Notus Developer Guide

10/30/03

Modern GUI/GDI template library. Notus is an attempt to build a portable GUI framework using design concepts of modern C++ libraries such as STL and boost.

Table of Contents

1 Overview

Notus is an unique GUI library that is bringing the concepts of generic programming to the GUI development process. It is an attempt to create a common generative domain model for GUI development. In this domain, the construction of GUI interfaces and interface families consists of the following steps.

1. According to the domain rules, define a set of custom interchangeable software "parts".
2. Using the custom and predefined software "parts", build custom compositions by specializing and combining C++ templates according to the domain rules.
3. A C++ compiler uses the template specializations as a set of rules for generating a highly customized GUI code.

2 Main Concepts

2.1 Overview.

The library main concept is Data/Display/View (DDV). In short, 'views' connect together application data and a display that can present the data to the user and accept user input.

The 'view' type is declared as

```
template<

      //view display
      typename Disp,

      //view strategies (none by default)
      typename Strategies = boost::tuples::tuple<>,

      //view data
      typename Data = typename Disp::display_data

> struct view;
```

'Display' is the thing that can present 'Data' to the user. 'Strategies' are application defined types that process view events. 'Strategies' must be combined into boost::tuple. The strategy tuples can be nested.

```
      typedef boost::tuple< strategy1, strategy2 > my_strategies;
      view<display, boost::tuple<strategy_x, my_strategies> > v;
```

Views can generate two event types:

- Events originated by the controlled display.
- View data update events. These events are generated when the application modifies the view's data.

There are two methods for an application to process view events.

1. Register a callback (view::connect).
2. Supply a strategy.

The callback method is typically used for building vertical class hierarchies. Strategies is a good way to extend and customized GUI elements horizontally.

2.2 Strategies

An application can define different strategy types for different events. Here is how to define a strategy that will be called for mouse events.

```
struct my_strategy : handle_event< events::mouse >
{
   template< typename View >
   bool on_event( View& v, events::mouse& event );
};
```

The handle_event<...> type indicates what events are handled by this strategy. One strategy type can process multiple event types. The View parameter is the view that generated this event and that the strategy is associated with.

```
struct my_strategy :
      handle_event< events::mouse >
      handle_event< events::dataupdate >
{
   template< typename View >
   bool on_event( View& v, events::mouse& event );

   template< typename View >
   bool on_event( View& v, events::dataupdate& event );
};
```

The application can define many strategies for the one view as tuple components

```
typedef tuple<some_strategy> default_strategy;
typedef view
      <
            my_display,
            tuple
            <
                  my_strategy,
                  default_strategy
            >
      >
```

2.3 View events

Views can generate two event types:

- Events originated by the controlled display.
- View data update events. These events are generated when the application modifies the view's data.

The events are defined in event.hpp

When the view dispatches the events to the connected callbacks, it first packages them as the notus::event<> type.

View strategies receive view events differently. See the 'Strategies' section.

2.4 Displays

A display can host another display. A display hosts another display if the hosting display manages the physical space of the hosted display.
For example an application can create a rectangular area in a window. The rectangular area is a display that is hosted by the window display.

NOTE: Child windows are *not* hosted by their parents. A child window has it is own display space that is independent from the parent window.

3 How It All Comes Together

To demonstrate how it all actually works, we'll discuss two sample. The first sample is a simple dialog box. The second sample is an advanced menu manager.

3.1 Modal Dialog Box

This section shows how to implement a simple modal dialog box that has the OK and system close buttons. We'll construct a custom dialog class without a single overload or polymorphic inheritance.

The sample dialog contains only one active control: the 'OK' button. For example, it could be an About dialog box.

