

BSc PROJECT FINAL REPORT

Submitted for the BSc Software Engineering

May 2008

Building User Interface Design Tools

By

Ivan Zlatev

## **ABSTRACT**

The aim of this project is to further understand and explore the requirements and principles for creating rich user interface design tools. To achieve that goal it will evaluate existing approaches and implementations and based on the analysis this project will design, develop and test an abstracted framework for creating user interface design tools for the .NET platform and build a proof of concept Windows Forms designer on based on it.

# Table of Contents

1. Introduction.....	5
1.1. Project Summary.....	5
1.2. Problem Statement.....	6
1.2.1. Building Graphical User Interface Design Tools.....	6
1.2.2. Windows Forms Designing on non-Window platforms.....	7
2. Background Research.....	8
2.1. Graphical User Interface Toolkits.....	8
2.2. The Graphical User Interface Design Tool.....	10
2.3. Design-Time Concepts.....	11
2.3.1. Metadata Description.....	11
2.3.2. Type Transformation.....	12
2.3.3. State Persistence.....	12
2.3.4. Design-Time Behaviour.....	13
2.4. Design-Time Technologies.....	14
2.4.1. GObject and GTK+.....	14
2.4.1.1 Generic Type System.....	15
2.4.1.2 Property System.....	15
2.4.1.3 State Persistence.....	16
2.4.1.4 Design-Time Behaviour.....	17
2.4.1.5 Summary.....	17
2.4.2. Qt.....	18
2.4.2.1 Meta-Object System.....	19
2.4.2.2 Metadata Filtering.....	20
2.4.2.3 State Persistence.....	20
2.4.2.4 Custom Components.....	21
2.4.2.5 Summary.....	21
2.4.3. .NET Framework.....	22
2.4.3.1 Metadata.....	23
2.4.3.2 Attributes.....	24
2.4.3.3 Type Transformation.....	25
2.4.3.4 State Persistence.....	25
2.4.3.5 Windows Forms.....	25
2.4.3.6 Summary.....	26
3. Design.....	27
3.1. Architecture Overview.....	27
3.2. Loose Coupling Concepts.....	28
3.2.1. Components, Containers, Sites.....	28
3.2.2. Services.....	29
3.2.3. Design-Time Attributes.....	29
3.3. Design Surface.....	30
3.3.1. State Persistence.....	30
3.3.2. Design-Time Container.....	30
3.3.3. Designers.....	31

3.3.4. Design-Time Services.....	32
3.4. Design-Time Serialization.....	33
3.5. The Designer Tool.....	35
3.5.1. Metadata Editing.....	36
3.5.2. Component Toolbox.....	37
4. Development Plan.....	38
4.1. Goals.....	38
4.2. Milestones.....	38
4.3. Time Plan.....	40
4.4. Testing.....	41
4.4.1. Automated Test Results.....	41
4.4.2. Interaction-Based Testing Results.....	42
4.5. Challenges.....	43
4.5.1. Open Source Code Base.....	43
4.5.2. Windows Forms Input Filtering.....	43
4.5.3. Serialization.....	44
4.5.4. Designer Transactions.....	44
5. Conclusion.....	46
5.1. Implementation Summary.....	46
5.1.1. Approach.....	46
5.1.2. Deliverables.....	46
5.2. Critical Evaluation.....	48
5.3. Future Developments.....	48
6. Intellectual Property Rights.....	49
7. Bibliography.....	50
8. Appendix 1 - GObject Type.....	51
9. Appendix 2: Expected Data.....	54
10. Appendix 3 - Result Data.....	56

# 1. INTRODUCTION

## ***1.1. PROJECT SUMMARY***

The aim of this project is to further understand and explore the requirements and principles for creating rich user interface design tools. To achieve that it will evaluate existing approaches and implementations.

This project will design, develop and test an abstracted framework for creating user interface design tools for the .NET platform, written in C#. It will act as an universal back-end which will allow the transparent hosting of an arbitrary front-end - the design tool. The different design tools will be able to target a particular graphical user interface toolkit, including desktop and web based such.

The implementation will be compatible with the Microsoft® .NET one with the ultimate goal for inclusion in the Novell® sponsored Mono Project, which provides an open source implementation of the .NET Framework for Linux and other operating systems.

This project will also develop a proof of concept Windows Forms designer, which will run both on Mono and Microsoft .NET. It will enable developers to design and edit existing user interfaces in a way compatible with Microsoft Visual Studio on platforms other than Microsoft Windows.

## **1.2. PROBLEM STATEMENT**

This project will approach two different problems, which even though different in nature are still related.

### **1.2.1. Building Graphical User Interface Design Tools**

Historically designing user interfaces with a predefined set of components and being limited only to those had not been an issue. However for the past years with the introduction of richer and more sophisticated user interfaces and with the focus on code reusability and extensibility the situation has changed.

Features like creating and extending custom components, providing custom behaviour during the designing to ease the design process, the transparent code generation and much more have become a requirement for a user interface design tool.

As the background research of this project will reveal, building rich user interface design tools from the ground up in an environment like C or C++ requires a lot of preliminary foundation work mostly introducing concepts foreign to the platform in order to provide ground for implementing only the basic functionality.

Each of the evaluated user interface toolkits has approached the foundation work differently in order to deal with the development platform limitations. The result can arguably be called successful according to the nowadays standards as it either lacks the richness or requires superfluous work from the developer. The designer tool and the foundation work end up being tight coupled with the particular user interface toolkit.

This project will try to address those problems by building a framework to allow the creation of rich user interface design environments in a generic way with as minimum effort by the developers and as much extensibility as possible. The framework will be build with the managed .NET platform, which contains a large set of built-in facilities and implements foreign concepts to pre-existing technologies.

This framework will also provide a compatible implementation under an open source license of the public Design-Time Application Programming Interface defined in the Microsoft Developer Network documentation, part of the .NET Class Library and currently only available in the Microsoft .NET implementation.

## 1.2.2. Windows Forms Designing on non-Window platforms

Currently there are two major implementations of the .NET Framework - the proprietary one by Microsoft® and the Novell® sponsored community developed open source project called Mono. While Microsoft®'s implementation is available only for the Windows™ operation system, Mono is available for Linux, BSD, Microsoft® Windows™, Sun Solaris, Apple OS X and numerous architectures, including x86, x86-64, IA64, ARM, Alpha, MIPS, HPPA, PowerPC, SPARC. This makes Mono an attractive platform for cross-platform development purposes and a large number of developers take the advantage.

While source code in .NET is compiled down to bytecode and can run on any platform with the runtime installed, .NET does provide functionality that allows dynamic invocation of external native libraries from managed code. This breaks the cross-platformability on systems where the native library is not present.

This is the case with the two Integrated Development Environments available for .NET - Microsoft® Visual Studio™ and SharpDevelop. Even though they both implement the user interface designer support based on the .NET Design-Time framework, they both make massive use of that ability to invoke code in native Windows libraries and this is why they cannot run on any other operation system other than Windows. This leaves .NET developers working on other operation systems without a tool to design Windows Forms user interfaces.

This project will develop a proof of concept Windows Forms designer under an open source license that will be fully cross-platform and will be based on the .Net Design-Time framework.

## 2. BACKGROUND RESEARCH

### 2.1. GRAPHICAL USER INTERFACE TOOLKITS

Graphical User Interfaces (GUI) usually consist of a set of controls laid out in a container. The controls can be buttons, text boxes, menus and others. The container can be a window, usually referred to as a form, or a web page.

Developers can programmatically create user interfaces by directly accessing the functionality of the underlying graphical system. In the case of Microsoft Windows this is integrated in the core dynamically linked libraries and for UNIX platforms it is provided by the X Window System. In both cases the developers have to deal with very low level code, which leads to large source code bases, difficult to maintain and extend due to their complexity. For example the X Window System does not even have a concept of controls or menus. This is on purpose as it is meant to provide the functionality required for building a GUI toolkit.

A GUI toolkit provides a set of standard ready to use controls and usually also offers a way for the developers to build custom controls.

While toolkits have high level functionality, the issues of programmatically building GUI applications still remain (maintainability and extensibility). By looking at the very basic “Hello World” example in C for the GTK+ toolkit (Code Snippet 1: A “Hello World” GTK+ GUI Application, Page 9) it can be observed that even though the code is straight forward to read it does not provide a visualization of the actual end result - the user interface. This could be a real problem for applications which require rich and sophisticated user interfaces. It also makes rapid prototyping difficult, which is important for projects following the agile software development methodology.

Those problems can be solved by the use of a GUI Design tool.

```
#include <gtk/gtk.h>

int main (int argc, char *argv[])
{
    GtkWidget *window;
    GtkWidget *label;

    gtk_init (&argc, &argv);

    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
    gtk_container_set_border_width (GTK_CONTAINER (window), 20);
    gtk_window_set_title (GTK_WINDOW (window), "hello 0.1");
    gtk_window_set_default_size (GTK_WINDOW (window), 200, 50);
    g_signal_connect (G_OBJECT (window), "destroy",
                     G_CALLBACK (on_destroy), NULL);
    label = gtk_label_new ("Hello, World");
    gtk_container_add (GTK_CONTAINER (window), label);

    gtk_widget_show_all (window);
    gtk_main ();
    return 0;
}

static void on_destroy (GtkWidget *widget, gpointer data)
{
    gtk_main_quit ();
}
```



