Views 2: Reflections on Views

by

Jonathan Eli Mason
B.Sc., University of Victoria, 2003

A Thesis Submitted in Partial Fulfillment of the Requirements
for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

Supervisor: Dr. R. Nigel Horspool

ABSTRACT

Views 2 is a toolkit for declaratively specifying graphical user interfaces (GUIs) with XML. Views 2 was created by the author based on prior work by Bishop and Horspool in *C# Concisely*. This thesis describes the features and design rationale of Views 2 and discusses how Views 2 improves over similar XML GUI toolkits. While other toolkits are mostly simple translations of current specifications of GUIs, Views 2 capitalizes on XML's flexibility and adds new strengths to allow software designers to express their designs in a maintainable and modifiable way.

Views 2 allows the XML to be split up, so that the specifications of components of the GUI can be separated into different logical concerns. Views 2 also supports Cascading Style Sheets (CSS) for formatting concerns. This separation of concerns is very flexible. It can be adapted to the designer's needs and to support good Software Engineering practice. Views 2 also offers customizable layout managers. Layout managers exist in other XML GUI toolkits, but in most cases there is only a small, fixed set available. Views 2 allows designers to create their own custom layout managers.

Views 2 makes heavy use of reflection in its implementation. Using reflection, Views 2 can instantiate any control available in the WinForms libraries and make use of all their properties and events. None of this information needs to be hard-coded into Views 2, as it is all available at runtime through reflection. Thus Views 2 has both completeness and flexibility which are not present in the original Views implementation

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

1.1.1 Views 2 for Graphical User Interface Design

Views 2 is a toolkit for declaratively specifying *graphical user interfaces* (GUIs) with *extensible markup language* (XML). XML based toolkits for GUI creation are a relatively recent, but popular, idea in GUI specification and design. XML is a very versatile and widely supported standard that is being used for the specification of many types of data. Microsoft, Mozilla and many others have created or are in the process of creating similar toolkits. Most of these differ only slightly from each other; the usual style is to have a tag named for a *control* to be created, and attribute-value pairs within the tag outline the parameters of the control. 'Control' is terminology from the Microsoft Windows Application Program Interface (API) for building GUIs. This API, known as WinForms, has a class named `Control` as the parent class of all basic GUI elements such as buttons, labels or text boxes. These types of GUI elements are also often known as widgets. Although Views 2 could be built using many underlying GUI APIs, the first (and currently only) implementation is built using the WinForms API, so Views 2's terminology follows WinForms. Therefore, the term control, not widget, is used for basic GUI elements.

An example of a button tag would be:

```
<Button Name="Button1" Text="OK" Width="75" />
```

This XML specification of the button has advantages over the specification of a button in non-XML toolkits (these alternatives will be examined in chapter 2). It is a very simple translation of the source code that would do the task in most of these toolkits. Views 2 tries to move a step beyond simply using an XML specification as a surrogate for the source code specification of controls, this serves only as the starting point for Views 2. Providing a set of tags in approximately the format shown in the example above will produce a GUI in Views 2 and most other XML based GUI toolkits. However, in most of these toolkits, this format would be the only way to input a GUI specification; Views 2 provides alternatives that are superior in the long term.

Views 2 embraces state of the art principles of good software design by encouraging modular design of GUIs. Specifically, Views 2 allows the XML input to be split into multiple sections, and properties for controls can be logically grouped together to separate the design concerns. A control can have its properties specified in more than one section of the XML; the entire list of properties need not be combined as attribute-value pairs in a single XML tag. When used systematically, this will improve overall readability and maintainability of the system. It also allows specific aspects of the GUI to be isolated and altered for different needs. The primary example of this is formatting information, for which Views 2 also supports a CSS-based style sheet system. Views 2 does not attempt to force any particular separation of concerns. The system is entirely flexible so that designers can separate their specification into whatever logical units are relevant to their situation. There is a small amount of overhead in using this style, but over the long term life of a software project the improvements in readability and maintainability are well worth the small overhead.

Rather than advocating any particular GUI design methodology, Views 2 focuses on flexibility. Likewise, this thesis will not explicitly outline how a GUI should be designed. Readers interested in that subject are instead referred to work by Bishop

and Horspool [1].

In keeping with the theme of encouraging flexible design, Views 2 offers layout managers. Layout managers are routines that, given a set of controls, will calculate their positions according to some rules. For example, a vertical layout manager would arrange a set of controls in a vertical list. Layout managers exist in some other GUI toolkits. Java's Swing offers numerous layout managers although it is not an XML based toolkit. XUL and the original Views offer some basic layout managers with vertical and horizontal arrangement of groups of controls and Views 2 provides this functionality as well. However, Views 2 also allows designers to add more layout managers by creating simple, lightweight layout manager classes. These layout managers can be registered with any instance of the Views 2 system and referenced by an associated name in the XML in a similar way to the layout managers that are built-in. These layout managers are easy to create and even easier to reuse, allowing designers to have controls laid out in any way they wish without having to manually specify coordinates.

Views 2 may not be the only XML based GUI toolkit available, but it offers a few key areas of major improvement over the existing state of the art.

