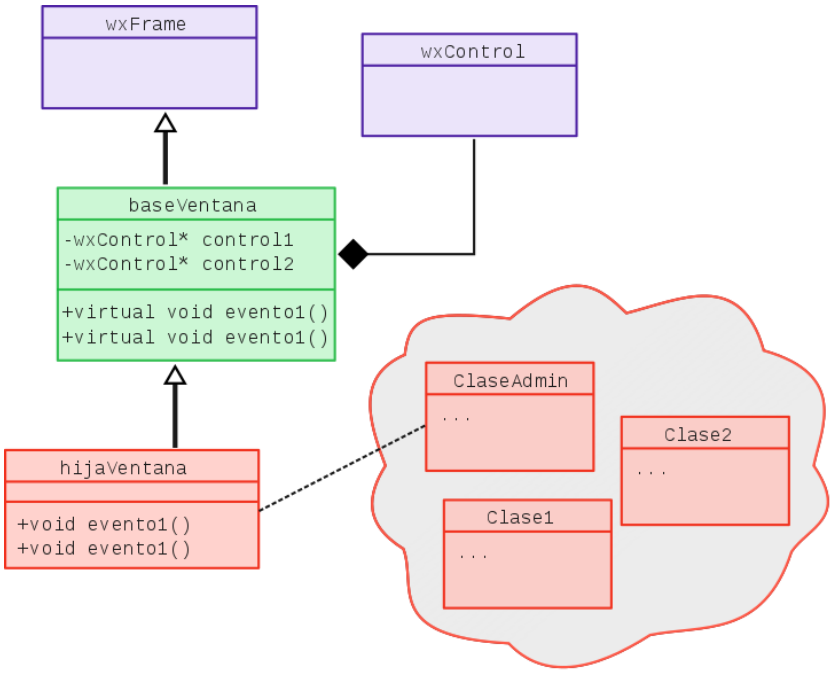
Se recomienda descargar ZinjaI y el complemento para wxWidgets desde su sitio oficial (http://zinjai.sourceforge.net). Una vez instalado/descomprimido ZinjaI, para instalar el complemento (archivo .zcp) debe buscar la opción “Instalar Complemento...” del menú “Herramientas”.

En GNU/Linux, el complemento no incluye realmente la biblioteca (sólo incluye la configuración para ZinjaI). Debe utilizar el gestor de paquetes de su distribución (aptitude, synaptic, yum, yast, etc) para instalar la biblioteca wxWidgets. Asegúrese de buscar e instalar el paquete wxWidgets que termine -dev o -devel.

Por último, en cualquier caso (ya sea GNU/Linux o Ms. Windows), el diseñador wxFormBuilder debe ser instalado por separado. En GNU/Linux, usualmente encontrará el diseñador disponible en el gestor de paquetes de la distribución. Para sistemas Windows puede descargar el instalador desde la siguiente dirección: https://github.com/wxFormBuilder/wxFormBuilder/releases.

Desarrollo del código base: Cómo no pensar en la interfaz

Como se mencionó anteriormente, 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 utilizarse desde un cliente de consola, desde un programa gráfico, o en un servidor web por ejemplo. 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.



Otra ventaja radica en la detección y corrección de errores. Es común hacer pequeños programas cliente de consola para probar cada una de las funcionalidades que se van implementando en estas clases, de modo que al comenzar a implementar la interfaz tengamos la seguridad de que las clases bases funcionan correctamente. Así, si aparece bug en la segunda etapa, sabremos que hay que buscar su raíz en el código de la interfaz. Si se desarrollan ambas cosas en paralelo, al encontrar un error (por ejemplo, un listado que no muestra los datos que esperamos que muestre), no sabremos si es un problemas de las clases bases (por ejemplo que los datos no se guardan o leen bien en sus archivo), o de la interfaz (que no se inicializa correctamente la tabla por ejemplo).

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/contactos. 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 también la escritura y lectura desde un archivo.

Para definir concretamente la interfaz que estas clases exponen a las rutinas cliente, podemos pensar en las operaciones más comunes que quisiéramos 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 m_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()) {
   cout << ”Los datos no son correctos:” << endl;
   cout << errores << endl;
} else
   m_agenda.Agregar(nueva_persona);
```

- 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 previos de la 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 interese 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 m_agenda;
// ...cargar personas...
Persona una_persona = agenda[4]; // obtener 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 (que no implique volver a buscar) como un entero con el índice, antes que el nombre de la persona o el contenido de algún otro campo.

```
m_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 cómodo tener la lista de datos en un contenedor STL (`std::vector` o `std::lista`) para poder aprovechar las rutinas de STL (búsqueda, 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 m_agenda("datos.dat"); // cargar desde un archivo
// ... agregar/borrar/modificar registros ...
if (m_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, para facilitar su utilización:

```
Agenda m_agenda("datos.dat"); // cargar desde un archivo
agenda.Ordenar(ORDEN_NOMBRE);  // el criterio indica qué función
                               // de comparación utilizar
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 o en contenedores STL, 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 (notar que 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 otra vez desde el
                                          siguiente
}
```

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

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

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

public:

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

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

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

   // modificar los datos
   void ModificarNombre(std::string a_nombre);
   void ModificarApellido(std::string a_apellido);
   void ModificarDireccion(std::string a_direccion);
   void ModificarLocalidad(std::string a_localidad);
   void ModificarTelefono(std::string a_telefono);
   void ModificarEmail(std::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
   std::string nombre_archivo;

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

public:

   // crea el objeto y carga los datos desde un archivo
```

Notará que en las declaraciones se utiliza “std::string” y “std::vector” en lugar de simplemente “string” y “vector”. Esto se debe a que no se recomienda colocar directivas “using namespace ...” en los archivos cabecera (.h).