*Code Snippet 1: A “Hello World” GTK+ GUI Application*

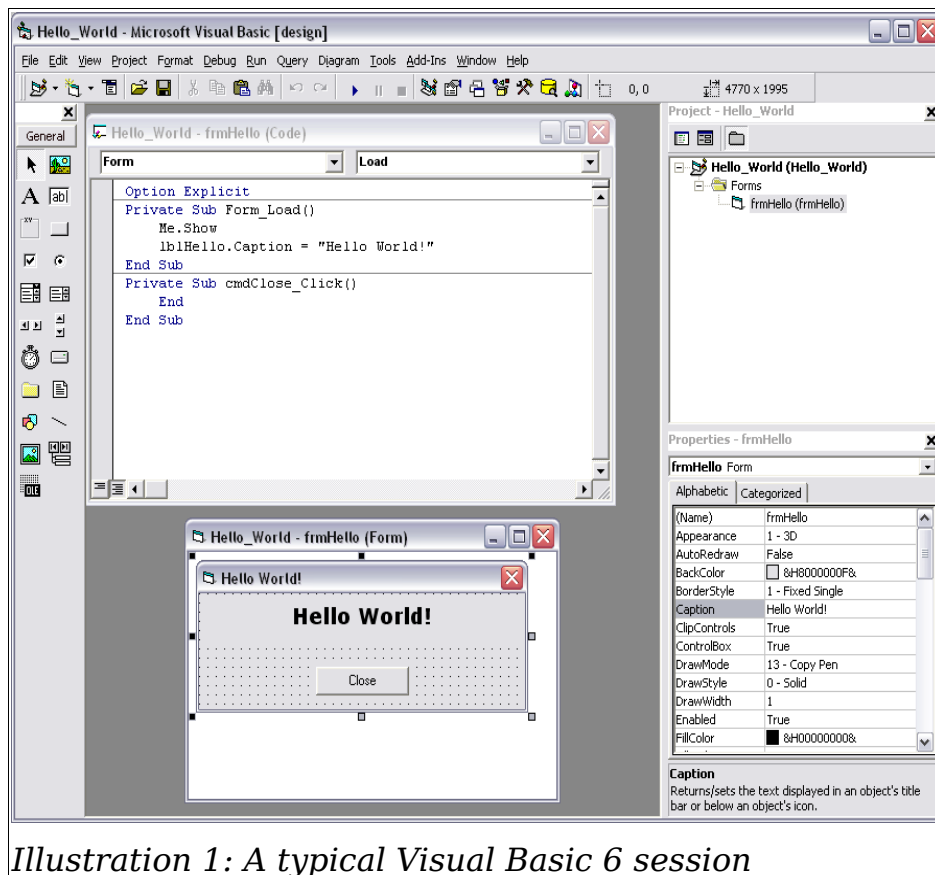
## 2.2. THE GRAPHICAL USER INTERFACE DESIGN TOOL

A graphical user interface design tool provides a fully visual way of building user interfaces.

It typically contains a toolbox with a predefined set of ready for use components and a design surface where those can be laid out. The property grid is for developers to modify and customize the characteristics of the objects, such as dimensions or location. The design tool should be able to persist the built user interface in some form, which it also would be able to load back and visualize for further editing.

In addition to those basic features a good design tool will provide dynamic help with the layout of the components, such as automatically aligning the controls to a grid, with the visual resizing and more, to the developer in a non-intrusive way. It would also expose an interface for components to customize their behaviour during design time.

A typical design tool environment can be observed on Illustration 1: A typical Visual Basic 6 session.



## 2.3. DESIGN-TIME CONCEPTS

The Design-Time concepts are the required fundamental blocks required for building a graphical user interface design tool.

### 2.3.1. Metadata Description

Each of the controls in a GUI toolkit has specific characteristics (properties). Altering those for a single instance is a way to customize it. Changing the location of a button, resizing a text box - both of those operation are just modifying an associated property of the instance - location and dimensions accordingly.

A fundamental requirement for a designer tool is to be aware of this metadata in order to provide appealing means for dynamically editing it. Possibly an editor can be associated with a property or more generically - with a type. For example a colour picker as an editor for a property "Colour" or a type Colour is a much more user friendly way to alter it than a text box requiring the user to input a RGB value.

Metadata description poses as a problem, because the languages GUI toolkits are written in are usually lower level ones (because of their high performance) which do not implement the concepts of object types or properties. This is valid for GTK+ (written in C), Qt (written in C++) and the Microsoft Windows controls.

It is possible that the list of properties for each object and their types can be hardcoded in the designer tool if the set of components in the toolkit is predefined. But this would set a constrain on the use of custom components - the designer tool won't know anything outside the scope of that list. With each update of the GUI toolkit, introducing new features and enchantments, components and/or properties might get added, removed or modified requiring the developers to synchronise the hardcoded data. Every update of the GUI toolkit would require an update of the designer tool, which increases the risk of errors and the work load.

To avoid such a situation a good solution would be if the design tool was able to request the metadata of an object type at run time. This is a concept known as reflection and is typically implemented in high level development platforms, which as already mentioned are not traditionally common for building GUI toolkits. That is why the design tools available for GTK+ and Qt have implemented their design-time support based on a custom

technologies. This report will evaluate those solutions.

### 2.3.2. Type Transformation

In an object oriented environment type transformation is the concept of performing complex conversions between types on which casting cannot be directly applied, such as an integer to string or a string (e.g. "True") to boolean.

Transformations are substantial for editing components' properties in e.g. a property grid and persisting their state, because they allow the design-time technology to convert arbitrary types and their binary representation to and from a more transparent form, such as XML or source code.

### 2.3.3. State Persistence

The values of all properties of an instance of a control describe its state. The state combined with the type of the control form an instance descriptor. An instance descriptor is an unit of metadata that contains sufficient information for a new instance of the type to be dynamically build at run-time, one equivalent to the one that the descriptor originated from.

Instance descriptors provide the foundation for state persistence, which is another key concept - the ability of the design-time technology to preserve and load back the state of the design surface or a single component at run time.

State persistence is a concept also known as serialization and deserialization and is important at minimum in the context of the following use cases:

- Preservation of the user interface metadata across sessions to allow further editing.
- Duplication of components as well as persistence of their state at a given time - useful for operations such as Copy, Paste, Undo and Redo.

#### 2.3.4. Design-Time Behaviour

At run-time the interaction with a control usually triggers an action (e.g. clicking on a ComboBox control will show its drop-down list) or will provide some sort of feedback (e.g. holding the mouse pressed over a button will typically result in a pushed-in state).

During the lay out of the controls on the design surface those actions are unimportant and actually get in the way during interaction. They have to be suppressed and replaced by the design-time behaviour - the ability to select, resize and move the controls on the surface. This can be achieved in a generic way only if the toolkit allows input redirection and filtering either per-control basis or at application level.

Design-time behaviour does not necessarily have to be limited only to drag and drop functionality. If the design-time technology provides a way for the components to know whether they are in design mode or not they could perform custom painting (e.g. draw borders around the edges of the control to make them better distinguishable) or offer extended functionality to improve the user experience.

## 2.4. DESIGN-TIME TECHNOLOGIES

To better explain the design-time concepts this report will evaluate two GUI toolkits and their design-time technology.

### 2.4.1. GObject and GTK+

GTK+ is an open source cross-platform library for creating graphical user interfaces. It provides a standard set of user interface controls such as buttons, text boxes and labels. It runs on the three major operation systems being Linux (and other UNIX-like platforms), Microsoft Windows, Apple OS X and is the GUI toolkit of choice for the GNOME Linux desktop environment. The design tool for GTK+ is called GLADE (Illustration 2: GLADE GTK+ User Interface Designer). Both GLADE and GTK+ are fully written in an object oriented manner in the C programming language and implemented on top of the GLib object system.

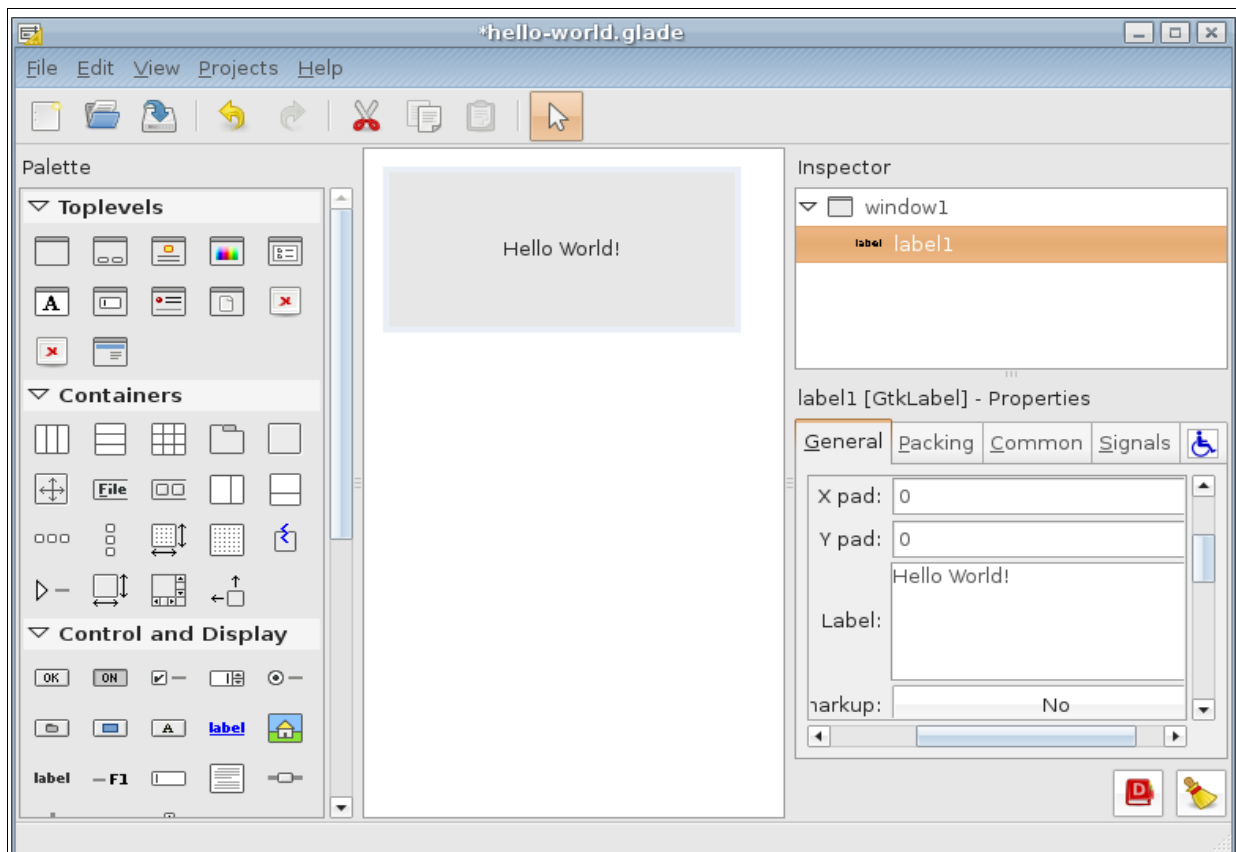


Illustration 2: GLADE GTK+ User Interface Designer

The GLib Object system (GObject) is a library that implements a flexible and extensible object-oriented framework for C. The substantial elements that are provided and are in the context of this report can be summarized as follows.