1.1.2 Views 2 in Education

Views 2 has significant educational applications. The original Views system was designed as a companion to the introductory C# textbook *C# Concisely* [2] and as such was targeted towards novice programmers. Most of Views 2 is backwards compatible with Views, as the XML specification allowed in Views is a subset of that allowed in Views 2. Therefore, Views 2 offers novice programmers a route from Views to industrial strength application design. Because of the flexibility of Views 2 specifications, this change can be quite gradual. The only point of major incompatibility is the event model. Views has a simplified event model consisting of one method for the user to call. This method blocks until a control raises an event that is registered with Views,

and then returns the name of that control (this is called a wait loop event handler). Views 2 uses the more standard style. Users may register a method to handle any event supported by the underlying API and then, when that event is raised, control will be passed to the method they registered (this is called a callback event handler). Once a Views programmer understands this model, it is very easy to use Views 2 and they can then gradually progress to more and more complicated GUI designs.

All of the control and property names used in Views 2 specifications have the same names as the underlying classes and fields in the API being used (in the case of the current implementation this is the WinForms API from the .NET Framework). Therefore, whether they realize it or not, the programmers are learning about the underlying API as they build their applications. This, coupled with the gradual increase in the variety of features used, lead to Views 2 being an excellent educational tool as well as being a toolkit for industrial strength GUI design.

1.1.3 The Design of Views 2

In addition to encouraging good software design in the applications that use Views 2 specifications, Views 2 itself is an application of a new software design technique. The engine in Views 2, which creates objects from the XML, is an example of reflective middleware. Views 2 parses the XML input and extracts names of controls and properties; it then queries the underlying API for classes and fields of matching names. Views 2 then instantiates classes, assigns values to fields or creates event handlers as appropriate to the structures in the API and the names in the XML. This is all done at runtime through use of reflection, with no need to recompile any portion of the system.

No hard-coded lists of supported controls or properties needs to be included in Views 2. The list of supported controls is exactly the list of controls that exist in the underlying API at runtime, which is significant because new controls can be added to the underlying API without recompiling Views 2. If the user specifies a control in

his or her XML, Views 2 will discover the needed classes at runtime, whether they existed when Views 2 was compiled or not.

This new type of middleware is being researched in many areas. Reflection is not a new technique; it has been provided in programming languages for many years, but only recently has started to be used to enable a new generation of extremely flexible middleware. Views 2 encourages state of the art software design, but it is itself also designed with state of the art techniques.

1.2 Results

In chapters 5 and 6, an example of a small, but reasonably detailed, program — the colour picker — is specified in Views 2, Microsoft's XAML and Mozilla's XUL. The specification in Views 2 offers many improvements over these other options. In chapter 2, small examples of XAML and XUL are shown to demonstrate what the toolkits' specifications look like. The differences from these and equivalent Views 2 XML are minor. But the colour picker example, which is still only a single frame application, shows the strengths of Views.

In XAML and XUL, the tags for many of the controls take up multiple lines and this starts to reduce readability rapidly. Views 2 has the minor drawback that control names must be repeated in separate files, but this is a small bit of overhead at design time and the result is a specification that is much easier to read and understand. Controls are split into multiple tags in separate files and each tag has only a few properties being set at a time. It easy to understand what task each part of the specification is meant to perform. This will aid maintenance of programs drastically, because it not only makes it easier to understand the specification by breaking it down into easily readable parts, but it also lets designers separate design phases. When maintainers are updating the specification they can easily update the design in the same phases it was originally created in. For example, when they are deciding what

information a control is going to convey they can view only the XML that specifies the information content of the other controls; they do not need to be reading over formatting information or event handlers for other controls until they come to these phases of the design for the new controls.

Not only does this separation allow design to be easily revisited in stages, but it also allows sections to be isolated and altered for different platforms while the rest is reused. For example, formatting information could be changed to provide a large format version of the GUI for the visually impaired, while the rest of the specification remains unchanged. Views 2 also supports *cascading style sheets* (CSS), which for the particular concern of formatting information can greatly reduce repetition of properties by making use of CSS classes and applying properties to all controls of a particular type or CSS class.

Also the presence (or absence) of layout managers is immediately obvious in the specification. XUL supports only simple horizontal and vertical list layouts (`<hbox>` and `<vbox>` tags), but that is all that is required by this application. XAML, however, has to specify manually determined coordinates for every control. Not only is this much more difficult to read than the XUL or Views 2 version with layout managers, but if another control were to be added in the middle of the form, the locations of many other controls would need to be recalculated. With layout managers this is not the case, simply adding the control as a child of the correct layout manager group will cause it to be placed appropriately and all others to be moved to make room for it with no manual recalculation. Layout managers save time both in the initial design and in any modifications of the program. XUL and Views 2 both support simple horizontal and vertical list layout managers, but XUL supports *only* those two. Views 2 allows designers to create their own, so that controls in their application may be laid out in any manner they see fit. It is very easy to create new layout managers and the usage of them in the XML specification is the same as with the built-in Views 2 layout managers. The designers simply supply Views 2 with the string they wish to use to

refer to their layout manager and this string can be included in the XML just as any of the built-in layout managers.