The 'OK' button has a custom functionality that is when the user presses it, the dialog is dismissed.
According to the generic programming ideas, we need to implement a custom component that dismisses the dialog box upon button-click events. Then using the custom component and notus domain model, we should be able to compose the final dialog box DDV (see the Main Concepts section). The dialog box DDV will look as follows:

typedef view<notus::displays::dialog<>, boost::tuple< strategies > > about_dialog;

First, let's define a button strategy that closes the modal dialog.

```
struct button_strategy_close_dialog :
      notus::handle_event< notus::events::ev_button >
{
      displays::dialog *d_;

      button_strategy_close_dialog() : d_(0) {}

      template< typename View >
      void on_event(View& v, notus::events::ev_button& ev)
      {
            //button click event
            if(ev.action == notus::ev_button::click )
                  //close the modal dialog and set the exit code
                  //the button id
                  d_->end_modal(v.get_display().id());
      }
};
```

The library has a display class for the standard button control, displays::controls::pushbutton. Using this display and the button strategy we can define a button DDV that closes the modal dialog when the button is clicked.

```
typedef notus::view
     <
        displays::controls::pushbutton<>,
        boost::tuple< button_strategy_close_dialog >
     > close_dialog_button;
```

Obviously the close_dialog_button type should be instantiated by about_dialog when the dialog is initialized. This means that we need to implement a custom dialog initialization strategy. Here is how we can do this:

```
template< int BtnId >
struct dialog_strategy_close_button :
      notus::handle_event< notus::events::state >
{
      typedef typename close_dialog_button::display button_display;


      //the close button
      boost::shared_ptr<close_dialog_button> btn_;

      //process the state changes events
      template< typename View >
      bool on_event( View& v, notus::events::state& e )
      {
            typedef typename View::display display;

            //initializing state
            if( e.state() == notus::events::state::initializing )
            {
                  //attach the BtnId button to the button display
                  typename close_dialog_button::shared_display btn(
                        new button_display(
                                    *v.get_display(), //button parent
                                    BtnId, //button id
                                    true //attach to the existing button
                                    )
                              );
                  //using the button display create a button view
                  btn_.reset( new close_dialog_button(btn) );
                  //the button strategy needs a pointer
                  //to the dialog display so that when
                  //clicked, the button could close this dialog
                  button_strategy_close_dialog s( v.get_display().get()
);
                  btn_->set_strategy(s);
            }
            return true;
      }
};
```

We need one more strategy. When the user clicks the system close button, the dialog has to be closed as well. The system close command is generated as the syscommand event. It is similar to WM_SYSCOMMAND in win32.

```
struct dialog_strategy_syscommand :
      notus::handle_event< notus::events::syscommand >
{
      template< typename View >
      bool on_event( View& v, notus::events::syscommand& e )
      {
            if( e.cmd == notus::events::syscommand::close_window )
            {
                  //close the dialog and set the exit code
                  //to id_cancel
                  v.get_display()->end_modal(notus::traits::id_cancel);
            }
            return true;
      }
```

```
};
```

Using these components, we can construct the final dialog type.

```
typedef notus::view<
      notus::displays::dialog<>,
      boost::tuples::tuple<
            dialog_strategy_close_button<notus::traits::id_ok>,
            dialog_strategy_close_button<notus::traits::id_cancel>,
            dialog_strategy_syscommand
            >
      > about_dialog;
```

We have implemented a custom dialog class without a single overload or polymorphic inheritance. Our custom strategy components can be reused in many other dialog constructions without changes.

3.2 Menu Manager (Advanced)

Here I'd like to show some details on how menus are implemented internally.
Some of the requirements are:

1. Individual menu items should provide a way to present some application data. menu label, bitmap, etc.
2. An application should be able to receive events that are generated by individual menu items.
3. A menu item can be created as an independent object and added to another menu item (with some restrictions such as the command menu item cannot contain other items).

Looking at the requirements, it seems that menu items can be best implemented as typical views (DDV). A menu is a list of views stacked together.
First we implement menu displays using the low-level interface that is defined by impl::menu. We need several menu displays:

menu::bar - window main menu.
menu::popup – popup menu (context sensitive menus)
menu::system – icon at the window top-left corner with commands like Minimize/Close. In Windows, they generate WM_SYSCOMMAND messages
menu::command – command menu item
menu::separator – menu separator

It is very simple.

```
template< typename Traits >
struct command : display<Traits>
{
      typedef Traits traits;
      typedef display<traits> display;
      typedef impl::menu<traits> menuimpl;
      typedef typename menuimpl::hnd hnd;
      typedef typename traits::string string;
      typedef typename menuimpl::command_info display_data;

      command() : h_(menuimpl::create(menuimpl::type_command))
      {
```

```
            menuimpl::connect( h_,
                  boost::bind( &display::eventproc, this, _1 ) );
      }

      hnd get_hnd() const { return h_; }
      void set_data( const display_data& info )
      {
            menuimpl::set_command_info( h_, info );
      }

      void enable() { menuimpl::enable(h_); }
      void disable() { menuimpl::disable(h_); }
      void check() { menuimpl::check(h_); }
      void uncheck() { menuimpl::uncheck(h_); }

      //cannot add anything to command
      template< typename M >
      void add( M& m ) { throw; }

protected:
      detail::handle<menuimpl> h_;
};
```