#### **2.4.1.1 Generic Type System**

GObject implements a generic type system (GType) to register arbitrary single-inherited flat and deep derived types, which provides implementations of the fundamental types, such as integers, doubles, enumerations and the base GObject type. The system handles the type instantiation, object initialization and automatic memory management based on reference counting.

As this is a foreign concept to C, the code required for the implementation just of a single object type is extremely verbose - more than a hundred lines of code (Appendix 1 - GObject Type, Page 51). The instantiation is just a matter of registering the type and a single call of the *g\_object\_new (Gtype \*type)* function.

GObject is a dynamic system - types can be registered and unregistered at run time by supplying their information to the system. The type information itself though has to be available at compile time as it includes pointers to the constructors, destructor and initializer functions of the type. Those cannot be persisted in any other form unless compiled. Instances can be created only if the type is registered, which has to happen manually.

From the perspective of the design tool this is a problem because it has to know beforehand the list of available types in order to register them with the GObject system, a list which would have to be hardcoded.

For that reason the GTK+'s designer GLADE has the major limitation of being able to handle only the standard set of GTK+ controls and does not support custom controls.

#### **2.4.1.2 Property System**

The GObject library offers a ready to use property system, which allows objects to register their properties during initialization by name and type and to associate them with a getter and setter methods to get and set the value accordingly (Appendix 1 - GObject Type , Page 51). The system allows basic retrospection by enabling listing of the property names types for an

object type.

### 2.4.1.3 State Persistence

The GObject library implements an extensible generic value system (GValue), where the developer can wrap any object type in a more abstract object of a fixed type in order to treat all object types generically - a concept known as boxing.

What is important in the context of the design time state persistence is not the boxing or unboxing themselves, but the fact that GValue adds the ability to unbox the value to an arbitrary type as long as a transformation function has been registered to convert from the contained object type to the destination type.

Combining the type transformation support of the generic value system with the reflection abilities of the generic type system provides all foundation blocks for the design-time state persistence. The designer tool can register transformation routines (e.g. for conversion to string) for all of the necessary types, iterate over all properties of an object and their values, box them in a generic value, request the conversion and then create an instance descriptor and persist it in some form.

Preferably state should be persisted in a transparent form to allow hand editing. Based on the described design GTK+'s designer GLADE serializes the example "Hello, world" design surface in a compact XML format as can be seen on Code Snippet 2: Output from GLADE, Page 16.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE glade-interface SYSTEM "glade-2.0.dtd">
<glade-interface>
  <widget class="GtkWindow" id="window1">
    <child>
      <widget class="GtkLabel" id="label1">
        <property name="visible">True</property>
        <property name="label" translatable="yes">Hello World!</property>
      </widget>
    </child>
  </widget>
</glade-interface>
```

*Code Snippet 2: Output from GLADE*

#### 2.4.1.4 *Design-Time Behaviour*

The GObject library provides a signal system which serves as a powerful notification mechanism. The GTK+ toolkit uses that mechanism to implement an event driven user interface development where one can connect listeners and respond to events such as “*clicked*” and “*focus*”.

There are three key factors that make the Design-Time behaviour and more precisely the input filtering in GTK+ possible:

1. The GTK+ toolkit is object oriented and all controls derive from a base class *GtkWidget*, which provides the core control functionality. So if anything has to be filtered it will be on that level in order to be done generically also for all derivative controls. And indeed the *GtkWidget* class has a “*button-press-event*” signal, where the designer tool can handle input.
2. The GObject signal system allows prepended connection of a listener. This means that if there are multiple event handler subscribed to the event GObject will prepend the new one at the beginning of the list, so that it gets executed first.
3. The GTK+ signal dispatcher allows an event handler to suppress the execution of the next handlers in the queue. All the handler has to do is to return TRUE as in “*event handled*”.

Combining those 3 features of GTK+ and GObject in an arguably straight forward way a developer can suppress input handling in the control and replace it with design-time behaviour.

Because GTK+ and GObject lack the concept of a design-time mode there is no way for a control to know if it's being design and thus no way to provide specific design-time behaviour on its own.

#### 2.4.1.5 *Summary*

GObject provides ready to use facilities, such as object introspection, type transformation and generic values, to implement in a very straightforward way the basic concepts required for rich design-time functionality in GTK+. It is a fascinating pieces of software, but due to its type system limitations in a dynamic context and the lack of custom components support is not very attractive for building designer tools.

## 2.4.2. Qt

Qt, like GTK+, is a cross-platform library for creating graphical user interfaces and runs on UNIX-like platforms, Microsoft Windows, Apple OS X and is the GUI toolkit of choice for the KDE Linux desktop environment. Both Qt and the Qt designer (Illustration 3: The QT Designer, Page 18) are written in C++ and implement important new design-time concepts.

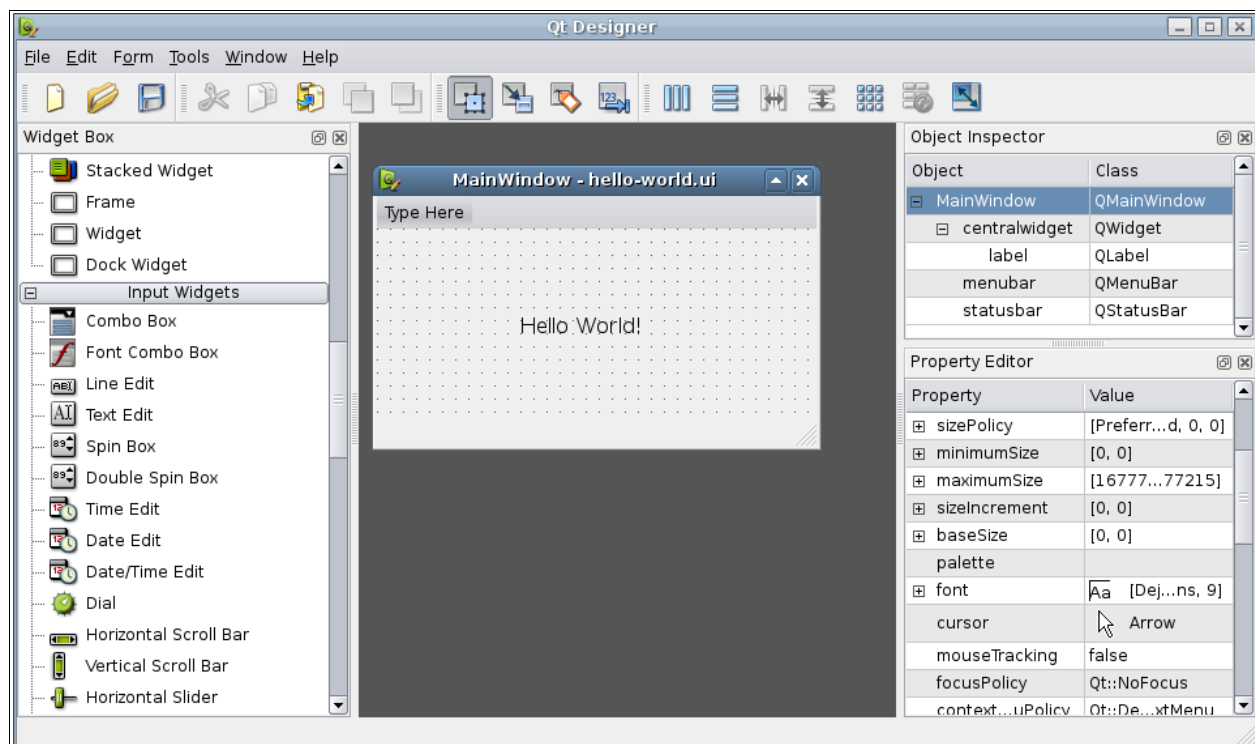


Illustration 3: The QT Designer

### 2.4.2.1 Meta-Object System

Some C++ compilers keep metadata for the object types and allow it to be accessed by the means of RTTI (Runtime Type Information or also known as Runtime Type Identification), which has to be enabled during compilation. Unfortunately RTTI merely allows one to only have access to the a string containing the class name and basic information about the inheritance tree. This information is not sufficient to implement reflection, so there is no way for a designer tool to know anything about the metadata of the components.

This problem Qt workarounds by the implementation of a Meta-Object System, which also provides full reflection functionality. It is based on three things:

- The fundamental base type QObject.
- A set of C++ extensions.
- The Meta-Object compiler.

Objects that want to participate in the metadata system have to derive from the QObject class and use the Qt C++ extension macros to register their metadata, such as their properties (Code Snippet 3: A Qt meta object definition, Page 19). Before compilation the source code has to be pre-processed by the Meta-Object compiler which parses all the extension macros and their parameters, then generates and references a separate C++ source code file with the code for the metadata registration.

```
class MyClass : public QObject
{
    Q_OBJECT
    Q_PROPERTY(Priority priority READ priority
               WRITE setPriority DESIGNABLE true)
    Q_ENUMS(Priority)

public:
    enum Priority { High, Low, VeryHigh, VeryLow };