These ideas, and the comparisons and examples in chapter 5, show that Views 2 implements some very useful and powerful ideas in GUI design. Separation of XML to match separation of design concerns, CSS for formatting information and layout managers all add up to Views 2 being an extremely powerful way for designers to specify programs that are comparable in capabilities to any modern GUI design toolkit, but also provide a much better underlying design model in accordance with good practice. Combine this with the educational benefits of having the original Views system as a basis (refer back to section 1.1.2) and the result is an extremely flexible system that has much to offer everyone from novice programmers to expert professional software designers.

1.3 Chapter Outline

This thesis is organized as follows:

- Chapter 2 gives the background information for the subject. It covers middleware and reflection and it places Views in the context of the two. It also covers a variety of other GUI creation toolkits, including an outline of the original Views system.
- Chapter 3 discusses design goals for the Views 2 system, particularly where improvements to the current state of XML GUI toolkits can be made. It also discusses why Views serves as a solid basis for an industrial strength GUI design toolkit and introduces the improvements Views 2 makes over Views.
- Chapter 4 discusses the design of Views 2. It begins with an overview of the approach Views 2 takes to turn the XML specification into GUI objects. Then it moves on to a discussion of the rationale of the design decisions. The final section of this chapter discusses how reflection is beneficial as a tool for GUI

design.

- Chapter 5 gives a detailed example of the specification of a reasonably complex GUI. This shows how Views implements the ideas from 3 and 4.
- Chapter 6 evaluates Views 2 by comparing it to two major XML GUI toolkits. The example program from chapter 5 is reworked in both Microsoft XAML and Mozilla XUL to evaluate how Views 2 improves over these toolkits.
- Chapter 7 discusses future work and conclusions. It outlines some ideas for new projects related to the body of work represented by Views 2, both as a GUI toolkit and as reflective middleware. Then it closes the thesis with a summary and concluding thoughts on the insight gained through this work.

Chapter 2

Background

2.1 Pieces of Views

The Views 2 system is built on ideas drawn from several related projects, as well as being based on several state of the art technologies and techniques. Views 2 is in many ways a middleware application. When most people think of middleware they think of systems like CORBA [3]. While Views 2 does not serve a purpose similar to CORBA, it is in some ways middleware. Looking at Views 2 from a middleware perspective will help give some insight into its usefulness and functionality; so this chapter provides some information on middleware. Views 2 also makes extensive use of reflection [4], which is a programming technique that has been available in high level languages for many years, but is not widely utilized, or even widely understood. Therefore, this chapter will also consider reflection and in particular how it aids middleware in its primary purpose.

There are other software systems and projects that Views 2 is related to or draws ideas from. First is the *Common Language Infrastructure* (CLI) itself and its various implementations. The CLI is the execution environment for *Microsoft Intermediate Language* (MSIL) bytecode, which is the output of most C# compilers. The Win32 .NET Framework that Microsoft distributes for use with their own operating systems is the primary implementation of the CLI. However, in addition, Microsoft has submitted the CLI as an ECMA Standard [5] and created the Rotor system, which is

their own cross-platform open source CLI implementation. Rotor provides a compiler and runtime environment for C# on Linux, BSD and Mac OS X. Rotor has one significant drawback in that it does not implement the System.Windows.Forms namespace which is required to create (and execute) graphical programs (this section of the .NET Framework is not part of the CLI as outlined by the ECMA Standard mentioned above). To create graphical programs with Rotor, a third party GUI toolkit must be used. In addition to Rotor there is Mono [6], which is a third party cross-platform open source implementation of the CLI. Unlike Microsoft's Rotor, Mono includes an implementation of System.Windows.Forms (WinForms). It does not, however, have support from Microsoft, and as a result may have areas where it does not operate exactly like the Windows version. Views 2 does not necessarily depend upon C# and WinForms, but the only current implementation of the Views 2 system does. In principle it could work on Mono (though it has not been tested), but to work under Rotor will require some additional programming — which was done with Views 1, but has yet to be done for Views 2.

There are many other toolkits available to create GUIs and this chapter will investigate a few of the major ones. Since Views 2 is the second generation of the Views project it makes sense to take a brief look at the Views 1 system. In addition, Mozilla XUL, which is the largest XML based GUI creation toolkit currently in use, will be examined. Perhaps of greatest interest is Microsoft's XAML. Microsoft is in the process of creating a GUI toolkit based on XML for the next generation Windows platform. Because the current implementation of Views 2 is based on the .NET Framework using WinForms, there are quite a few similarities between Views 2 and XAML, as well as some significant differences. These comparisons will be visited again in Section 3.4.2, but some background information on XAML is provided towards the end of this chapter.

2.2 Introduction to Middleware

2.2.1 What is Middleware?

An accepted definition for middleware does not seem to exist. However, Vinoski [7] defined middleware as follows:

> Middleware is integration software.

This definition not only covers systems like CORBA, which is generally accepted as being an example of middleware, but also many other applications. According to this definition, middleware may be used to link two computers over a network or to combine two pieces of software not originally designed to work together.

Most middleware research is in the area of networking. That is because networks of sufficient size typically contain a heterogeneous mix of computing platforms. Middleware may be used as an intermediary between these platforms to enable reliable, effective communication across the network.