```
        Agenda(std::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(std::string parte_nombre, int pos_desde);
        int BuscarApellido(std::string parte_apellido, int pos_desde);
        int BuscarDireccion(std::string parte_direccion, int pos_desde);
        int BuscarTelefono(std::string parte_telefono, int pos_desde);
        int BuscarEmail(std::string parte_correo, int pos_desde);
        int BuscarCiudad(std::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`).

Hay un último detalle que es importante destacar, ya que es una fuente frecuente de errores y problemas. Ya definimos que para cada persona se guardarán los atributos nombre, apellido, teléfono, dirección, localidad, email y fecha de nacimiento; siendo la mayoría de los campos cadenas de texto. Aquí se plantea una disyuntiva importante: si queremos poder manipular la cadena con facilidad (inicializar, copiar, comparar, concatenar, etc) es conveniente representarlas mediante objetos `std::string`, pero estos objetos **no pueden** guardarse directamente en archivos binarios; si queremos guardar/leer directamente cadenas en/desde archivos binarios deberíamos representarlas como c-strings de longitud fija (`char[N]`). La solución que aquí se propone consiste en utilizar `std::strings` en la clase `Persona`, y un struct auxiliar como intermediario para guardar/leer que contenga cadenas de caracteres al estilo c. Podemos colocar estas responsabilidades en la clase persona agregando en la misma métodos para que cada instancia escriba/lea sus datos mediante el struct auxiliar aplicando las conversiones pertinentes.

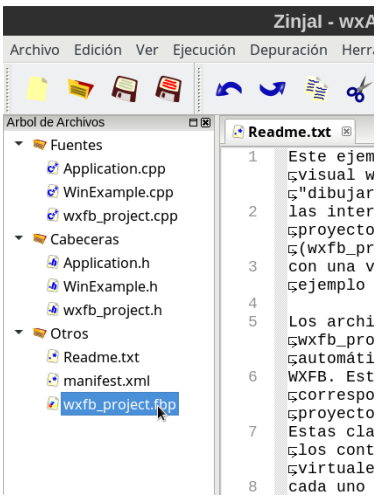
Recuerde que **si una clase o struct contiene punteros**, **no debe escribirse/leerse en/desde archivos en modo binario**, ya que se estaría escribiendo/leyendo solo el valor del puntero (una dirección de memoria) pero no los datos a los que efectivamente apunta.

Dibujando la interfaz de usuario: la magia de wxFormBuilder

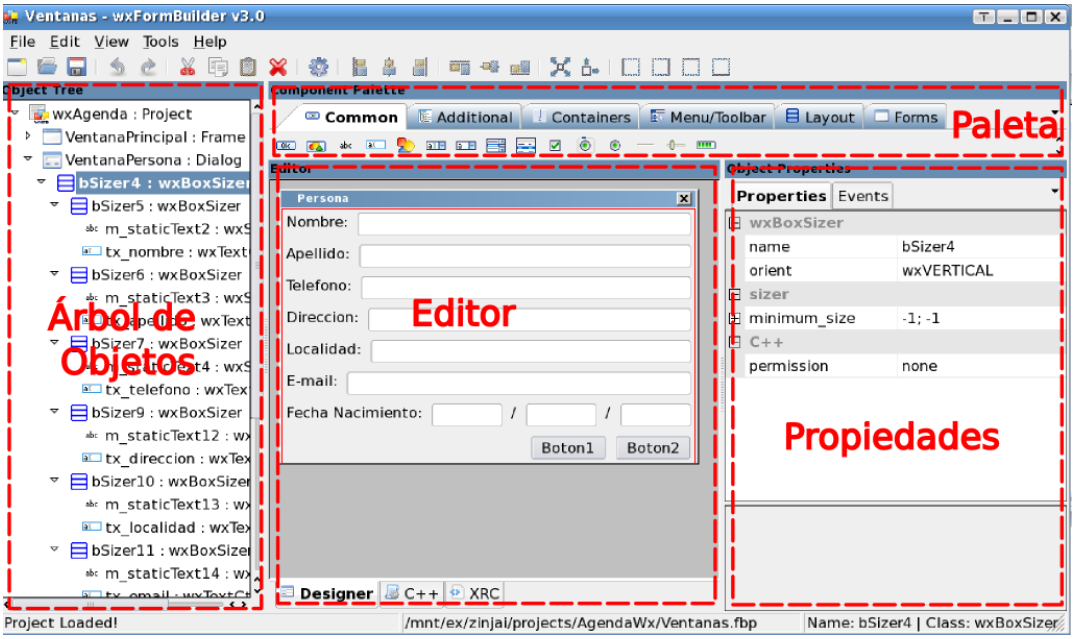
Utilizando la herramienta wxfb, vamos a poder dibujar las ventanas de forma visual casi sin requerir conocimientos de programación. Pero hay un detalle a tener en cuenta respecto a la política con se acomodan las cosas en una ventana. wxWidgets permite (al igual que muchas otras bibliotecas como GTK, QT, Swing, etc.) acomodar y dimensionar los componentes visuales automáticamente según una estructura jerárquica y algunas indicaciones (como por ejemplo, cómo alinear o cuándo estirar si sobra espacio), a diferencia de otros diseños donde los controles están fijos en tamaño y posición. En una primera impresión, esta forma parece más difícil, 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 sistema operativo, la resolución de la pantalla, el tema de escritorio, 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 área donde se lo coloca (por ejemplo, toda la ventana), y la divide en celdas generalmente horizontales o verticales. En cada celda podemos anidar 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. De esta forma, el sizer se encarga de calcular y ajustar adecuadamente los tamaños y posiciones de los demás controles o sizers que contiene, respetando las indicaciones proporcionadas mediante dichas banderas.