    MyClass(QObject *parent = 0);
    ~MyClass();

    void setPriority(Priority priority);
    Priority priority() const;
};
```

*Code Snippet 3: A Qt meta object definition*

The Qt metadata system is an elegant solution to bringing full reflection functionality to the C++ platform, but it adds the dependency of the metadata compiler for source code pre-processing. Also the meta-object compiler has the following limitations:

- Inability to process C++ templates
- Inability to handle nested classes

#### **2.4.2.2 Metadata Filtering**

It is important to note that Qt is the first to offer the concept of Design-Time attributes, which it allows to be associated with the properties. In Qt only properties that are defined as *DESIGNABLE* (Code Snippet 3: A Qt meta object definition, Page 19) will be shown in the designer tool. This is very useful when the developer would like to expose only certain properties of a component and hide such that are not significant in the context of component customization during design-time, such as a *“HasFocus”* property.

This concept is known as metadata filtering.

#### **2.4.2.3 State Persistence**

Similarly to GObject, Qt implements a generic value system (QVariant) with support for type transformations, which combined with the built-in C++ support for streams and streaming and the fact that all fundamental Qt types implement streaming provides the basic blocks for implementing serialization. All properties can be reflected and then wrapped in generic values, converted to e.g. a string type and then streamed to a persistent storage.

Unlike GObject the Qt generic value system does not allow registration of custom type conversion methods and limits the developer to the predefined ones. While the list of built-in transformations includes such as ones for all basic types and even for data types like rectangle, cursor, colour and brush, there is no way for the developer to expose custom types to the designer tool for editing by the user. This is an extensibility constrain and also limits the customizability of the components on the design surface.

#### **2.4.2.4 Custom Components**

A major step forward in the Qt framework in comparison to GObject is the support for designing custom components.

Because the type system of Qt still suffers from the same constrain like GObject, being the inability to know about the existence of a type before its first instance has been created, the designer tool can only deal with a predefined set of component types.

The Qt framework deals with this limitation on the designer level. When a developer builds a custom component he can also develop and deploy a plug-in for the Qt designer. This plug-in is loaded dynamically by the designer tool from the plug-ins directory. It implements a standard interface (*QDesignerCustomWidgetInterface*) to expose information about the custom component, such as its name and type and most importantly it exports a *createWidget* method to create an instance of it.

The down-side of the whole concept is the development overhead - for each custom control built a designer tool plug-in has to be developed.

#### **2.4.2.5 Summary**

Unlike C, C++ is an object oriented language and doesn't require lengthily code to define a type as this is supported by the syntax directly (in comparison to C and GObject where it isn't).

The Qt framework makes a step forward to providing rich design-time support, by introducing the concepts of design time attributes, support for custom components and by decreasing the level of source code verbosity. The major limitation of Qt in the context of component customizability is its inability to deal with custom user types.

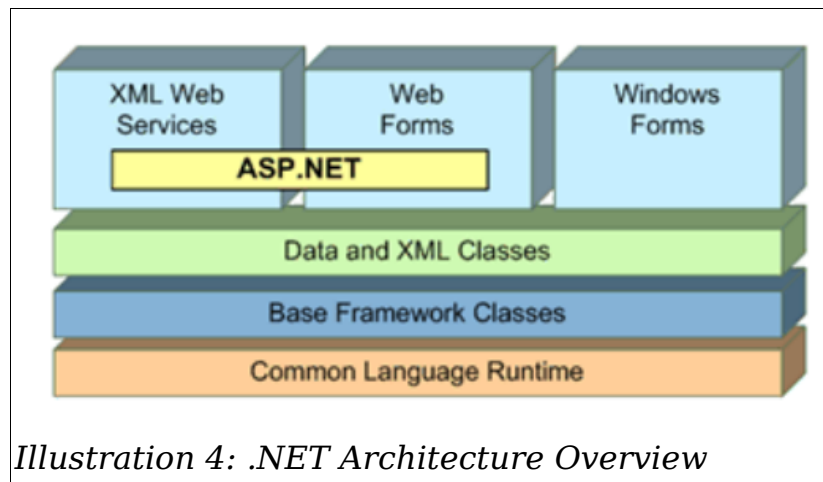
### 2.4.3. .NET Framework

The .NET Framework is a software development platform that focuses on rapid application development and platform independence.

Programs written for .NET get compiled down to bytecode known as the Common Intermediate Language (CIL) and usually referred to as managed code. Programs get executed in a virtual machine - the Common Language Runtime (CLR). The compiled code is stored either in executable assemblies (.exe) or library assemblies (.dll).

The dominating .NET programming language is C#, but there are many compiler implementations targeting the CIL and CLR, including Visual Basic.NET, Ruby, PHP, Python and more.

One of the main and largest components of the .NET Framework is the Class Library - a comprehensive, object-oriented collection of reusable types. An overview of the .NET components can be seen on Illustration 4: .NET Architecture Overview, Page 22.



*Illustration 4: .NET Architecture Overview*

There are two major implementations of the CLR and class library available - the proprietary Microsoft and the Novell sponsored community developed open source Mono Project. The latter is fully cross platform and runs on multiple operating systems such as Linux, FreeBSD and Apple OS X in contrast to first which only runs on Windows.

### 2.4.3.1 Metadata

Unlike other native platforms and languages .NET has the concept of metadata. It also has properties as a language structure. A property in C# would look like on Code Snippet 4: Property in C#, Page 23.

```
class Sample
{
    private int _propertyValue;

    public int Property {
        get { return _propertyValue; }
        set { _propertyValue = value; }
    }

    public void PropertyUser ()
    {
        Property = 5;
        int value = Property;
    }
}
```

*Code Snippet 4: Property in C#*

In .NET the compiler will automatically generate the metadata during compilation and store it in the assemblies containing the CIL. Unlike C++'s RTTI, the metadata in the CIL contains full type description information, including:

- Information for the assembly and all the types in it.
- Methods with their names, return types and parameters types
- Properties. including their names, types
- Fields with their names and type.
- Type inheritance chain
- and more.

This enables very powerful reflective programming techniques allowing not only full type description, but also invocation of methods, dynamic reflective instantiation of objects at run time and more.

Very importantly it solves the limitation of GObject and Qt of type unawareness until the first instantiation. In .NET once an assembly is loaded in memory all of the type information is loaded as well and is made available for retrospection.

If a designer tool wants to find out all available controls and derivatives of a particular type, e.g. "Button" it can just enumerate all types in all loaded assemblies and filter out all types that derive from "Button". This is a

powerful technique as it allows loading for designing arbitrary components from arbitrary assemblies at run time, without the need for the design tool to priorly be aware of their existence. Ideal for dealing with custom components.

### 2.4.3.2 Attributes

Developers can add custom metadata to their code through attributes, which sets a flag for the compiler to generate that metadata. An attribute is a regular class that inherits from the special *Attribute* class and can be used on any method, property, class or entire assembly (Code Snippet 5: Example use of an Attribute, Page 24).

```
class Sample
{
    private int _propertyValue;

    [Description ("A test Property")]
    public int Property {
        get { return _propertyValue; }
        set { _propertyValue = value; }
    }