The application can register an event callback with the command menu display the same way as with a normal window display.
Now we can use the displays to create menu views (DDV's). So how can we use menu views to combine individual menu items into a hierarchical structure of a typical menu? To do that, we build a menu manager that is derived from tree<T> type. The tree<T> has a node and a list of children that have the same tree<T> type.
Each node in the tree can represent menu bar, popup menu, or some other menu item. Obviously command cannot contain a popup menu, so the menu manager will generate an exception if application tried to do it. The menu manager is declared as:

```
enum separator_type
{
      menu_separator
};

template< typename Traits=notus::traits,
      //strategies
      typename S = boost::tuples::tuple< strategies::dataupdate<> > >
      struct menu :
            notus::protected_container< std::list< menu<Traits,S> > >
{
      typedef menu<Traits,S> menu_t;
      typedef notus::protected_container< std::list<menu_t> > list;

      typedef Traits traits;
      typedef typename traits::string string;
      typedef typename impl::menu<traits> menuimpl;
      //menu views
      typedef view<displays::menu::bar<traits>, S> bar;
      typedef view<displays::menu::popup<traits>, S> popup;
      typedef view<displays::menu::system<traits>, S> system;
      typedef view<displays::menu::command<traits>, S> command;
      typedef view<displays::menu::separator<traits>, S> separator;

      typedef boost::shared_ptr<bar> bar_pnt;
```

```
typedef boost::shared_ptr<popup> popup_pnt;
typedef boost::shared_ptr<system> system_pnt;
typedef boost::shared_ptr<command> command_pnt;
typedef boost::shared_ptr<separator> separator_pnt;
typedef boost::variant
<
      bar_pnt,
      popup_pnt,
      system_pnt,
      command_pnt,
      separator_pnt
> node_type;

menu() :
      d_( bar_pnt(new bar) ) {}
menu( const string& label ) :
      d_( popup_pnt(new popup) )
      {
            *get<popup_pnt>() = menuimpl::menu_info(label);
      }
menu( const string& label, int id,
      int accel_key = 0, key_flag kf = kf_none ) :
      d_( command_pnt(new command) )
{
      *get<command_pnt>() =
            menuimpl::command_info(label,id,accel_key, kf);
}
menu( const string& label, int id,
      typename command::slot_type c ) :
      d_( command_pnt(new command) )
{
      command_pnt cmd( get<command_pnt>() );
      *cmd = menuimpl::command_info(label,id);
      cmd->connect( c );
}
menu( const string& label, int id,
      int accel_key, key_flag kf,
      typename command::slot_type c ) :
      d_( command_pnt(new command) )
{
      command_pnt cmd( get<command_pnt>() );
      *cmd = menuimpl::command_info(label,id,accel_key, kf);
      cmd->connect( c );
}
menu( separator_type ) :
      d_( separator_pnt(new separator) ) {}
menu( const menu_t& t ) : list(t), d_(t.d_) {}

const node_type& data() const { return d_; }

template< typename R >
R& get() { return boost::get<R>(d_); }
bar& get_bar() { return *get<bar_pnt>(); }
popup& get_popup() { return *get<popup_pnt>(); }

menu_t& operator+( const menu_t& rhs )
{
      binary_variant_visitor<menu_t> v(*this);
      boost::apply_visitor(v, data(), rhs.data() );
      push_back(rhs);
      return *this;
```

```
	}

protected:
	node_type d_; //data

	//called by the operator+()
	template< typename T, typename U >
	void visit( T& parent, U& child )
	{
		parent->get_display().add( child->get_display() );
	}

	friend struct binary_variant_visitor<menu_t>;
};
```