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 si la biblioteca está correctamente instalada. Los archivos que interesan son `wxfb_project.h`, `wxfb_project.cpp` y `wxfb_project.fbp`.



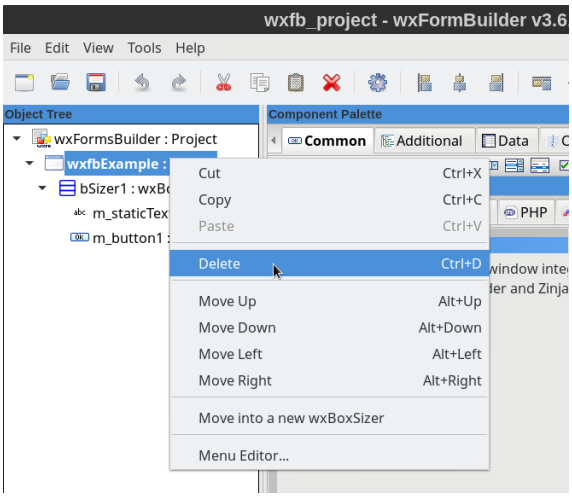
Si hace doble click sobre este último y wxfb está correctamente instalado, se abrirá el diseñador y podrá ver la estructura de la ventana del ejemplo.



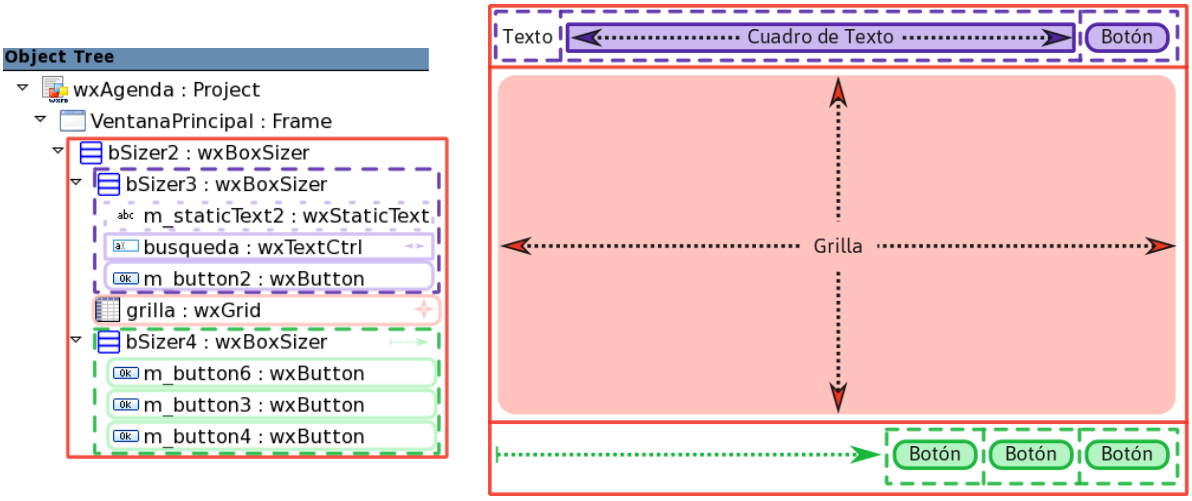
En la ventana de wxfb se distinguen las siguientes áreas:

- Árbol de objetos: muestra que componentes hay y cuál es su jerarquía.
- Paleta: contiene los elementos que podemos agregar al árbol. 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 ya existente se que halla estado seleccionado previamente.
- 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.

Seleccione la ventana del ejemplo (`wxfbExample`) en el árbol de objetos o desde su barra de título y pulse Ctrl+D para eliminarla.



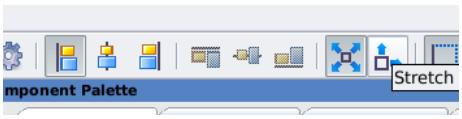
Ahora vamos a construir la ventana principal, que presentará la siguiente estructura:



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, y utilizamos el ícono de “Stretch” de la barra de herramientas).