    public void PropertyUser ()
    {
        Property = 5;
        int value = Property;
    }
}
```

*Code Snippet 5: Example use of an Attribute*

The metadata that the attributes provide is unimportant for CLR and it will be ignored during the execution of the code by the runtime. It is valuable for user applications, which can check for specific attributes using reflection and. Attributes are also used extensively by the .NET Framework itself. For example ASP.NET uses custom attributes to expose methods as web services and the Visual Studio designer checks for a *CategoryAttribute* when it's sorting the components in the toolbox.

### 2.4.3.3 Type Transformation

One of the many use cases of the custom attributes is for associating a particular type with a type converter. The *TypeConverter* in .NET is the base class from which custom type transformers can derive. Unlike GObject, where a different transformer has to be registered for each destination type for a source type, in .NET one type converter handles all supported destination types.

Another important concept related to type transformation in .NET is the type editor. Type editors take an object instance and visualize it for editing. An example of a type editor is a colour picker for a Colour property. Type Editors are derivatives of the *UITypeEdit* class and are associated with a type by the means of an attribute, which the property grid could check for and display.

It is important to note that in .NET every type is fundamentally an “*Object*”, so there is no need for a generic value system like the one available in Qt and GTK+.

### 2.4.3.4 State Persistence

Because of the powerful type conversion, type description and reflective programming abilities of the .NET platform it supports SOAP serialization, binary serialization and XML serialization out of the box.

### 2.4.3.5 Windows Forms

Windows Forms is the name given to the event-driven graphical user interface application programming interface included as a part of .NET Framework. All controls in Windows Forms derive from the base *Control* class.

Windows Forms is merely a thin wrapper around the native Windows controls API. While Windows Forms provides a higher level approach to input handling such as events/signals and polymorphism (one can subclass a control and override virtual handler methods such as *OnMouseDown*, *OnKeyPress*, etc.), it still retains the access to the lower level Windows message loop functionality.

This is useful from a design-time perspective because the heart of all native controls is the message loop, where they intercept input and other messages

sent by the operation system. More importantly Windows Forms allows a developer to redirect the message loop to an arbitrary object that implements the IWindowTarget interface and that would allow the designer tool to fully redirect the input from the control.

#### **2.4.3.6 Summary**

Due to the powerful reflection, serialization, abstraction, polymorphism as well as the extremely flexible type description and loading the .NET Framework provides the ideal platform for building a design-time framework for the creation of rich designer tool.

## 3. DESIGN

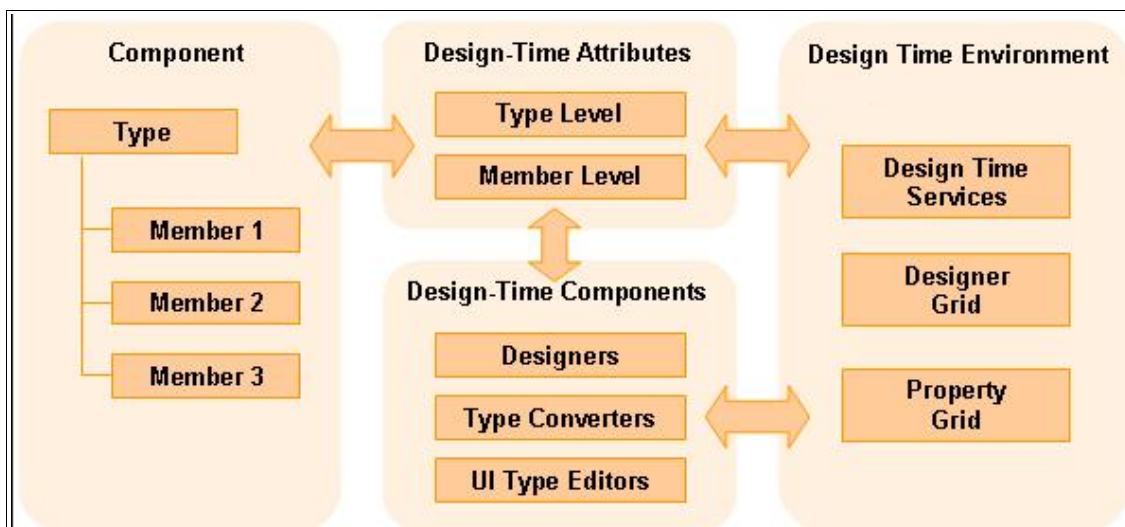
This project will design and implement a .NET Design-Time framework to allow the creation of rich user interface design environments in a generic way with as minimum effort by the developers and as much extensibility as possible. On top of this framework a proof of concept Windows Forms designer tool will be build.

### 3.1. ARCHITECTURE OVERVIEW

The .NET Design-Time framework has a highly abstracted, extensible and complex service-oriented architecture which provides loose coupling between components, containers, services and the designer tool.

In the .NET Design-Time architecture components associate their Design-Time functionality independent of the design tool. A list of Design-Time services is provided, which communicate with the components, designers and between each other. The design surface is composed by those services, the designers and the components. The designer tool hosts the design surface and makes use of the available services as well as adds additional such.

The architecture makes use of attributes with metadata to describe behaviour for components and their members. Dynamic addition, removal and modification of the metadata of the components is possible.



*Illustration 5: .NET Design-Time Framework Architecture Overview*

## 3.2. LOOSE COUPLING CONCEPTS

Loose coupling is a key concept which formed the design of the Design-Time framework due to the extensibility and modularity it provides.

### 3.2.1. Components, Containers, Sites

At the core of the Design-Time framework is an interface called *IComponent*. Anything that implements this interface is called a component. Components have three important characteristics:

1. They can be owned by/hosted in a container.
2. They can request services from the Site.
3. They can have a name assigned.

The first characteristic essentially allows the lifetime of the component to be controlled by the container. The second characteristic gives the component access to functionality provided by other parts of the framework and the designer tool. The third characteristic allows the container to assign unique identifier to each component. Components are design-time aware through their Site.

This is a fundamental concept as it provides full abstraction of visual and non visual components. Also the component can be a Windows Forms or an ASP.NET control and as long as it implements *IComponent* that will not be of significance to the design-time framework.

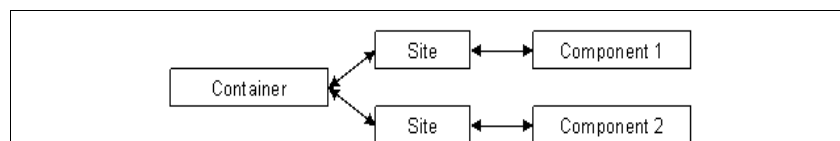


Illustration 6: Component, Site, Container Relation

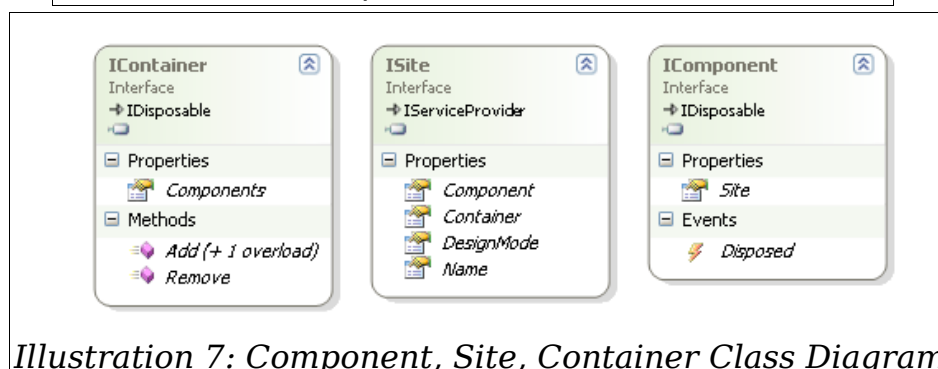


Illustration 7: Component, Site, Container Class Diagram

### 3.2.2. Services

A service is an instance of an object that implements a specific set of functionality. Services are stored in service containers and can be retrieved from service providers such as a component's Site and *IServiceProvider* implementations in general.

The power of services lies in their loose coupling: an application publishes the interface or base class that defines the service, but does not publish the class that implements the service. This design pattern allows the components to request the service and retrieve an instance they can use, but never have to know about the actual implementation nor where it comes from. In addition it

Service containers contain a table of services in the form of type-object key-value pair. To reduce the memory footprint service containers have been designed to support lazy instantiation of service objects.

### 3.2.3. Design-Time Attributes

Design-Time attributes associate a type or type member with a class that extends its design-time behaviour without creating a functional dependency for the type on the class. This framework makes use of the following attributes:

<b>Attribute</b>	<b>Description</b>
<i>DesignerAttribute</i>	Associates a type with a designer.
<i>TypeConverterAttribute</i>	Associates a type or type member with a type converter.
<i>EditorAttribute</i>	Associates a type or type member with a type editor.
<i>DesignerSerializerAttribute</i>	Associates a type with a serializer

### 3.3. DESIGN SURFACE

The *Design Surface* is a thin front-end to the .Net Design-Time framework and is designed to be instantiated and used by the designer tool directly.

The designer surface has to be initialized by the designer tool with a root component type, such as a System Windows Form or an ASP.NET Web Page and is responsible for:

- Allowing the designer tool to request the surface to be loaded and provide access to the errors if any during the loading process.
- Provide the designer tool with a visual representation of the design surface. This is the “View” that will be presented to the user.
- Provide a service container for the underlying components of the framework.
- Allow the designer tool to request the persistence of the design surface.

#### 3.3.1. State Persistence

The design surface delegates its loading and persisting to a *Designer Loader*, which is to be supplied by the designer tool during initialization. The Designer loader implementation of the tool is responsible for feeding the Design-Time Serialization system with an object graph recovered from the persistent storage as well as for persisting to the desired by the tool format an object graph resulted from the serialization of the surface. That way system maintains its format neutrality.

At load-time the Design-Time Serialization system will process the object graph to create component instances, which it will then add to the Design-Time container.

#### 3.3.2. Design-Time Container

The Design-Time container is designed to be the central storage of components in design mode. They are added by type, instantiated dynamically using reflection together with their associated designer. Each

component has a unique identifier - its name, which it gets assigned after being added to the container.

### 3.3.3. Designers

Designers are classes that are associated with a component type via the *DesignerAttribute* and implement the *IDesigner* interface. Designers are initialized with an instance of a component, which they manage.