In effect, middleware adds a layer of indirection to the integration software which allows a more modular, reliable design.

2.2.2 Middleware and GUI Design

If looking at middleware as integration software, many modern GUI design systems are in fact a form of middleware. This thesis looks at several modern XML based GUI design tools. The engines that generate objects from XML are a kind of middleware. The engine integrates XML specifications and, in most cases, other user programs together with some underlying system structures to actually display the GUI to the user. This is middleware. The user is now no longer dependent on a particular underlying system. Because of the integrating power of middleware, the user's programs and XML can be integrated to use any system available. If the XML layer is designed well this can free the programmer from needing to be concerned with specific imple-

mentation details. This gives us the opportunity for excellent design abstractions as well as as strong separations of concerns.

Not only is this approach to middleware sound Software Engineering, giving us good abstractions for GUI design, but if we think of the engines themselves more abstractly as middleware systems, there is no reason the same principles cannot be expanded to other problem domains. The XML based GUI design systems generally use XML to specify the initial state, as it is a static definition. Then some kind of program associated with the XML specification operates on it possibly changing the structure and performing appropriate operations. This could apply to many systems, since many applications work on this general principle. The programmer sets up some initial state and then operates on it. There is no reason that the same abstraction could not be applied to systems other than GUI design.

2.3 Reflection Applied to Middleware

2.3.1 What is Reflection?

Reflection is meta-computation about a running program [4]; i.e. a program uses reflection to perform computation on itself. Non-reflective programs can only invoke methods or change variables that exist statically (at compile time). Programs using reflection in programming languages with the simplest level of reflection can gather information about classes at runtime and invoke methods or change variables that are discovered dynamically (at run time). This simple level of reflection is referred to as *introspection*. Programming languages that support more advanced forms of reflection allow changes to the structure of programs — or even the behavior of certain language features — at run time. Powerful forms of reflection can be divided into two types: *structural reflection* and *computational* (also known as *behavioral*) *reflection* [8].

The most basic form of reflection, introspection, allows a program to examine

itself. In a programming language supporting introspection, a running reflective program can inspect the interface of classes through the reflection system; usually by performing queries with strings that represent names of classes, methods or fields (member variables of a class). The reflection system will respond to these queries with meta-data objects that can be used for a variety of information gathering purposes and usually these meta-data objects can be used to invoke methods or change field values. Introspection also allows a program to make use of classes that were not available at compile time. Since the interface to the reflection system is parameterized (usually by string names), the name of a class or method that did not exist at compile time may be specified and then used through the meta-data. Introspection allows changes to the state of the system, by invoking methods discovered through reflection, but does not allow any changes to the system itself.

Structural reflection is a technique that allows changes to the structure (classes, methods, fields, etc.) of a running reflective program. A programming language that supports structural reflection has meta-classes that specify the structure of a class and allows classes themselves as first class objects. All objects, including classes, are instances of a class. A regular object is an instance of a class, which specifies its structure (this is no different from non-reflective object-oriented languages). In structural reflection terms, a class is an object's meta-data. This relationship also exists for classes (it must because classes are a type of object); classes are instances of meta-classes, which are the classes' meta-data and specify the structure of the class. This allows information gathering on the structure of a class, like introspection. However, languages that fully support structural reflection also allow changes to this structural meta-data. For example, they may add a field or method that did not exist at compile time.

Computational (or behavioral) reflection is a technique that allows changes to the behavior of running programs. A programming language that supports computational reflection has specific meta-objects for each instance of an object. In structural re-

flection the meta-object (i.e. meta-data) of an object is its class. This is not the case in computational reflection; classes and meta-objects are different. Two objects that are instances of the same class will not share the same meta-object, as every object has its own unique meta-object. A meta-object is responsible for an object's behavior and explicitly represents information that would be implicit in a non-reflective language. Consider, for example, message handling behavior. The meta-object may be undefined for a particular object, as it is created only when requested. But if a meta-object for a particular object exists and that object is passed a message, the object does not handle the message. Instead the meta-object is told to handle that message on its object. This allows a running reflective program to change its own behavior, since the meta-object might take a different action for a particular message from the action than the object itself would take under default language semantics. Thus a running program, in a system supporting computational reflection, may change the semantics of the language and even introduce new language features that did not exist at compile time (such as multiple inheritance or debugging tools) [9].

2.3.2 Reflection as an Integration Tool

When middleware is used as integration software, reflection becomes a powerful tool for building middleware. Reflection allows a runtime system to interrogate and manipulate software systems dynamically. This allows middleware a great deal of flexibility in terms of integration if it can dynamically gather information about the systems it is attempting to integrate. It is even better if the reflection system in use supports the higher levels of reflection that allow manipulation of the code base at runtime.

2.4 Overview of the Views System

2.4.1 Introduction

The Views system was originally created by Horspool and Bishop and is published in their introductory C# textbook, *C# Concisely* [2]. Views is a relatively simple toolkit for creating graphical user interfaces, but at the same time is an interesting introduction into using XML for GUI specification. In the Views system there are supported XML tags for a variety of GUI controls: Button, Label, TextBox, etc. and each of these controls has a variety of supported attributes: Name, Width, Text, etc. The tag names map in an obvious way onto subclasses of System.Windows.Forms.Control. The XML attributes map in an almost equally obvious way onto C# properties.