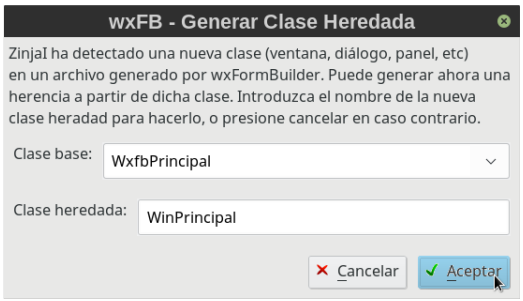


4) En el segundo lugar del sizer principal (hay que seleccionar el primer sizer en el árbol de objetos), colocamos una tabla (`wxGrid`, de la pestaña “Data” de la paleta). Para que la tabla ocupe todo el espacio libre, activamos las opciones Strech y Expand desde la barra de herramientas (ambos íconos se encuentran contiguos), y desactivamos Stretch del segundo sizer. 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 del mismo. Para este sizer también Stretch y Expand, y activamos la alineación a la derecha (“align right”, el tercer botón de la imágen anterior); para que el contenido se ubique a la derecha en lugar de estirarse. Para los rótulos de los botones, modificamos la propiedad “label”.

Finalmente, antes de guardar los cambios y regresar a ZinjaI es conveniente definir el

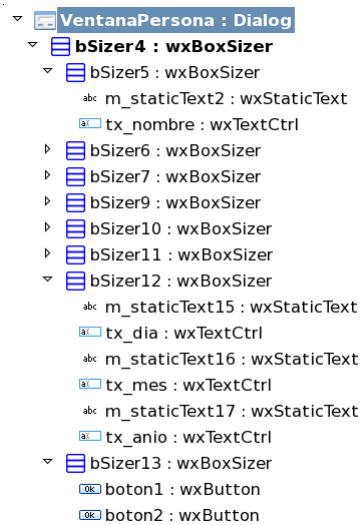
nombre de las clase que generará esta ventana. Para ello, seleccionamos la ventana en el árbol de objetos (el `wxFrame`) y cambiamos su propiedad "name" en el panel de la derecha, de "`MyFrame1`" a "`WxfbPrincipal`".

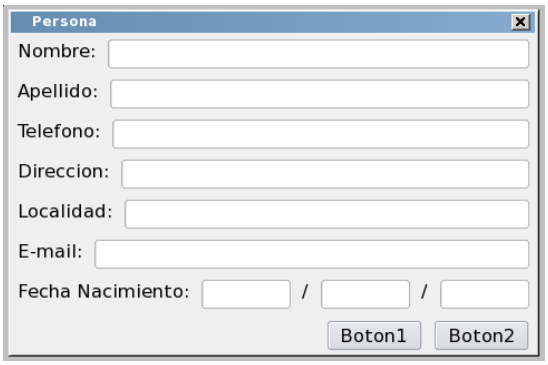


Ahora debemos guardar el archivo (menú File→Save o Ctrl+S) y retornar a ZinjaI. ZinjaI reconocerá automáticamente[2] que el archivo de wxfb ha cambiado, y hará que wxfb regenere el código fuente de `wxfb_project.cpp` y `wxfb_project.h`. Luego de que estos fuentes se actualicen, ZinjaI también detectará que hay una clase/ventana nueva y que otra clase/ventana ya no está, ofreciendo generar una nueva herencia para la primera, y eliminar la herencia de la segunda. Responderemos afirmativamente a ambos ofrecimientos, introduciendo "`WinPrincipal`" como nombre para la nueva herencia.

Recuerde que **cada ventana estará implementada en dos clases**: una generada por wxfb con la estructura, y otra heredada en la cual se implementarán los eventos. Para distinguirlas fácilmente, nombraremos a las primeras con el prefijo "Wxfb", y a las segundas con el prefijo "Win".

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





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

2 Si al abrir/crear el proyecto la configuración de wxfb no era correcta, esta integración automática estará temporalmente desactivada hasta que reinicie ZinjaI o reabra el proyecto. Puede realizar estas acciones manualmente utilizando los comandos del submenú "Diseñar interfaces" del menú "Herramientas".

principio es más fácil lograr el efecto deseado combinando objetos `wxBoxSizer` verticales y horizontales alternadamente.

Si necesitara mover un componente, debe hacerlo arrastrándolo desde el árbol 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 imagen anterior de la ventana completa.

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). Observamos que al crearla no se dibuja más que la barra de títulos. Esto es porque el tamaño del cuadro de diálogo se ajusta al contenido, pero inicialmente está vacío. A medida que incluya controles dentro de los sizers el cuadro de diálogo irá creciendo y reajustando su tamaño.

Usualmente se utiliza **wxFrame** para ventanas *comunes*/independientes, y **wxDialog** para cuadros de diálogo (ventanas *emergentes* que aparecen temporalmente por encima de otra ventana). Los dialogs tienen además la ventaja de poder mostrarse de forma *modal*, concepto que se discutirá más adelante.