The root component, such as a Windows Forms form or an ASP.NET web page, has to implement the *IRootDesigner* interface, which is special in that it has to provide the visual representation of the component - the View. It is very important to note that the *GetView* method (Illustration 8: *IDesigner* and *IRootDesigner*, Page 31) is explicitly designed not to return a specific type (such as a Windows Forms "Control" or an ASP.NET web page), but a generic "Object", which can be casted to the appropriate type by the designer tool. The framework implements two Windows Forms root designers - for forms and for custom controls (*UserControl*).

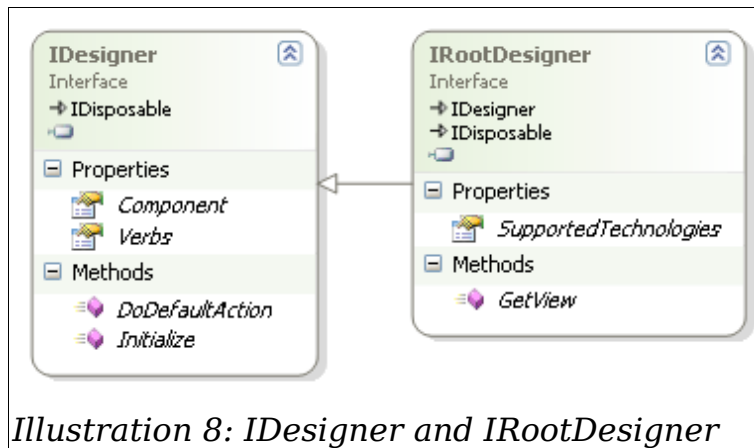


Illustration 8: *IDesigner* and *IRootDesigner*

The designers are responsible for:

- Performing custom initialization for a component in design mode.
- Alter and extend the behaviour or appearance of components in design mode.
- Adjust the attributes, events, and properties exposed by a component with which the designer is associated through the powerful type description functionality of .NET.

- Add menu items to the shortcut menu of a component.

### 3.3.4. Design-Time Services

The design surface provides a service container to the rest of the components of the design-time framework and populates it with a default implementation of several important design-time services, including the following.

<b>Service</b>	<b>Description</b>
<i>INameCreationService</i>	Generates or validates names for a components.
<i>ISelectionService</i>	Defines an interface for programmatic component selection and selection tracking, but not a visual one.
<i>IMenuCommandService</i>	Allows designers to add actions to the right click menu of a component during design time.
<i>IServiceContainer</i>	The design-time service container, where other parts of the framework can add their provided services.
<i>IComponentChangeService</i>	Notifies for when components are added/removed from the design surface and also when they get modified.

### 3.4. DESIGN-TIME SERIALIZATION

Design-Time serialization is the process of converting an object graph into a source file (code, markup or other format) that can later be used to recover the object graph. It is based on reflection and type transformation.

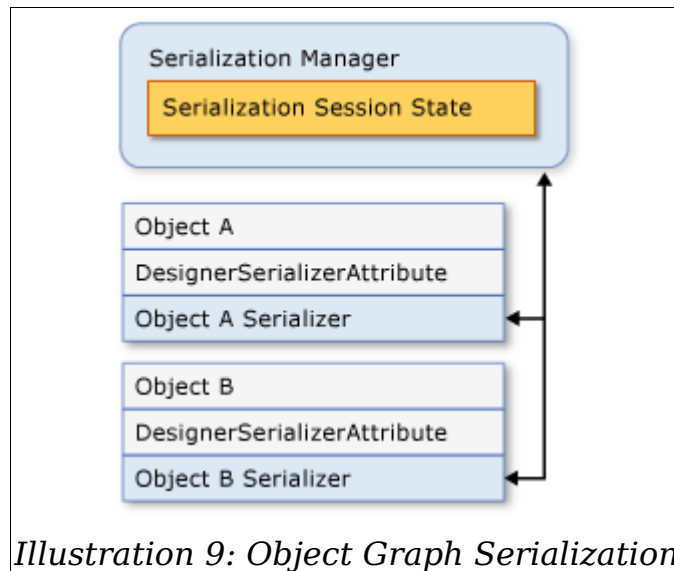


Illustration 9: Object Graph Serialization

The Design-Time serialization differs from the standard object serialization in the following ways:

- It separates the object that is performing the serialization from the one that is being serialized. Again, the keyword here is loose coupling.
- It serializes only properties that have been modified in order to minimize the output as well as provide only meaningful such.
- It ignores objects that aren't convertible to instance descriptors instead of throwing exceptions and interrupting the serialization process.

The design aims at providing a serialization system that is:

- Modular - Types are loosely associated with serializers via the *DesignerSerializationAttribute*.
- Format neutral - Each serializer handles the serialization for a specific type and it's up to the serializer to decide in what format the data is to be stored.
- Extensible - The system is designed to give priority to objects defined

as serializer providers before checking for *DesignerSerializationAttribute*. Those object feed the serialization system with serializers based on an internally stored type-serializer table. This allows default serialization associations via attributes to be overridden or association of serializers with types that lack a *DesignerSerializationAttribute*.

- Context Sensitive - The serialization process is managed by a serialization manager an instance of which is passed to all serializers to allow them to communicate.

### 3.5. THE DESIGNER TOOL

Based on the proposed design the designer tool can be said to be just a thin front end to the whole Design-Time framework (Illustration 10: A designer tool hosting a design surface, Page 35), with only a few responsibilities:

1. Instantiate the design surface with an arbitrary root component type and host the design surface view.
2. Provide a two way “bridge” between the Design-Time serialization system object graph and the data persistence format of choice.
3. Provide means for editing the metadata of the components in the surface.
4. Provide a way to add components to to the surface.

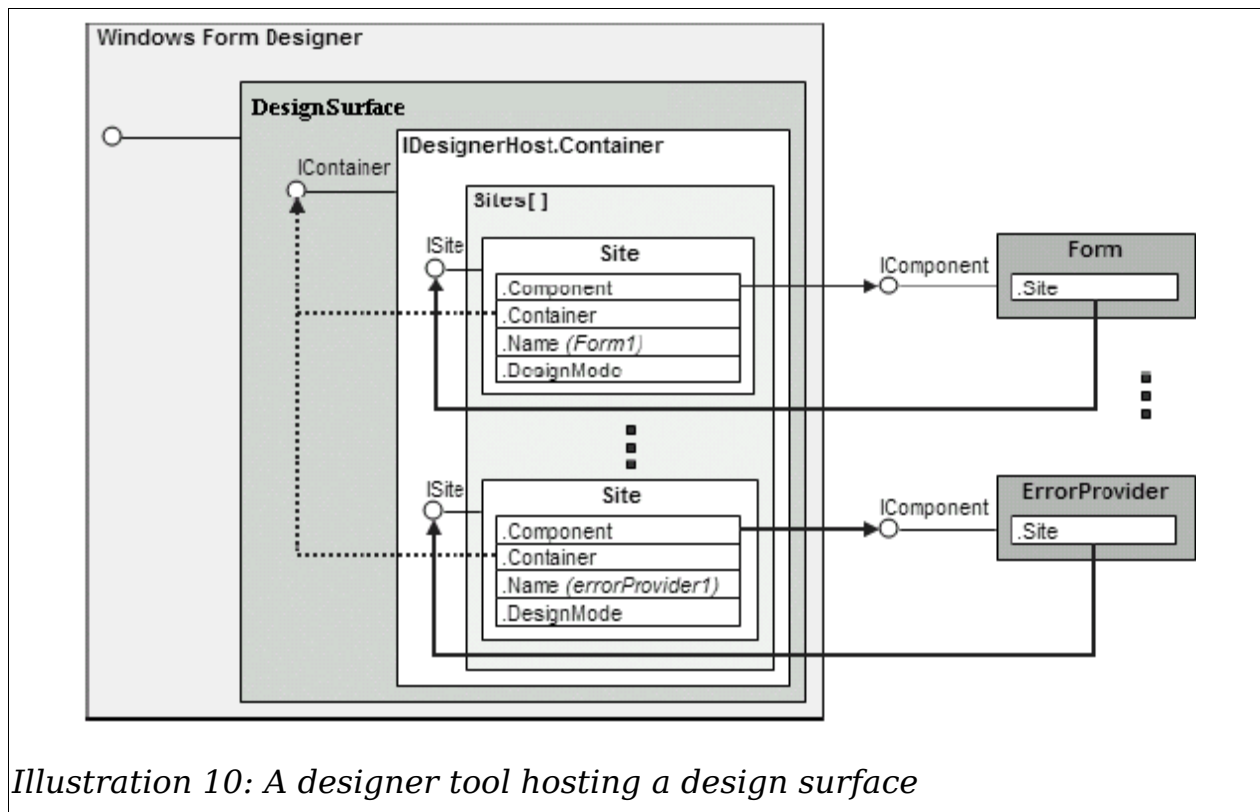
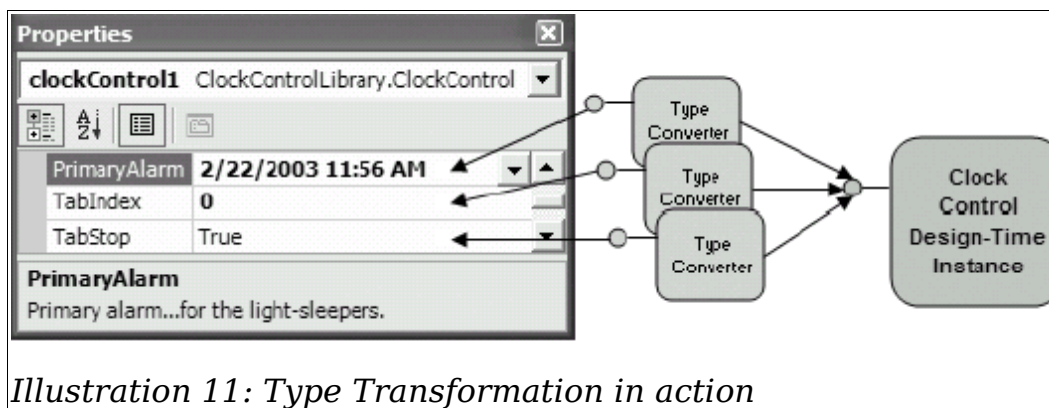


Illustration 10: A designer tool hosting a design surface

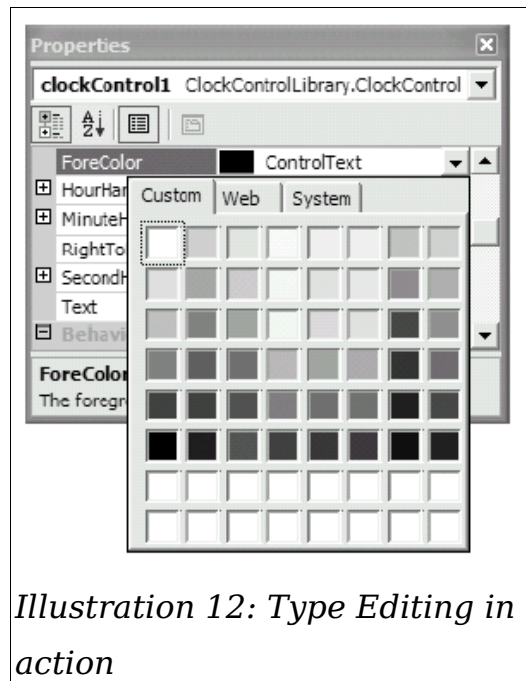
### 3.5.1. Metadata Editing

With the proposed design the designer tool has access to the *ISelectionService*, offered by the design surface, which provides information about the currently selected component as well as a mechanism for notification when the selection changes.

The designer tool will make use of that service to monitor the surface and populate a property grid with the currently selected object's properties using reflection. It will use type transformation (Illustration 11: Type Transformation in action, Page 36) and type editing (Illustration 12: Type Editing in action, Page 36) to provide user friendly facilities for metadata customization.



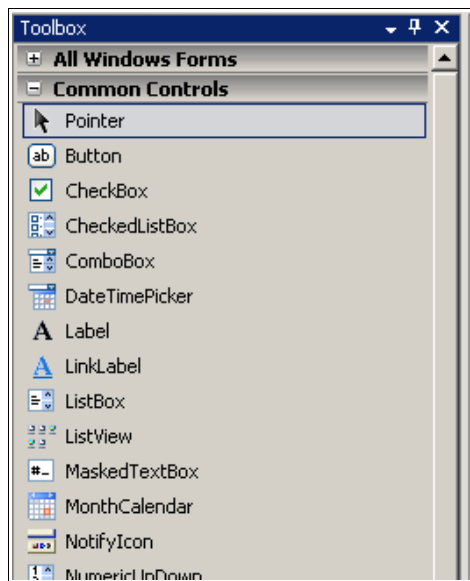
*Illustration 11: Type Transformation in action*



*Illustration 12: Type Editing in action*

### 3.5.2. Component Toolbox

Based on the powerful reflection functionality of .NET the designer tool can populate a component toolbox for a particular GUI toolkit by filtering all types that derive from the toolkit's foundation control type (e.g. "Control" for Windows Forms). The user will be able to drag and drop a component toolbox item onto the design surface which will be handled by the designer for the control under cursor to request the instantiation of the component.



*Illustration 13: A toolbox*

## 4. DEVELOPMENT PLAN

### 4.1. GOALS

This project will design, develop and test an abstracted framework for creating user interface design tools for the .NET platform in C#. It will act as an universal back-end which will allow the transparent hosting of an arbitrary front-end - the design tool. The different design tools will be able to target a particular graphical user interface toolkit, including desktop and web based such.

In addition it will implement a proof of concept and expected to be very fragile Windows Forms designer tool that will be capable of the following:

- Load a sample C# Windows Forms form generated by Visual Studio 2005 and supplied as part of the source code for this project.
  - Add simple controls, such as buttons and panels from the toolbox and modify their properties.
  - Move and resize those controls.
  - Undo/Redo/Cut/Copy/Paste those controls.
  - Persist the form modifications back to compilable source code.
- Pass the automated testing harness.
- Run both on Microsoft .NET on Windows and Mono on Linux.

### 4.2. MILESTONES

The milestones are specific to the implementation and do not cover the time required for research.

- Setup the project infrastructure
- Develop the Design Surface component hosting code with the default design-time services.
- Design and implement an automated test harness
- Implement the Windows Forms Designer stack

- Implement a proof of concept Windows Forms designer front-end
- Implement the Design-Time serialization system
- Implement CodeDom serialization for the Design-Time serialization system and Windows Forms
- Implement persistence support for the Windows Forms designer front-end
- Testing

**4.3. TIME PLAN**

	<b>Week 1</b>	<b>Week 5</b>	<b>Week 12</b>	<b>Week 18</b>	<b>Week 19</b>	<b>Week 20</b>	<b>Week 22</b>	<b>Week 23</b>	<b>Week 24</b>	<b>Week 27</b>	<b>Week 30</b>
	1 Oct	29 Oct	17 Dec	28 Jan	4 Feb	11 Feb	25 Feb	3 Mar	10 Mar	31 Mar	21 Apr
Initial Report	█										
Intermediate Report		█									
Final Report	█										
Research	█										
Design Surface			█								
Design Surface Test Harness					█						
Windows Forms Designers						█					
Windows Forms Designer Tool								█			
Design-Time serialization system									█		
CodeDom Serialization									█		
Windows Forms Designer Tool state persistence support										█	
Testing	█										

## 4.4. TESTING

Two levels of testing were applied throughout the duration of this project to ensure the software quality and to detect the introduction of regressions in the source code during the development process.

An automated unit based test harness was designed using the NUnit testing framework to simulate a design tool hosting a design surface in order to allow testing of the design-time service functionality, communication and interaction.

In addition, because interaction based facilities such as dragging and dropping controls, modifying metadata in the property grid, performing undo/redo/cut/copy paste operations was very time consuming to automate with programmatic tests, it was decided that those features will be tested by hand.

### 4.4.1. Automated Test Results

This project is delivered with all automated tests passing as can be seen on Code Snippet 6: Test Results, Page 41.