Views is not limited to the Windows platform, the first version was built up around the Windows Forms framework, but the concept of it can easily be applied to other platforms as well. In fact, originally this was one of the main purposes of Views. Though tag names are based on WinForms controls, there is an implementation of Views for Rotor that allows the same XML specification to work for both the Windows .NET Framework and the Rotor CLI with Qt plugged in for GUI rendering [10].

In principle, Views is not limited to C# either, as it is a versatile XML based specification. The specification could be readily used by almost any language [11].

2.4.2 Specification

The C# model for creating GUI components is a technique that may be characterized as "Construct and Set". C# GUI components are highly mutable. When a control is constructed, many of its parameters are unspecified. Instead, it has a set of default values for these parameters and the control may be mutated via its properties in order to provide the parameters with the necessary values. Visual Studio .NET does this automatically, it generates a block of code for each control that the user is told,

```
Form f = new Form();
TextBox tb_Entry = new TextBox();
Button b_OK = new Button();

f.Text = "Test 1";

tb_Entry.Text = "";
tb_Entry.Width = 150;

b_OK.Text = "OK"
b_OK.Width = 150;

f.Controls.Add(tb_Entry);
f.Controls.Add(b_OK);

tb_Entry.Top = 20;
tb_Entry.Left = 10;
b_OK.Top = 50;
b_OK.Left = 10;
```

Figure 2.1. *"Construct and Set" specification of a simple GUI*

with in-code comments, not to modify. However, this block of code is just exactly the "Construct and Set" code. Figure 2.1 shows a simple example of what this "Construct and Set" block would look like for a a simple form with a title, a text box and an OK button.

Views uses this style of constructing controls to its advantage. The XML can easily be parsed and there is no problem with determining which order to pass parameters to constructors and so on. Each tag will have one associated control that can be instantiated and each attribute will have one associated property that can be set.

In the original Views, the supported XML tags are hard-coded. The names of supported tags usually correspond to names of subclasses of Control in the System.Windows.Forms namespace, but this just determines the tag name. Whether there is any actual dependence on that namespace depends on which implementation

```
<Form Text="Test 1">
 <vertical>
  <TextBox Name="tb_Entry" Text="" Width="150"/>
  <Button Name="b_OK" Text="OK" Width="150"/>
 </vertical>
</Form>
```

Figure 2.2. *Views specification of a simple GUI*

of Views is being used, since the XML itself is platform-independent.

A Views XML tag has many attributes that define various aspects of the control. For example, an attribute called Text will define what text is displayed on the control. These are specified as attribute-value pairs and most controls are singleton tags. Certain controls such as Panel, GroupBox and the document element Form are allowed to contain nested Controls. These tags are not singleton tags and they have other controls as XML children. Certain other controls such as DropDownList have a list of contained elements. In Views, these controls are also not singleton tags and the elements are specified with a set of child Item tags, which must be strings.

Figure 2.2 shows a simple example of Views XML to make a simple form with a title, a text box and an OK button. Also compare this to Figure 2.1 and notice the many improvements the XML based format offers. It completely eliminates the need to add controls explicitly and the vertical layout tags eliminate the need to specify positions for the controls.

2.4.3 Event Model

The event model in Views is fairly simple. Every control (at least those that can raise events) must have a name attribute in the XML. The event model is a simple wait loop system, the user calls the GetControl() method of the Views.Form class. This method is a blocking call and when an event is raised returns a string that is the name of the control that caused the event. See figure 2.3 for a simple example of how

```
Views.Form f = new Views.Form( /* XML spec string here */ );
string cName = f.GetControl();
switch(cName)
{
...
  case "b_OK": // event handler code for the OK button
...
}
```

Figure 2.3. *A short piece of C# code to handle an event from Views*

this might work for the button 'b_OK' listed above. This event model is simple and does not offer the flexibility advanced programmers might want. It is, however, very good for novice programmers, as the events that are registered with the wait method are the most likely set of events a programmer would wants with each control. It is also a very flexible method as a variety of underlying event models can be used to implement this simple wait loop style.

2.5 Overview of Mozilla XUL

2.5.1 Introduction

XUL is created by Mozilla as an XML based cross platform GUI creation tool [12]. XUL is rendered by both the Netscape and Mozilla browsers, using the XPToolkit portion of the Mozilla Layout Engine [13]. Netscape often calls this 'Gecko', which can for most purposes be considered a synonym for the Mozilla Layout Engine. However, Mozilla.org refers to the engine as nglayout, so that is the name that is used in this thesis. XUL applications can be run through the engine present in any installation of these browsers. XUL borrows from HTML, but does many things HTML cannot. Because it borrows from HTML and is rendered by the same nglayout engine that renders HTML in the compatible browsers, there are many things that can be done equally easily. For example, *cascading style sheets* (CSS) can be applied to XUL

as easily as they can to HTML. XUL is used primarily in the development of the browsers and extensions to them, but it allows creation of GUI programs that do not having anything directly to do with the browser aside from being rendered by it. XUL programs running on the local machine are accessed with a special kind of URL — called a chrome URL — and have enhanced privileges that allow them to perform many functions needed for general purpose application, but because these chrome URLs cannot access XUL applications remotely, it does not offer a large security hole that might allow a remote application to perform tasks that it should not [14].