De una misma clase generada por wxfb se pueden heredar luego dos o más ventanas heredadas, de igual apariencia pero comportamiento diferente. Esta ventana se va a utilizar como base para dos casos: para mostrar los datos completos de una persona, y para agregar un persona nueva o modificar una existente. Para implementarlos por separado, generaremos dos herencias: “`WinAgregar`” y “`WinEditar`”. En ZinjaI, para hacer una segunda herencia puede utilizar el comando “Generar clase heredada...” del submenú “Diseñar interfaces” del menú “Herramientas”.

Iniciando la aplicación

En una aplicación con wxWidgets, la inicialización de la aplicación se lleva a cabo en el método `OnInit` de la clase `wxApplication`, que sería de algún modo el equivalente a la función `main` (la verdadera función `main` será implementada por la biblioteca). El proyecto de ZinjaI ya cuenta con una herencia de esta clase llamada Application. Allí debemos crear la primer ventana. Para ello hacemos simplemente "`WinPrincipal *w = new WinPrincipal(nullptr);`". Debe ser creada dinámicamente (**con `new`**) para que no se destruya inmediatamente al finalizar 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 nullptr por ser la primera). Al crear una ventana, esta no se muestra, sino que solo se carga en memoria. Por esto, para que efectivamente se haga visible debemos invocar a su método `Show`.

Se debe notar que las ventanas regulares se crean con **new**, pero en ningún momento colocaremos el correspondiente **delete**. En general, la biblioteca tiene mecanismos internos para gestionar sus ventanas y controles y destruirlas correctamente al finalizar la aplicación. Sin embargo, si en un momento dado se quiere forzar la destrucción de una ventana, puede hacerse mediante su método Destroy.

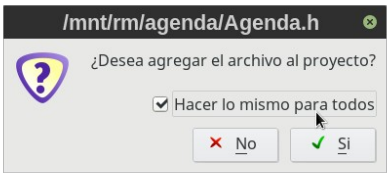
Al finalizar `OnInit`, se le pasa el control automáticamente al bucle de eventos de wxWidgets, y entonces la biblioteca comenzará a detectar eventos sobre esta primer ventana, e invocar a los métodos asociados a los mismos (asociación que aún resta definir).

A esta altura, podemos ejecutar la aplicación y ver cómo se crea y se muestra la ventana principal, aunque todavía no reaccionará a ningún evento interesante más que el de cerrarse.

Cuando la aplicación se carga también debe leer la base de datos (crear una instancia de la clase `Agenda`) para luego poder mostrar y modificar su contenido en las diferentes ventanas y cuadros de diálogo. 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 de personas. Hay varias formas de resolver este problema y cada una tiene sus ventajas y desventajas. La más simple y menos recomendable es un puntero global (un poco mejor si está escondido dentro de un **Singleton**). La que utilizaremos en este ejemplo consiste en asociar el ciclo de vida de la instancia de Agenda a una de las clases principales (puede ser la clase `Application`, o la de la ventana principal); y luego hacer que esa clase le "preste" el objeto a las demás clases/ventanas.

Singleton es un patrón de diseño muy utilizado y también muy cuestionado que garantiza que una clase tendrá solo una instancia. Para ello restringe el acceso a la clase prohibiendo su construcción directa, y proporcionando un único punto de acceso a la misma (mediante una función global o un método estático).

Vamos entonces a incluir las clases `Agenda` y `Persona` en el proyecta actual. Para ello, en ZinjaI vamos al menú "Archivo", seleccionamos "Abrir..." y elegimos los cuatro archivos (.cpp y .h de cada clase). ZinjaI preguntará si queremos agregarlos al proyecto, y ofrecerá también copiarlos a la carpeta del mismo si estaban originalmente en otra ubicación. Una vez hecho esto, debemos ver a los cuatro archivos en el árbol del proyecto, y entonces podremos incluir "`Agenda.h`" y comenzar a utilizar la clase en cuestión.



Para este ejemplo, la instancia de `Agenda` estará declarada en `Application`, como atributo, para que su ciclo de vida coincida con el de toda la aplicación (se cree al iniciar, se destruya solo al salir). Para poder inicializarla en ello, necesitaremos también agregar un constructor a `Applicación` para que le suministre al constructor de la instancia de Agenda el nombre del archivo de datos. Una vez construida la instancia, la clase `Application` deberá pasarle un puntero o una referencia de la misma a la ventana principal para que ésta última pueda consultar los datos y cargar su grilla. Entonces `Application` queda:

> Si el atributo fuera un puntero a Agenda, se podría inicializar con new en OnInit y prescindir del constructor, pero a cambio deberíamos asegurarnos de colocar el delete correspondiente. Una solución más elegante sería utilizar ***puntero inteligente***. Para saber más, busque sobre las clases std::unique_ptr y std::share_ptr.

```
// en Application.h
class Application : public wxApp {
public:
   Application(); // ctor, para inicializar m_agenda
   virtual bool OnInit(); // primer evento del programa
private:
   Agenda m_agenda;
};

// en Application.cpp
Application::Application():m_agenda("datos.dat") { // cargar los contactos

}
bool Application::OnInit() {
   WinPrincipal *v = new WinPrincipal(&m_agenda); // crear la ventana
   v->Show(); // mostrar la ventana
   return true; // true indica que todo se inicializó sin problemas
}
```