```
NUnit version 2.2.0
Copyright (C) 2002-2003 James W. Newkirk, Michael C. Two, Alexei A. Vorontsov,
Charlie Poole.
Copyright (C) 2000-2003 Philip Craig.
All Rights Reserved.

OS Version: Unix 2.6.22.13    Mono Version: 2.0.50727.42

.....

Tests run: 28 (all pass), Not run: 0, Time: 4.16457 seconds

Tests run: 28, Failures: 0, Not run: 0, Time: 4.16457 seconds
```

*Code Snippet 6: Test Results*

#### 4.4.2. Interaction-Based Testing Results

During the interaction based testing results the following defined goals were achieved:

- Load a sample C# Windows Forms form generated by Visual Studio 2005 and supplied as part of the source code for this project.
  - Add simple controls, such as buttons and panels from the toolbox and modify their properties.
  - Move and resize those controls.
  - Undo/Redo/Cut/Copy/Paste those controls.
  - Persist the form modifications back to compilable source code.

The appropriate successful testing data can be found in Appendix 2: Expected Data, Page 54 and Appendix 3 - Result Data, 56.

## 4.5. CHALLENGES

All projects have their challenges and this section of the report will look at some of the challenges faced during this project.

### 4.5.1. Open Source Code Base

Working with an open source code base, such as the .NET class library implemented by the Mono Project was challenging by itself. On several occasions bugs were encountered which had to be either filed on the Mono's bug tracker or fixed. Most of the encountered bugs in Mono were fixed by the author of this report throughout the project.

### 4.5.2. Windows Forms Input Filtering

In order to implement the Windows Forms designers stack this project had to find a way to redirect the input of a Windows Form control to customize its behaviour at design time.

Overriding the virtual *WndProc* method, which accepts the messages was not an option as this required sub-classing each individual control and would have completely broken the whole idea of abstraction and loose binding applied to the Design-Time framework.

Reading about the workings of the Microsoft Windows native window message loop was required and finally a solution was found. The concept was to redirect the native windows messages, which the Windows Forms control receives, prior to it to be able to handle them. The redirection mechanism would forward the messages to the designer instead, which will be able to filter out only the input messages and pass all the others back to the control.

This was achieved with the implementation of a *WndProcRouter* class to replace the *Control.WindowTarget* of the control class and forward messages to the designer. The replacement happens during the initialization of the designer with the control.

### 4.5.3. Serialization

Implementing the Design-Time serialization system proved to be a very difficult task. It was not taken into account that in some cases during the serialization the context will be important for the current expression. It was assumed that all serialization will take the form of:

```
Control.Left = 5;  
Control.Top = 0;
```

*Code Snippet 7: The wrong assumption*

But it turned out that serialization of complex types requires more than one level of nested expressions such as:

```
Control.Bounds.X = 5;  
SplitContainer.Panel1.Left = 6;
```

*Code Snippet 8: Unexpected serialization case*

And because the initial serialization was not designed to handle that it resulted in invalid code such as:

```
Bounds.X = 5;  
Panel1.Left = 6;
```

*Code Snippet 9: Invalid serialization*

Fixing that required major refactoring to introduce an expression context tracking stack in the serialization manager to allow serializers to know that e.g. the “*Panel1.Left*” expression is a sub-expression of “*SplitContainer.*”

### 4.5.4. Designer Transactions

While the design surface was designed with component state notification in mind and specifically for that purpose a design-time service (*IComponentChangeService*) was provided to allow subscription to the state notification events for all components, it wasn't taken in account that there will be need for encapsulating multiple modifications in a single notification unit.

This became obvious when the implementation of Undo and Redo was approached. It became apparent that due to the fact that a control on the surface is being dragged interactively, for every mouse move during the move operation the location of the control will be set, which will raise a component state change notification. The problem is that for a control move operation there will be tens of those location changes and because the undo engine was tracking them in order to undo the move as a whole one had to undo each of those small changes.

In order to fix that the concept of transactions was introduced and provided as a service to all parts of the Design-Time framework. Now when a move operation starts a transaction will be opened and when it ends it will be closed. The undo engine will aggregate all state changes between the start and end of the transaction and will be able to undo the move operation as a single unit.

## 5. CONCLUSION

This final chapter of the report reflects on the project process and offers a critical evaluation of the work performed and ways in which the project could be extended in the future.

### 5.1. IMPLEMENTATION SUMMARY

#### 5.1.1. Approach

As with most projects the first task was to set about researching the field. Fortunately GTK+, Qt and .NET were very well documented. Along with .NET itself other areas and concepts were researched such as runtime environments, retrospection, code reusability and user interface design, a topic well covered in computer science.

Once the research was complete the design phase began. It soon became apparent that the concept of rich Design-Time behaviour covers huge areas and a lot of concepts. Once the initial design was chosen the implementation began.

In order to implement the project efficiently the components were worked on one by one in self contained chunks. Often during the implementation of a component the design had to be revisited then be reincorporated back into the component. Testing was performed throughout the whole duration of the project.

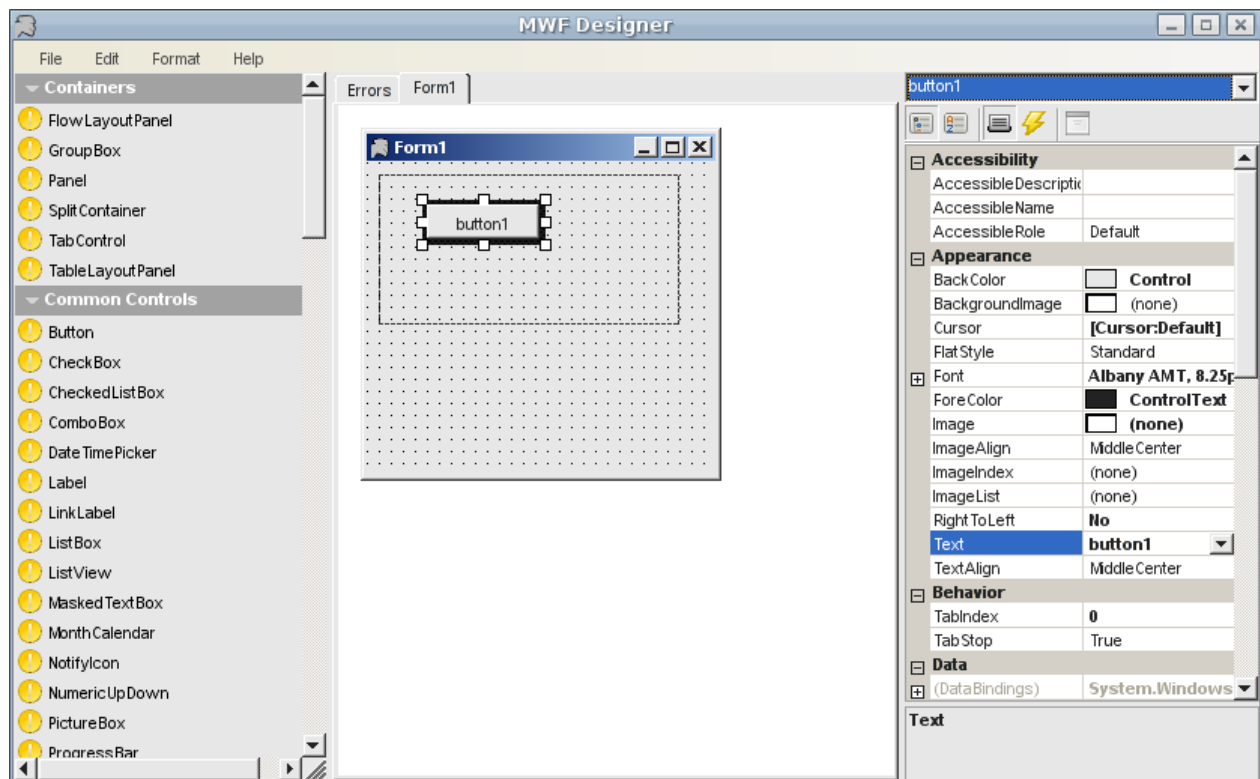
#### 5.1.2. Deliverables

This project successfully completes the defined in the development plan goals, by designing, developing and testing an abstracted framework for creating user interface design tools for the .NET platform, written in C# with a proof of concept front-end Windows Forms Designer, which can:

- Load a sample C# Windows Forms form generated by Visual Studio 2005 and supplied as part of the source code for this project.

- Add simple controls, such as buttons and panels from the toolbox and modify their properties.
- Move and resize those controls.
- Undo/Redo/Cut/Copy/Paste those controls.
- Persist the form modifications back to compilable source code.
- Pass the automated testing harness.
- Run both on Microsoft .NET on Windows and Mono on Linux (Illustration 14: The proof of concept Windows Forms designer, Page 47).

It should be noted however that as per the goals the proof of concept Windows Forms Designer is a very fragile piece of software due to the complexity of the underlying code.



*Illustration 14: The proof of concept Windows Forms designer*

## **5.2. CRITICAL EVALUATION**

Compared to contemporary projects the problems discussed in this report are of reasonable complexity and of high scale.

When other similar approaches to the same problematic are examined, such as the GTK+ and Qt ones, the elegance of the proposed design with its extensibility and abstraction cannot be denied.

The project started out with a clear concept of what is to be achieved, but possibly by underestimating the scale and complexity. While those did not have effect for the completion of the defined goals, they did provide insight into the requirements for future long-term developments, which will surely require the resources of more than a single developer.

## **5.3. FUTURE DEVELOPMENTS**

As this project is an investigation into the workings of a huge concept which covers a lot of things from runtimes/virtual machines and metadata retention and retrospection to user interface design technologies there are endless number of areas left for future work and development.

In the short term (3 to 6 months) though there are a couple of concrete improvements and new features that can be added to the existing code base:

- Improve the Windows Forms designers to aid better laying out controls with features such as automatic aligning, lining up and spacing.
- Improve the overall stability of the Windows Forms designer tool.
- Implement support for editing menus directly in the Windows Forms designer tool by the means of a Design-Time service.
- Research on new ways to easy graphical user interface design.

## **6. INTELLECTUAL PROPERTY RIGHTS**

All of the source code of this project is covered by the MIT license. The documentation, including this report, is covered by the Creative Commons Attribution-ShareAlike 3.0 license.

## 7. BIBLIOGRAPHY

The User Interface: Concepts & Design  
Lon Barfield

C Programmin Language  
Ernest C.ACKermann, Ph.D.

Teach Yourself C++, Fourth Edition  
Jesse Liberty, David B. Horvath

C# Essentials  
Ben Albahari, Peter Drayton & Brad Merrill

The Official GNOME 2 Developers Guide  
Matthias Warkus

Software Engineering 8  
Sommerville

Qt Reference Manual  
<http://doc.trolltech.com>

GTK+ Reference Manual  
<http://library.gnome.org/>

GObject Reference Manual  
<http://library.gnome.org>

Microsoft Developer Network Documentation  
<http://www.msdn.com>

Mono Project  
<http://mono-project.com>

SharpDevelop  
<http://www.icsharpcode.net/opensource/sd/>

MIT License  
<http://www.opensource.org/licenses/mit-license.php>

Attribution-ShareAlike 3.0 License  
<http://creativecommons.org/licenses/by-sa/3.0/legalcode>

## 8. APPENDIX 1 - GOBJECT TYPE

The following example source code demonstrates how a type is written in C using the GObject library.