2.5.2 Specification

XUL has a simple model for specifying controls. A tag is named for the control type the programmer wishes to create with attribute-value pairs describing the parameters, such as text and colour and so on. Most of the expected parameters of a control can be adjusted with XML attributes. The specification is similar to that of an HTML form, but the top level tag is window. The concept of `id` and `class` are the same as in HTML; `id` is for event handlers and both are used for CSS style rules. But XUL supports a much wider range of control types and each control has its own tag rather than HTML's style where all controls are input tags with different type attributes. Figure 2.4 shows an example of an XUL file to make a simple form with a title, a text box and an OK button (very similar to the Views example shown previously)[1].

2.5.3 Event Model

XUL requires the use of JavaScript to handle events. The program loads a JavaScript with a script tag that points to a file containing the script and then simply adds attribute-value pairs to the controls. The attribute is the name of an event and the value is a call to the function that handles it. These values are in fact small

[1]The text in figure 2.4 copied into a .xul file and opened with an appropriate browser comprises a complete XUL program, albeit one without events.

```
<?xml version="1.0"?>
<?xml-stylesheet href="chrome://global/skin/" type="text/css"?>

<window
    title="Test 1"
    orient="horizontal"
    xmlns=
    "http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul">

  <vbox>
   <textbox id="tb_Entry" value="" Width="150"/>
   <button id="b_OK" label="OK" width="150"/>
  </vbox>

</window>
```

Figure 2.4. *XUL specification of a simple GUI*

JavaScript scripts and if the event handler is small enough it does not necessarily need to reference any outside program, but the recommended and usual style is to call a JavaScript function that exists in a referenced script file. The event model in XUL uses an interesting cascading function (similar to HTML) to find an event handler. To determine which element raised the event, the event is propagated down the document element hierarchy until the element is found and then sent back up the hierarchy until an event handler for that event is found. Therefore, an event handler does not need to be attached to the element that raised the event for it to handle the event; it could be attached to one of the element's parents in the element hierarchy. In fact, multiple event handlers could work on one event because of this. The reader is referred to XUL Planet's XUL Tutorial [12] for the details of XUL event handling semantics[2]. This method of locating an event handler could be used either to take some functionality based on the event at multiple points or to have a pass through to decide if a particular function should handle this event or not. Aside from this

[2]The reader may also wish to refer to the XUL specification [15] but at the time of this writing that document is a work in progress.

cascading technique of locating which event handler function will do the handling, XUL's event model uses a fairly straight forward callback event handler functions style.

Adding events is easy. A `<script>` tag is used to attach Java Script source to the XUL (either inline or by an external reference) and then the designer can simply update a tag with the appropriate event handler attribute-value pair. For example, consider the b_OK button in figure 2.4. The user might define a function `b_OK_command()` in his Java Script section and then link it to the command event by modifying the Button tag as follows:

```
<button id="b_OK" label="OK" width="150"
    oncommand="b_OK_command();" />
```

2.6 Overview of Microsoft XAML

2.6.1 Introduction

XAML is Microsoft's next generation GUI design technology. Like XUL it is an XML based specification of a GUI and links to code for event behavior. XAML is being developed by Microsoft as the premier GUI design tool for the next generation of Windows. XAML will integrate with Visual Studio .NET so that the designer can design GUIs in the same way that is done presently, but in principle instead of generating hundreds of lines of complicated C# code, which Visual Studio .NET flags with *do not modify* comments, a XAML file will be generated. XAML is based on the existing System.Windows.Forms namespace and, as a base, starts with the same functionality it does. The XAML engine from Microsoft — code-named *Avalon* — has not been released yet, as it is part of the next generation Windows platform, but there is a third party XAML engine called Xamlon available now running on current generation Windows. Since most of the current work in XAML is done with Xamlon

```
<?xml version="1.0"?>

<?Mapping XmlNamespace="wf" ClrNamespace="System.Windows.Forms"
    Assembly="System.Windows.Forms"?>

<wf:Form Name="Form1" Text="Test 1"
    xmlns="http://schemas.microsoft.com/2003/xaml/"
    xmlns:def="Definition" xmlns:wf="wf">
  <wf:Form.Controls>
    <wf:TextBox Name="tb_Entry" Text="" Width="150" Location="5, 5"/>
    <wf:Button Name="b_OK" Text="OK" Width="150" Location="5, 30"/>
  </wf:Form.Controls>
</wf:Form>
```

Figure 2.5. *XAML specification of a simple GUI*

somewhat speculatively on what will be available in Avalon, some details of XAML are still subject to change [16].

2.6.2 Specification

XAML, Views and XUL share the same basic concept using tags that are named for controls and attribute-value pairs to specify parameters. The top level tag of an XAML document can be a number of things. A Canvas is the most basic application space but it lacks some functionality. A Canvas will often be a child of a Window tag, which has more functionality. Figure 2.5 is an example of XAML to make that same simple form with a title, a text box and an OK button. In this case, the XAML specifies current standard WinForms controls, so the top level container control is a Form.