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

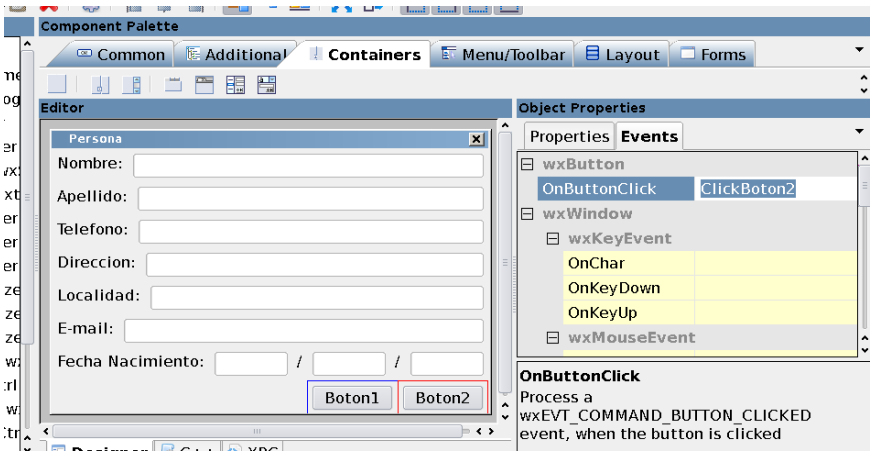
A esta altura tenemos diseñadas las dos ventanas, y e implementadas las dos clases (`Persona` y `Agenda`) con sus funciones y estructuras auxiliares. Para unir las dos partes debemos:

1) Definir que controles y eventos interesan
2) Generar las clases heredadas de las ventanas en ZinjaI
3) Codificar los eventos usando las clases ya desarrolladas
4) Compilar, ejecutar, depurar…

Primero debemos definir los nombres de los controles que nos interesará referenciar desde nuestro programa (nombres de objeto/atributo). Por ejemplo, necesitaremos referenciar la grilla para cargarle los datos de las personas, y el cuadro de texto de la búsqueda para obtener su contenido y con él realizar la búsqueda. Definimos entonce a la grilla como "`m_grilla`" y el cuadro de texto como "`m_busqueda`") seleccionando los controles y utilizando la propiedad "`name`" en el panel de propiedades de la derecha.

De igual forma, no olvide colocar los nombres adecuados a los componentes de la ventana `WxfbPersona` (los cuadros de texto como "`m_<atributo>`" y los botones como "`m_boton1`" y "`m_boton2`").

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: "`OnClickBuscar`"
  - Para el botón Agregar: "`OnClickAgregar`"
  - Para el botón Editar: "`OnClickEditar`"
  - Para el botón Eliminar: "`OnClickEliminar`"
- Click en el título de alguna columna de la tabla: Vamos a permitir reordenar la lista por diferentes columnas mediante esta acción. El evento es "`OnGridLabelRightClick`" y al método lo vamos a llamar "`OnClickGrilla`"

> Es usual utilizar el **prefijo On** para denominar a los métodos asociados a eventos. Puede verse como el equivalente en inglés para la partícula "al".

En la ventana de la Persona, los eventos que interesan son los clicks en los dos botones;

llamaremos a los métodos "`OnClickBoton1`" y "`OnClickBoton2`".

Una vez definidos los eventos, guardamos el archivo (menú File→Save o Ctrl+S) y retornamos a ZinjaI. ZinjaI, luego de hacer que wxfb regenere el código fuente de `wxfb_project.cpp` y `wxfb_project.h`, detectará que hay métodos nuevos en las clases base y agregara automáticamente dichos métodos en las herencias. Si abre algunos de los cpps Win* verá los nuevos métodos al final. Estos usualmente contendrán "`evt.Skip()`", que es la forma de decirle a wxWidgets que ignore nuestro código y prosiga de la forma habitual (que generalmente implicará no hacer nada).

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 de la instancia de `Agenda` a la grilla. Pero para ello, deberá recibir el puntero a dicha instancia (enviado desde `Application::OnInit`), y además será conveniente guardarlo como atributo en la clase para poder utilizarlo luego en otros métodos/eventos de la misma. Entonces, el constructor de `WinPrincipal.cpp` sería:

```
WinPrincipal::WinPrincipal(Agenda *agenda) : m_agenda(agenda) {
   int c_pers = m_agenda->CantidadDatos(); // cantidad de personas
   m_grilla->AppendRows(c_pers); // agregar tantas filas como registros
   for (int i=0;i<c_pers;i++) CargarFila(i); // cargar todos los datos
   m_grilla->SetSelectionMode(wxGrid::wxGridSelectRows);
}
```

Notar que se modificó el prototipo del constructor que había generado ZinjaI para la herencia `WinPrincipal`. Esto es válido (deberá modificarse de igual manera en el .h), y puede servir para que una ventana reciba información adicional al crearse. Además, el constructor utiliza una método auxiliar (que puede declararse en la parte privada de `WinPrincipal`) para cargar una fila de la grilla. Esta separación será útil cuando otros eventos también deban actualizar la grilla.