```
#define SAMPLETYPE_TYPE (sampletype_get_type ())
#define SAMPLETYPE(obj) (G_TYPE_CHECK_INSTANCE_CAST ((obj), \
    SAMPLETYPE_TYPE, SampleType))
#define SAMPLETYPE_CLASS(klass) (G_TYPE_CHECK_CLASS_CAST ((klass), \
    SAMPLETYPE_TYPE, SampleTypeClass))
#define SAMPLETYPE_IS_BAR(obj) (G_TYPE_CHECK_INSTANCE_TYPE ((obj), \
    SAMPLETYPE_TYPE))
#define SAMPLETYPE_IS_CLASS(klass) (G_TYPE_CHECK_CLASS_TYPE ((klass), \
    SAMPLETYPE_TYPE))
#define SAMPLETYPE_GET_CLASS(obj) (G_TYPE_INSTANCE_GET_CLASS ((obj), \
    SAMPLETYPE_TYPE, SampleTypeClass))

typedef struct _SampleType SampleType;
typedef struct _SampleTypePrivate SampleTypePrivate;
typedef struct _SampleTypeClass SampleTypeClass;

struct _SampleType { /* instance members */
    GObject parentType;
    SampleTypePrivate *private;
};

struct _SampleTypePrivate { /* instance members */
    gchar *test_property;
};

struct _SampleTypeClass { /* class members */
    GObjectClass parent;
};

enum {
    SAMPLETYPE_TESTPROPERTY = 1,
};

GType sampletype_get_type (void)
{
    static GType type = 0;
    if (type == 0) {
        static const GTypeInfo info = {
            sizeof (SampleTypeClass),
            NULL, /* base_init */
            NULL, /* base_finalize */
            sampletype_class_init, /* class_init */
            NULL, /* class_finalize */
            NULL, /* class_data */
            sizeof (SampleType),
            0, /* n_preallocs */
            NULL /* instance_init */
        };
    }
}
```

```

    };
    type = g_type_register_static (G_TYPE_OBJECT, "SampleType", &info, 0);
}
return type;
}

static GObject *
sampletype_constructor (GType          type,
                       guint          n_construct_properties,
                       GObjectConstructParam *construct_properties)
{
    GObject *obj;

    {
        /* Invoke parent constructor. */
        SampleTypeClass *klass;
        GObjectClass *parent_class;
        klass = SAMPLETYPE_CLASS (g_type_class_peek (SAMPLETYPE_TYPE));
        parent_class = G_OBJECT_CLASS (g_type_class_peek_parent (klass));
        obj = parent_class->constructor (type,
                                       n_construct_properties,
                                       construct_properties);
    }

    return obj;
}

static void
sampletype_set_property (GObject      *object,
                        guint         property_id,
                        const GValue *value,
                        GParamSpec   *pspec)
{
    MamanBar *self = (MamanBar *) object;

    switch (property_id) {
    case SAMPLETYPE_TESTPROPERTY: {
        g_free (self->private->name);
        self->private->name = g_value_dup_string (value);
        g_print ("test property value: %s\n",self->private->test_property);
    }
    break;
    /* We don't have any other property... */
    G_OBJECT_WARN_INVALID_PROPERTY_ID(object,property_id,pspec);
    break;
    }
}

static void
sampletype_get_property (GObject      *object,
                        guint         property_id,
                        GValue        *value,
                        GParamSpec   *pspec)
{
    MamanBar *self = (MamanBar *) object;

    switch (property_id) {

```

```

case SAMPLETYPE_TESTPROPERTY: {
    g_value_set_string (value, self->private->test_property);
}
break;
default:
    G_OBJECT_WARN_INVALID_PROPERTY_ID(object,property_id,pspec);
    break;
}
}

static void
samplotype_class_init (gpointer g_class,
                      gpointer g_class_data)
{
    GObjectClass *gobject_class = G_OBJECT_CLASS (g_class);
    SampleTypeClass *klass = SAMPLETYPE_CLASS (g_class);

    gobject_class->constructor = samplotype_constructor;

    GParamSpec *pspec;

    gobject_class->set_property = samplotype_set_property;
    gobject_class->get_property = samplotype_get_property;

    pspec = g_param_spec_string ("test-property",
                                "test property nick",
                                "test property description",
                                "test property default value",
                                G_PARAM_CONSTRUCT_ONLY | G_PARAM_READWRITE);
    g_object_class_install_property (gobject_class,
                                    SAMPLETYPE_TESTPROPERTY,
                                    pspec);
}

```

## 9. APPENDIX 2: EXPECTED DATA

```
//
-----
// <autogenerated>
//     This code was generated by a tool.
//     Mono Runtime Version: 2.0.50727.42
//
//     Changes to this file may cause incorrect behavior and will be lost if
//     the code is regenerated.
// </autogenerated>
//
-----

namespace asd
{

    #region Windows Form Designer generated code
    public partial class Form1
    {
        private void InitializeComponent()
        {
            this.button2 = new System.Windows.Forms.Button();
            this.panel1 = new System.Windows.Forms.Panel();
            this.button1 = new System.Windows.Forms.Button();
            this.panel1.SuspendLayout();
            //
            // button2
            //
            this.button2.Name = "button2";
            this.button2.Location = new System.Drawing.Point(27, 30);
            this.button2.TabIndex = 1;
            this.button2.ImeMode = System.Windows.Forms.ImeMode.Disable;
            this.button2.Text = "button2";
            this.button2.UseVisualStyleBackColor = true;
            //
            // panel1
            //
            this.panel1.Name = "panel1";
            this.panel1.Location = new System.Drawing.Point(8, 8);
            this.panel1.TabIndex = 1;
            this.panel1.Controls.Add(this.button1);
            this.panel1.Text = "panel1";
            //
            // button1
            //
            this.button1.Name = "button1";
            this.button1.Location = new System.Drawing.Point(32, 20);
            this.button1.TabIndex = 0;
            this.button1.ImeMode = System.Windows.Forms.ImeMode.Disable;
            this.button1.Text = "button1";
            this.button1.UseVisualStyleBackColor = true;
            //
            // Form1
            //
        }
    }
}

```

```
        this.Name = "Form1";
        this.ClientSize = new System.Drawing.Size(240, 235);
        this.Location = new System.Drawing.Point(48, 16);
        this.Controls.Add(this.panel1);
        this.Text = "Form1";
        this.panel1.ResumeLayout(false);
    }
    private System.Windows.Forms.Button button2;
    private System.Windows.Forms.Panel panel1;
    private System.Windows.Forms.Button button1;
}
#endregion
}
```

## 10. APPENDIX 3 - RESULT DATA

```
//
-----
// <autogenerated>
//     This code was generated by a tool.
//     Mono Runtime Version: 2.0.50727.42
//
//     Changes to this file may cause incorrect behavior and will be lost if
//     the code is regenerated.
// </autogenerated>
//
-----

namespace asd
{

    #region Windows Form Designer generated code
    public partial class Form1
    {
        private void InitializeComponent()
        {
            this.button2 = new System.Windows.Forms.Button();
            this.panel1 = new System.Windows.Forms.Panel();
            this.button1 = new System.Windows.Forms.Button();
            this.panel1.SuspendLayout();
            //
            // button2
            //
            this.button2.Name = "button2";
            this.button2.Location = new System.Drawing.Point(27, 30);
            this.button2.TabIndex = 1;
            this.button2.ImeMode = System.Windows.Forms.ImeMode.Disable;
            this.button2.Text = "button2";
            this.button2.UseVisualStyleBackColor = true;
            //
            // panel1
            //
            this.panel1.Name = "panel1";
            this.panel1.Location = new System.Drawing.Point(8, 8);
            this.panel1.TabIndex = 1;
            this.panel1.Controls.Add(this.button1);
            this.panel1.Text = "panel1";
            //
            // button1
            //
            this.button1.Name = "button1";
            this.button1.Location = new System.Drawing.Point(32, 20);
            this.button1.TabIndex = 0;
            this.button1.ImeMode = System.Windows.Forms.ImeMode.Disable;
            this.button1.Text = "button1";
            this.button1.UseVisualStyleBackColor = true;
            //
            // Form1
            //

```

```
        this.Name = "Form1";
        this.ClientSize = new System.Drawing.Size(240, 235);
        this.Location = new System.Drawing.Point(48, 16);
        this.Controls.Add(this.panel1);
        this.Text = "Form1";
        this.panel1.ResumeLayout(false);
    }
    private System.Windows.Forms.Button button2;
    private System.Windows.Forms.Panel panel1;
    private System.Windows.Forms.Button button1;
}
#endregion
}
```