XAML has an interesting model for assigning complicated property values in attribute-value pairs. Because XAML has a close relationship to classes in the .NET Framework there are many attributes whose value cannot be specified with a simple string. XML has the flexibility to specify values that are in themselves objects, but an XML node is not a legitimate value for an attribute-value pair and an arbitrary

object value will require an XML node for a full specification. XAML uses a technique called composite properties (also known as complex properties) to meet this requirement. Composite properties are defined in an x.y notation, similar to referencing inner classes or namespaces in C# code. For example Canvas.Resources is a composite property that will appear as a child of a Canvas. Composite properties are not directly interpreted as part of the XAML rendering process. Instead a composite property contains named XML fragments than can be referenced as attribute values by its siblings. If a Canvas defines a Canvas.Resources section then controls in that Canvas can use names from that Canvas.Resources section as attribute values. This allows the programmer to specify arbitrary XML nodes as values for a control, which in turn allows the use of XML's flexibility and capability to define any object as a value.

2.6.3 Event Model

XAML's event model is the expected standard. Events are raised and sent to target methods. Like XUL, XAML requires code to handle these events. In XAML this takes the form of a class, the designer then use a def:Class attribute to point to this class, but XAML also requires a `<?Mapping?>` tag to specify where it can find the assembly containing the class. The class then implements event handlers in exactly the same way that current C# programs do and the controls in the XAML file can then reference these event handlers by a string representing the method name. When the event is raised control passes to the appropriate method. The example from figure 2.5 has been expanded to include an event handler. The name of a method in the class Example26.Example26[3] can then be given as an event handler as shown on the Button tag. As noted before this is a speculative example of XAML based on Xamlon. These are slightly more complicated changes than were required in XUL, so figure 2.6

[3]The dot indicates a namespace, so in this case it is the class Example26 in a namespace that has the same name

```
<?xml version="1.0"?>

<?Mapping XmlNamespace="wf" ClrNamespace="System.Windows.Forms"
    Assembly="System.Windows.Forms"?>

<?Mapping XmlNamespace="Example26" ClrNamespace="Example26"
    Assembly="Example26"?>

<wf:Form Name="Form1" Text="Test 1"
    xmlns="http://schemas.microsoft.com/2003/xaml/"
    xmlns:def="Definition" xmlns:wf="wf"
    def:Class="Example26.Example26">
  <wf:Form.Controls>
    <wf:TextBox Name="tb_Entry" Text="" Width="150" Location="5, 5"/>
    <wf:Button Name="b_OK" Text="OK" Width="150" Location="5, 30"
      Click="b_OK_Clicked"/>
  </wf:Form.Controls>
</wf:Form>
```

Figure 2.6. *XAML specification of a simple GUI, with an event*

shows the entire XAML source after being updated to include an event. The method `b_OK_Clicked()` should be a public method with the appropriate signature for a C# event handler. The class containing this can be compiled with a regular C# compiler and the second `<?Mapping?>` tag points at the DLL generated.

2.6.4 Dynamic Document Changes

Because XAML is as fully functional as the existing System.Windows.Forms API, there must be a way to manipulate the GUI at runtime. In the current generation of the .NET Framework, the objects are all instantiated in the C# code and have properties set (perhaps in code generated automatically by Visual Studio .NET) to match the initial configuration laid out by the visual designer. Since the objects are instantiated by the designer's class, the designer has regular programmatic access to all objects directly as fields in the class. Should any dynamic changes be needed

properties can be altered or methods can be called directly using these fields. As with the event handlers, the XAML model is very similar to the existing model. XAML basically serves the purpose of setting up the initial state of the GUI, but any dynamic changes that need to be made must be done by regular C# code. However, in XAML the designer's class does not instantiate the objects, so they do not exist as fields in that class to be operated on directly.

The recommended technique for programmatic access to the created objects is to make use of the `IPageConnector` interface. `IPageConnector` is an interface in the XAML framework that defines a single method called `Connect()`. In the XAML a `<?Mapping?>` tag points to a class that the designer creates which implements `IPageConnector`. For every XAML tag that represents a control, `Connect()` will be called. `Connect()` has two parameters: a string and an object. When the XAML framework calls `Connect()` the string argument is the value from the optional ID attribute in the tag and the object argument is the object that was created for the tag. The objects passed to `Connect()` can be cached for later use in the C# code. These cached objects will be the same objects that would have been member fields in current generation .NET Framework programs and can be used in a similar way to make dynamic changes to the GUI.

2.7 Comparison of GUI creation toolkits

There are many XML based GUI creation toolkits. The concept of using a markup language to specify a GUI is not itself very new. This section will provide a comparison of several such toolkits that are available. Most of the relevant information is summarized in the table below, but a few notes on the toolkits will be provided in the section that follows the table. Finally there is a brief summary of the relevant points from the table.

2.7.1 Comparison Table

The following table outlines a brief comparison of XML based GUI toolkits. A description of the columns in the comparison follows the table itself.