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

Con el métodos `AppendRows` de grilla modificamos la cantidad de renglones para que coincida con la cantidad de registros. Luego, con `SetCellValue` cargamos los datos en cada celda de la tabla. 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 todos los métodos y propiedades de un objeto consulte la documentación de la clase en la referencia oficial de wxWidgets; en ZinjaI, menu Herramientas->Diseñar Interfaces->Ayuda wxWidgets.

Utilizaremos tres funciones (**wx_to_std**, **std_to_wx** y **c_to_wx**) para las conversiones entre los distintos tipos de cadenas (std::string, wxString y char*). Más adelante se definirán estas funciones y se discutirá su necesidad.

Para poder agregar registros debemos definir el comportamiento rotulado "`Agregar`". Este botón debería crear una instancia de la clase `WinAgregar`, 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 `WinAgregar`, o la puede hacer la ventana de `WinAgregar` 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 método que llamó a la segunda) 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 una 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 WinPrincipal::ClickAgregar( wxCommandEvent& event ) {
   WinAgregar nueva_ventana(this); // crear la ventana
   if (nueva_ventana.ShowModal()==1) { // mostrar y esperar
      m_grilla->AppendRows(1); // agregar el lugar en la grilla
      CargarFila(m_agenda->CantidadDatos()-1); // copiar en la grilla
   }
}
```

y dejamos la tarea de actualizar la base de datos (objeto `m_agenda`) al botón Aceptar de la ventana `WinAgregar`. Es importante destacar que debido al comportamiento de `ShowModal`, sólo en estos casos (cuadros de diálogos modales) tiene sentido construir una ventana de forma estática. El código para el botón Aceptar del `WinAgregar`, considerando lo que desarrollamos en la clase persona para la validación, puede ser:

```
void WinAgregar::ClickBoton2( wxCommandEvent& event ) {
   long dia, mes, anio; // convertir cadenas a numeros
   m_dia->GetValue().ToLong(&dia);
   m_mes->GetValue().ToLong(&mes);
   m_anio->GetValue().ToLong(&anio);
   Persona p( // crear la instancia de persona
      wx_to_std(m_nombre->GetValue()),
      wx_to_std(m_apellido->GetValue()),
      wx_to_std(m_telefono->GetValue()),
      wx_to_std(m_direccion->GetValue()),
      wx_to_std(m_localidad->GetValue()),
      wx_to_std(m_email->GetValue()),
      dia,mes,anio);
   string errores = p.ValidarDatos();
   if (errores.size()) // ver si no pasa la validacion
      wxMessageBox(std_to_wx(errores)); // mostrar errores
   else {
      m_agenda->AgregarPersona(p);
      m_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.

Supondremos ahora que el botón “`boton2`” sirve para agregar, mientras que “`boton1`” para cancelar la operación. Esto se puede definir en el constructor:

```
WinAgregar::WinAgregar(wxWindow *parent) : WxfbPersona(parent) {
```

```
   SetTitle("Agregar Persona"); // titulo de la ventana
   m_boton1->SetLabel("Cancelar"); // rotulo boton 1
   m_boton2->SetLabel("Guardar"); // rotulo boton 2
}
```

Como se vió antes, el método `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 WinAgregar::ClickBoton1( wxCommandEvent& event ) {
   EndModal(0);
}
```

Si el orden de las acciones hubiese sido otro y luego de generar las clases heredadas, se hubiese tenido que modificar los eventos de interés en el diseñador, al volver a ZinjaI, éste agregaría automáticamente los nuevos métodos en las clases hijas.

Se deja como ejercicio para el lector el desarrollo de la clase `WinEditar` y su llamada desde el método `ClickEditar` de `WinPrincipal`. 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. Recuerde que para saber cual es el índice de registro, se puede utilizar el índice de la fila seleccionada.

La clase **wxString tiene sobrecargas para el operador** `<<` que permiten concatenar fácilmente variables de otros tipos (como int), de forma similar a un stringstream. Ej:

```
wxMessageBox( wxString()
    << “El promedio es: “
    << sum/cant );
```

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 WinPrincipal::ClickBuscar( wxCommandEvent& event ) {
   int fila_actual = m_grilla->GetGridCursorRow();
   int res = m_agenda->BuscarApellidoYNombre(
                                    wx_to_std(m_busqueda->GetValue()),
                                    fila_actual);
   if (res==NO_SE_ENCUENTRA) {
      wxMessageBox(c_to_wx("No se encontraron más coincidencias"));
   } else {
      m_grilla->SetGridCursor(res,0); // seleccionar celda
      m_grilla->SelectRow(res); // marcar toda la fila
      m_grilla->MakeCellVisible(res,0); // asegurarse de que se ve
   }
}
```

Notar que esta búsqueda simplemente selecciona el registro encontrado. Sería mejor la interfaz si se filtrara la lista, mostrando solo los resultados coincidentes. Implementar este mecanismo requiere algo más de trabajo, pero además agrega el problema de que ya no habrá coincidencias entre el índice del registro y el número de fila en la grilla.

