Design of the GUI Library for the Mokka Framework and Its Implementation for Windows

Master Thesis Task Specification

Author: Ralf Reiterer

1 Introduction

Nowadays, a developer that wants to use the CLI/.NET platform has the choice between two API stacks: the Microsoft® .NET Framework stack and the Mono® stack. Both are primarily targeted for a specific platform. The first one targets Windows®, while the second one is mostly of interest for developing UNIX®-based applications, for example, applications running under GNOME.

1.1 The Mokka Framework

The Mokka Framework is an idea of the author and aims to ease the development of cross-platform CLI/.NET applications and libraries by providing a third, cross-platform API stack, in addition to the .NET Framework and Mono API stacks mentioned above, as well as cross-platform tools.

However, the term “Mokka Framework” does not refer to a single product but is intended to be an umbrella term which involves a set of .NET Framework, third party and own libraries and tools that are suitable for cross-platform software development.

The .NET Base Library (which involves all base libraries that are part of the .NET Framework such as BCL, Extended Numerics, Reflection) is almost fully portable. Only a small subset of types and members is Windows-specific. The author refers to the portable subset as the Portable Base Library (PBL) which serves as the base for the Mokka Framework libraries and tools that in turn fall into one of the following categories:

1. **Portable higher-level .NET Framework libraries**: The .NET Framework also contains some higher-level portable libraries that can be seen as part of the Mokka Framework. Two such examples are ADO.NET and ASP.NET.
2. **Cross-platform open source libraries and tools**: Examples are the SharpZip library, NHibernate, NSpring, NUnit and NAnt. These libraries and tools are cross-platform in general but may require modifications in some areas to fully achieve this goal. For example, the NUnit GUI frontend is based on Windows Forms. A port to the Mokka.Gui library might allow cross-platform use.
3. **Mokka Framework libraries and tools**: This category contains functionality that is not covered by libraries and tools of the previous two categories because they either simply do not exist or are not cross-platform.

An example is the Mokka.Gui library that should allow GUI applications to run unchanged on a variety of platforms. These applications will have the native look and feel of the respective GUI platform the user is already familiar with.

As one can see, the Mokka Framework does not fully replace the .NET Framework but actually reuse the portable part of its base library and higher-level libraries. This also gives application developers the flexibility to mix Mokka Framework libraries with unportable .NET Framework and/or platform-specific Mono libraries, instead of having to decide which of the three API stacks an application should be based on.

In other words, the Mokka Framework is not only of interest for developers of cross-platform applications but also for developers of platform-specific applications. For example, a Windows-specific .NET application may use the Mokka Framework Interop Layer for Windows to get convenient access to native Windows API functions.

Moreover, already familiar tools, such as Microsoft Visual Studio® or SharpDevelop, can also be used for developing cross-platform applications that should be based on the Mokka Framework.

One reason the name “Mokka” has been chosen was to underline the goal to bring the Java™ principles “write once, run anywhere” and “a portable and open API for (almost) every need” to the .NET world.

2 The Mokka.Gui Library

The Mokka GUI library, named Mokka.Gui, is the first own high-level part of the Mokka Framework that should be realized. The idea is to have a single API that is used by the GUI applications and implemented for various GUI platforms, e.g. the native Windows UI, Qt®, GTK+. These implementations should (ideally) be replaceable without causing a GUI application to behave differently. For example, a Linux® user may choose to use the Mokka.Gui implementation for GTK+ if he works under GNOME and the Mokka.Gui implementation for Qt if he works under KDE®. Regardless which implementation is used, the GUI applications should work as expected and adapt their look and feel accordingly because that is provided by the underlying GUI platform.

In other words, Mokka.Gui applications always have the native look and feel of the GUI platform they run on and therefore integrate themselves perfectly into the corresponding GUI environment.

2.1 The Master Thesis Task

The result of this master thesis should be a proof of concept for a GUI library that involves the design of a platform-independent API and its implementation for Windows and allows the development of standard GUI applications, i.e. applications that have a menu bar and content pane showing a text or graphics document or containing some GUI elements, which are called *widgets* in Mokka.Gui.

The concrete milestones are listed in the following:

1. **Design and implementation of a graphics library**: The library should provide basic drawing and filling operations for graphics primitives, bitmap and text drawing operations as well as abstractions for resources such as pens, brushes, fonts and bitmaps.

2. **Design of the widget class hierarchy and implementation of the widget classes**: Controls, containers and top-level windows should be represented in a way that allows to test whether a specific widget is a top-level window or a control or container by using a normal runtime type test. That of course implies that controls, containers and top-level windows have a common base class.

   Beside the top-level window, the following controls and containers should be at least provided: label, command button, check box, radio button, single-line and multiline text box, list box, panel and group box.

3. **Design and implementation of layout management**: In addition to adding the layout management feature to widgets, the following layouts should be provided:

   - `RowLayout`: arranges the children of a container in one row whose height is determined by the tallest child. The children are arranged horizontally based on their desired width. This is similar to a Qt `QHBoxLayout`.

   - `ColumnLayout`: arranges the children of a container in one column whose width is determined by the widest child. The children are arranged vertically based on their desired height. This is similar to a Qt `QVBoxLayout`.

   - `DockLayout`: Five children can be positioned. The top and bottom children get the full width of the container and their desired height. The left and right children and the central child get the height of the container minus the height of the top and bottom children. Finally, the central child gets the width of the container minus the desired width of the left and right children. This is similar to the `BorderLayout` in Java AWT.

4. **Design and implementation of the menu classes**: It should be possible to create a menu bar with menus that in turn have various kinds of menu items such as simple text-only menu items, check menu items, radio menu items, separator menu items and submenu menu items.

   In addition, it should be possible to create and assign a context menu to a widget.

5. **Design and implementation of command classes**: The general idea is to separate application commands, e.g. FileOpen, FileSave, EditCut, EditCopy, EditPaste, from their representation through GUI elements, such as menu items, toolbar buttons and command buttons, to achieve a consistent representation. In other words, all GUI elements that have the same command associated should represent it using the same label text, bitmap, tooltip text and enabled state.

   The `Command` class should represent an action that is being executed when the user clicks on a GUI element a `Command` instance is associated to and define properties such as the label text, description, tooltip text, bitmap and enabled state.

   When a command is associated to a GUI element, the GUI element should adapt its appearance and behavior by binding its respective properties (e.g. its label text and enabled state properties) to the corresponding command properties.

The `ToggleCommand` subclass should represent commands that also have a Boolean *selected state*, in addition to the Boolean enabled state. Examples are ViewToolbarVisible, ViewStatusBarVisible, ViewFontBold. The natural GUI elements for such commands would be the check box and check menu item. In addition, this class should also be used for mutually exclusive commands, i.e. a group of commands where only one of them can be selected at the same time. Examples are ViewFontSize8pt, ViewFontSize10pt, ViewFontSize12pt. The natural GUI elements for this kind of commands are the radio button and radio menu item which also have to ensure that the associated command is indeed mutually exclusive.

The command classes feature allows an application to conveniently manage the properties and state of a command at a central location and not have to worry about updating the appearance and behavior of the respective GUI elements accordingly.

In the future, this feature could be extended by adding further command properties, such as the shortcut key, adding support for commands in additional controls, such as toolbar buttons (once toolbars are included in the library), as well as providing a GUI customization feature that presents all application commands to the user and lets him decide which one he wants to have represented through menu items, toolbar buttons or other means, e.g. shortcut keys only.