4 Samples

4.1 Hello World!

```
#include "notus/platform/win32/traits.hpp"
#include "notus/impl/entries.hpp"
#include "notus/display/window.hpp"
#include "notus/view.hpp"
#include "notus/strategy/stdstrategies.hpp"

using namespace notus;

typedef
view< displays::window_frame<>, //display
	boost::tuples::tuple<
		strategies::dataupdate<>, //process data changes
		strategies::main_window //handle the exit command, etc.
		>
	> main_window;

int notus::main( int argc, char *argv[] )
{
	main_window main;
	main = "Hello World!";
	rungui();
	return 0;
};
```

4.2 Menus

The notus_test sample has a menu. You can look at the source code for more details. Here is how the menu is declared.

```
enum commands
{
	//file menu commands
	cmd_file_new_flower,
	cmd_file_new_swan,
```

```
      cmd_file_new_helix,
      cmd_file_exit
};

menu_ //set menu
(
  menu()+ //menu bar
  (
    menu("&File")+  //popup menu
    (
      menu("&New")+ //popup menu
        menu("&Flower\tCtrl+F", cmd_file_new_flower,
             'F', notus::kf_ctrl, //accellerator
             boost::bind(&app::on_file_new_flower, this, _1) )+
        menu("&Swan Nebula\tCtrl+S", cmd_file_new_swan,
             'S', notus::kf_ctrl,
             boost::bind(&app::on_file_new_swan, this, _1) )+
        menu("&Helix Nebula\tCtrl+H", cmd_file_new_helix,
             'H', notus::kf_ctrl,
             boost::bind(&app::on_file_new_helix, this, _1) )
    )+
    menu(notus::managers::menu_separator)+
    menu("&Exit",cmd_file_exit,
         boost::bind(&app::on_exit, this, _1) )
  )
)
```

5 Low-Level Interface

5.1 Overview

A common set of low-level interfaces is defined in the 'impl' namespace. The interfaces are parameterized by a traits template parameter. For win32 platform, the impl types will be parametrized by win32 traits; for wxWindows, by wxWindow traits and so on. For example the window interfaces are defined as follows:

```
namespace notus
{
namespace impl
{

//    window policies
template< typename Traits >
struct window
{
      typedef Traits traits;
      typedef typename traits::point point;
      typedef typename traits::rect rect;
      typedef typename traits::size size;
      typedef typename traits::hnd_window hnd;
      typedef typename traits::string string;
      typedef boost::signals::connection connection;
      typedef notus::impl::event<traits> event;
      typedef boost::function< void (event& e) > callback;

      static hnd nullhnd();
```

```
        static void destroy( hnd h );

        static void set_title( hnd h, const string& txt );
        static void set_menu( hnd h,
                              typename Traits::hnd_menu hm,
                              const point& pnt = point() );
        static void remove_menu( hnd h );

        //call default event handler
        static void defproc( hnd h, event& e );

        static size get_clientsize( hnd h );
        static size get_totalsize( hnd h );
        static void set_totalsize( hnd h, const size& s );

        static void repaint( hnd h );  //clear window
        static void repaint( hnd h, const rect&r );  //clear rectangle

        static void capture_input( hnd h );
        static void release_capture( hnd h );
        };
};
};
```

As you can see from this example, the library is not concerned with the internal structure of the low-level objects. The structure is platform dependent and is hidden from the library behind a handle. Typically the low-level interfaces are defined as a set of *static* functions operating on these handles.

5.2 Events

All low-level objects that can generate events must package them into the impl::event<Traits> type. If a low-level object generates events, it usually provide the 'connect' method that takes boost::function< void (impl::event<Traits>&)); as a parameter (thanks to Brock Peabody for the suggestion).
The low-level events are defined in the impl/event.hpp file.

6 Reference

Todo

6.1 Menus

Typically menu items send the command events to the parent window. In notus, menu items are individual display objects. The program can connect callback functions to the menu displays just like to any other display. For example, the declaration:

```
menu("&Flower\tCtrl+F", cmd_file_new_flower,
     'F', notus::kf_ctrl, //accellerator
      boost::bind(&app::on_file_new_flower, this, _1)
```

creates a menu item, “Flower” with accelerator, Ctrl+F and connects the application's on_file_new_flower method to it. This method will receive the menu item events.

Todo

7 Acknowledgments

Brock Peabody
J.F.K.
Boost community