# Porting the DDD debugger to GTK.

Peter Wainwright

# Porting the DDD debugger to GTK.

Peter Wainwright

# Table of Contents

# Introduction

DDD is a powerful graphical debugger for Linux and other Unix-like systems. It is in fact a graphical wrapper which can work with several underlying command-line debuggers, for example GDB. Actions invoked through the graphical user interface (GUI) are translated by DDD into corresponding textual commands which are sent to the underlying debugger. The output of these commands is captured by DDD and presented in a graphical fashion.

The existing versions of DDD are based on the venerable Unix GUI toolkit, X/Motif. Calls to the X/Motif libraries are scattered liberally throughout the source code. We wish to separate the GUI from the main program logic, so that development of DDD can proceed without requiring simultaneous updates to `N` separate platform-specific implementations.

In order to do this we will introduce a "shim" or "wrapper" which will connect DDD to the underlying GUI toolkit. In order to implement DDD using another toolkit, the only necessary modification is the creation of a new wrapper. Modifications to the DDD core logic will automatically affect all implementations.

Each wrapper will thus have two interfaces:

- The interface to DDD, or "upper" interface.
- The interface to the underlying toolkit, or "lower" interface.

The specification of the upper interface will be common to all wrappers. It will have the following properties:

- It will be object-oriented. Widgets will be C++ objects, they will be created and destroyed by C++ constructors and destructors respectively.
- Callbacks will be handled using the libsigc++ typesafe signal framework. Each widget class supports a number of "signals"; a signal is usually triggered by an event in the underlying toolkit, for example, a mouse click on a button or menu item. Callbacks will be handled by connecting a signal to a corresponding "slot", for example a class method or ordinary function.

The lower interface is the API of the underlying GUI toolkit, for example Motif, gtkmm or Qt. The bulk of the wrapper code will do two things:

- Intercept calls made by DDD to the upper interface and make corresponding calls to the underlying toolkit.
- Trap the callbacks produced by the underlying toolkit and translate them to signal invocations at the upper interface.

There is a considerable degree of freedom available in choosing the specification of the upper interface. However, since the first new target is gtkmm, it seems reasonable that the interface should look much like that of gtkmm. The classes, constructors, methods and signals have rather similar names and semantics. Where there are substantial differences, these are usually driven by the needs of the underlying toolkit. For example, gtk(mm) widgets can be created without reference to any "parent" widget – a separate call (`Container::add`) is made subsequently to set the parent. By contrast, Motif (and Qt?) widgets require a parent widget as an argument to the constructor. Therefore, our common upper interface is forced require the "parent" argument.

In general, the design of the upper interface should be such that it is relatively straighforward to write an implementation using the most popular underlying toolkits including Motif, gtkmm and Qt. The Qt port is presently only a theoretical possibility, but should be borne in mind as the interface evolves.

At present, two (incomplete) wrappers are provided in the gddd branch:

- GtkX, in the subdirectory `gtkx`.
- Xmmm, in the subdirectory `xmmm`.

Each of these wrappers provides a library (`libGtkX` or `libXmmm` respectively) which implements something like the upper interface described in the previous paragraphs. These implementations are found in the C++ namespaces `GtkX` and `Xmmm` respectively. There are (or should be!) corresponding widget classes in each namespace. The source code is configured by running `./configure` with the option `--with-gtk` or `--with-xmmm`, thus setting the macro `GUI` to have the value `GtkX` or `Xmmm` in `config.h`.

However, if neither of these options is given, *the original (Motif) code will be compiled, including direct calls to `libXm`*. This is available as a fallback. You will find the old code enclosed in a conditional, thus:

```
#if defined(IF_XM)
  <old code>
#else
  <new code>
#endif
```

The old code can be used as a guide when writing the cross-platform GUI calls. Insofar as this is possible we want to implement all the features relevant to DDD through the new cross-platform interface.

## Example

To create a button and assign a callback:

```
void my_callback_function(void);
GUI::Button *b = new GUI::Button(*parent, "name", "label");
b->signal_clicked().connect(sigc::ptr_fun(my_callback_function));
b->show();
```

This looks remarkably like gtkmm (save for the additional “parent” argument in the constructor). But if `GUI` expands to `Xmmm` it is implemented as Motif!

# Chapter 1. Cross-platform GUI interface

Here follows a description of the major classes and methods within the parallel namespaces `GtkX` and `Xmmm`. Either namespace will be referred to as the `GUI` namespace – The macro `GUI` may point to either, depending on which widget set was chosen when `./configure` was run.

The classes fall into three categories:

- Non-widget classes (utililty classes).
- Abstract widget classes (contain pure virtual functions, cannot be instantiated).
- Concrete widget classes – these generally correspond closely to the instantiable classes of the underlying widget set.

Each of these categories will be discussed in a subsequent section. The final section outlines how the DDD interface is built using the routines in `MakeMenu.C` and how these are used with the cross-platform GUI.