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 WinPrincipal::ClickGrilla( wxGridEvent& event ) {
   int columna = event.GetCol(), c_pers = m_agenda->CantidadDatos();
   switch(columna) { // ordenar en m_agenda
      case 0: m_agenda->Ordenar(ORDEN_APELLIDO_Y_NOMBRE); break;
      case 1: m_agenda->Ordenar(ORDEN_DIRECCION); break;
      case 2: m_agenda->Ordenar(ORDEN_TELEFONO); break;
      case 3: m_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 WinPrincipal::ClickEliminar( wxCommandEvent& event ) {
   int fila_actual = m_grilla->GetGridCursorRow();
   int res = wxMessageBox(c_to_wx("¿Eliminar el registro?"),
                          m_grilla->GetCellValue(fila_actual,0),wxYES_NO);
   if (res==wxYES) {
      m_grilla->DeleteRows(fila_actual,1);
      m_agenda->EliminarPersona(fila_actual);
      m_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.

Sobre la codificación de cadenas y caracteres

Usualmente en ZinjaI y con std::string, los caracteres se almacenan más o menos, según el código ascii (en realidad, según una norma ISO-8859-15). En esta forma de codificación, cada caracter se codifica con un solo byte (con un `char`), y entonces solo tenemos 255 caracteres posibles. Cuando queremos poder representar más caracteres (símbolos matemáticos, letras con acentos/tildes/diéresis/letras griegas/arábicas/chinas/etc) tenemos que pasar a usar otras codificaciones más complejas, donde los caracteres (a veces todos, a veces solo los especiales, según la codificación) ocupen más de un byte. wxWidgets utiliza internamente una de estas más complejas (usualmente UTF-8). Las funciones `std_to_wx`, `wx_to_std` y `c_to_wx` hacen las conversiones necesarias para pasar de una codificación a otra. En ambas coinciden los primeros 127 caracteres (porque la tabla ASCII original era de 7 bits por caracter), pero varían en el resto. Por esto, si no se hace la conversión adecuada, se obtendrá el resultado esperado solo si la cadena no tiene caracteres "especiales", pero parecerá vacío en caso contrario.

La implementación de estas funciones para este caso (en que el código fuente y los archivos de datos guardan ISO-8859-15), es:

```
/// para convertir un c-string, o una constante, a wxString
inline wxString c_to_wx(const char *c_str) {
   return wxString::From8BitData(c_str);
}
/// para convertir un std::string a wxString
inline wxString std_to_wx(const std::string &std_str) {
   return wxString::From8BitData(std_str.c_str());
}
/// para convertir wxString a std::string
inline std::string wx_to_std(const wxString &wx_str) {
   return static_cast<const char*>(wx_str.To8BitData());
}
```

Códigos Unicode de caracteres usuales en español:

```
á: 00E1   é: 00E9
í: 00ED   ó: 00F3
ú: 00FA   ñ: 00F1
¿: 00BF   ¡: 00A1
Ü: 00FC
```

El código fuente o los archivos de datos podrían estar codificados de manera diferente y entonces las conversiones serían otras. Para el código fuente la mejor solución es limitar el código a solo caracteres ASCII (los primeros 127), de modo que no importe la codificación, el resultado sea el mismo. Si se hace esto, para introducir en una cadena los caracteres especiales, se deben utilizar secuencias de escape; y para que la cadena los acepte deberá ser de tipo `std::wstring` en lugar `std::string` (que usar internamente un arreglo de `wchar_t` en lugar de `char`). Los caracteres Unicode se incluyen en la constante como `\uXXXX` donde **`XXXX`** es el código del caracter. wxWidgets incluye además un equivalente simple para la función `c_to_wx`, denominado `_T`. Por ejemplo, para mostrar el mensaje "`acentos: á é í ó ú, y otros: ñ ¿ ¡ ü`", se puede utilizar:

```
wxMessageBox( _T("acentos: \u00E1 \u00E9 \u00ED \u00F3 \u00FA,
                  y otros: \u00F1 \u00BF \u00A1 \u00FC") );
```

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 director io samples) y las búsquedas en Google (wxWidgets se utiliza mucho en el mundo del software libre, por lo que se pueden encontrar cientos de foros al respecto).

Los ejemplos y la documentación oficial se pueden descargar desde http://www.wxwidgets.org/downloads/. Si tiene el complemento actualizado en ZinjaI puede acceder a la referencia de clases y funciones de wxWidgets con el comando “Referencia wxWidgets” del submenú “Diseñar Interfaces” del menú “Herramientas”.

En la sección “Documentación” del sitio de ZinjaI puede encontrar además un enlace a un listado de preguntas frecuentes (http://zinjai.sourceforge.net/index.php?page=poo-faq.html) que incluye tanto preguntas de carácter general respecto a C++, como preguntas particularmente relacionadas wxWidgets, cómo compilar la biblioteca desde sus fuentes, utilizar temporizadores, agregar atajos de teclado, iconos en la bandeja del sistema, etc.

Además, en la misma sección se encuentra otro tutorial (más simple, pero en Inglés) que describe paso a paso y de forma más concisa cómo crear un pequeño editor de textos con ZinjaI y wxFormBuilder (http://zinjai.sourceforge.net/index.php?page=tinyeditor.html). Este segundo tutorial resuelve un ejemplo muy básico centrándose exclusivamente en el desarrollo de la interfaz, por lo que no utiliza un modelo de clases independientes.