## Utility classes

## Abstract widget classes

### GUI::Widget

`GUI::Widget` is the parent of all widget classes. It has the following methods:

GUI::Container ***get_parent**();*

GUI::RefPtr<GUI::Display> **get_display**();

void **show**();

void **hide**();

bool **is_visible**();

bool **is_realized**();

GUI::String **get_name**();

void **set_name**(*name*);

const GUI::String &*name*;

void **set_sensitive**(*on_off*);

`bool` *`on_off`*`;`

`void` **`add_destroy_notify_callback`**`(`*`data`*`,` *`f`*`);`

```
void *data;
void *(*f)(void *);
```

`void` **`remove_destroy_notify_callback`**`(`*`data`*`);`

```
void *data;
```

`bool` **`translate_coordinates`**`(`*`dest_w`*`,` *`src_x`*`,` *`src_y`*`,` *`dest_x`*`,` *`dest_y`*`);`

```
GUI::Widget &dest_w;
int src_x;
int src_y;
int &dest_x;
int &dest_y;
```

These methods correspond one-to-one with methods in the gtkmm widget set.

For the Motif implementation, note that `show`/`hide` correspond to `XtManageChild` and `XtUnmanageChild`. `add_destroy_notify_callback` and `remove_destroy_notify_callback` are implemented using the Xt callback mechanism and the `XtNdestroyCallback` callback.

## GUI::Container

`GUI::Container` is the parent of all container classes. It has the following methods:

`ChildList` **`get_children`**`();`

`void` **`remove`**`(`*`w`*`);`

```
Widget &w;
```

All widget constructors, other than those for top-level windows, require a reference to an existing `Container` in the first argument.

# Concrete widget classes

The following concrete widget classes are implemented:

- `HBox`
- `VBox`
- `Button`
- `ComboBoxEntryText`
- `RadioButton`
- `Dialog`
- `Window`

- `FileSelectionDialog`
- `FileSelection`
- `ListView`
- `MenuBar`
- `PopupMenu`
- `PulldownMenu`
- `MultiPaned`
- `Notebook`
- `RadioBox`
- `RadioButton`
- `ScrolledText`
- `ScrolledWindow`
- `SelectionDialog`
- `Toolbar`

Most of these correspond more or less directly to a widget in the underlying toolkit. A few are realized as composite widgets. For example, the GTK `Paned` widget can only have two panes; in this case the `MultiPaned` widget is defined as a nested structure built of GTK panes.

## The DDD Menu Builder

Although individual widgets can be created directly using the constructors for the above classes, most of the DDD interface is built using the routines in `MakeMenu.C`. Routines such as `MMaddItems` may be used to populate a menu by parsing an array of `MMDesc` structures. Each `MMDesc` structure describes one widget. It should be constructed using one of the `GENTRY` macros variants:

```
GENTRYL(name, label, type, callback1, callback2, submenu, widget_ptr)
GENTRYI(name, icon, type, callback1, callback2, submenu, widget_ptr)
GENTRYLI(name, label, icon, type, callback1, callback2, submenu,
         widget_ptr)
```

where:

- `name` is the name of the widget
- `label` is the text to be drawn on the widget (unused for some widgets)
- `icon` is the icon to be drawn on the widget (can be used in addition or in place of the label on buttons)
- `type` is the type of the widget, for example `MMPush` or `MMMenu`
- `callback1` is the callback and associated data used by the old Motif version
- `callback2` is the callback and associated data used by the new cross-platform version

- `submenu` is a pointer to another array of `MMDesc` structures, describing the submenu if any
- `widget_ptr` is a pointer to a location where the new widget will be stored. This may be 0 unless we need to refer to the widget elsewhere.

The available widget types are these:

- `MMPush` Create PushButton (default)
- `MMToggle` Create ToggleButton
- `MMMenu` Create CascadeButton with menu
- `MMSeparator` Create Separator
- `MMLabel` Create Label
- `MMRadioMenu` Create CascadeButton with RadioBox menu
- `MMOptionMenu` Create an option menu
- `MMPanel` Create a panel
- `MMRadioPanel` Create a panel with RadioBox menu
- `MMButtonPanel` Like MMRadioPanel, but no radio behavior
- `MMScale` Create a scale
- `MMTextField` Create a text field
- `MMEnterField` Like MMTextField, but use Enter to activate
- `MMFlatPush` Create 'flat' PushButton without shadows
- `MMArrow` Create an arrow button
- `MMSpinBox` Like MMTextField, but add two spin buttons
- `MMComboBox` Create a combo box
- `MMMenuItem` Create a MenuItem (same as MMPush for Motif)
- `MMCheckItem` Create a CheckMenuItem (same as MMToggle for Motif)
- `MMRadio` Create a RadioButton (same as MMToggle for Motif)

The implementation of these types in the underlying toolkit will be seen upon examination of the source code `MakeMenu.C`.