Comparison of XML GUI Toolkits						
	Tags match control names	Layout managers	CSS support	Call back events	Platform independent	Visual designer
Views	Yes	Yes	No	No	Yes	Yes
XHTML	No	No	Yes	Yes	Yes	Yes
XUL	Yes	Yes	Yes	Yes	Yes	Yes
XAML	Yes	No	No	Yes	No	Yes
JDNC	No	No	No	Yes	Yes	No
GNU	Yes	No	No	Yes	Yes	Yes
Bindows	No	No	Yes	Yes	Yes	No
SMaWL	Yes	No	No	Yes	Yes	Yes
Views 2	Yes	Yes	Yes	Yes	No[4]	No

Tags match control names The XML tags match the names of the controls created. This is the standard format in most, but not all, XML GUI toolkits. For example XHTML uses one tag called input for its controls.

Layout managers The toolkit supports some form of layout managers so that controls do not need to be positioned manually.

CSS support The toolkit supports CSS for formatting the display of the GUI.

Call back events The toolkit uses standard call back methods for event handling.

Platform independent GUIs created with this toolkit are capable of running on multiple platforms.

[4]Views 2 could be platform-independent, but there is only currently one implementation.

2.7.2 Toolkits

Views, XUL and XAML have been discussed in detail in sections 2.4, 2.5 and 2.6 respectively, so they not be discussed again here. This section provides short descriptions of the other toolkits in the comparison table.

2.7.2.1 XHTML

XHTML, or HTML in general, allows creation of fairly simple GUIs. It emphasizes creating text input forms more than full GUIs, but with the addition of JavaScript XHTML can provide most GUI functionality. It also serves in some ways as a baseline for comparisons because it will run on nearly any system. Most computers have a web browser capable of rendering XHTML and interpreting JavaScript.

2.7.2.2 JDesktop Network Components (JDNC)

Sun's JDesktop Network Components (JDNC) [17] is an interesting variation on the usual XML GUI toolkit idea. JDNC is still a work in progress, so like XAML some information on the project is still subject to change. But it is not intended to be a general purpose GUI construction toolkit. JDNC's idea, rather than entirely replacing Swing for general application development, is to try to provide a framework that makes it easy to build more complex applications with short cuts to higher level components. It uses a markup language to specify these higher level components to be used. However, rather than using XML to specify all the boxes and buttons that go into a database display table, it will capture all that functionality in a small amount of XML. JDNC has a lot of interesting ideas and seems like a very promising technology. However, it has a different purpose from Views 2. Sun still advocates Swing — which would be more comparable in purpose to Views 2 — for regular GUI design, so despite the interesting ideas JDNC puts forth, it will not be considered any further.

2.7.2.3 GNU Enterprise Forms

GNU Enterprise Forms [18] is a GNU project that is similar in concept to the JDNC, but unlike JDNC also provides functionality for including lower level GUI controls. Information on GNU Enterprise Forms is somewhat limited, but it follows the same basic principle of having tags for controls that most other XUL GUI toolkits follow. However, like JDNC, it also has shortcuts to higher level components. An example of a higher level component would be, a table to display the results of a database query. In terms of functionality as a basic GUI design toolkit, GNU Enterprise Forms does not offer much over the basic XML GUI toolkit themes that have been repeated in many such toolkits. It aims to broaden the available functional units to speed up design for certain common tasks, similar to JDNC. Views 2 has nothing to compare to these more complicated components that GNU Enterprise Forms and JDNC offer. In terms of GUI design with the more basic level of controls that XUL and XAML already cover GNU Enterprise Forms does not offer anything new. Like JDNC, it will not be compared against Views 2 in detail, although it does have clear areas of strength over Views 2.

2.7.2.4 Bindows

Bindows [19] is a fairly basic XML GUI toolkit. It uses JavaScript for its event handling like XUL and all of the basic tags follow the usual style for these XML GUI toolkits. It only has one major feature of interest to distinguish it; that being that it requires no new software to be installed on the client side. Even XUL requires the user to at least have a nglayout-based browser. Bindows should work in any modern browser; it only requires support for DHTML, XML and CSS. This makes Bindows nearly as flexible as basic XHTML in principle. Although the implementation is not quite as flexible as it seems in principle, it is an interesting idea to have no client side software.

2.7.2.5 SMaWL

Simple Markup Window Language (SMaWL) [20] is an XML GUI toolkit that focuses more on being easy to use than on being fully featured and robust. Only limited information has been published on SMaWL. It appears to share design goals with the original Views. SMaWL follows the idea that the majority of GUI components need only a minority of the available properties adjusted to meet design goals for the project. This idea seems fairly obvious but it is still an important note to make. One major benefit of the XML based GUI specification is that it allows novice designers to specify what they need in a simple format that does not clutter the program.

2.7.3 Summary of the Comparison

There are a few major points that can be made from the patterns emerging from the comparison table:

- Layout managers — even in the basic horizontal and vertical form — are a relatively uncommon feature.
- CSS support is also relatively uncommon.
- The call back event handler style is almost always used, the polling and wait loop styles are rare.
- Platform Independence is a common goal for GUI toolkits.
- Visual designers are usually provided, even when the goals of the toolkit include simplicity of XML specification (as is the case for Views and SMaWL).

Chapter 3

Problems and Solutions: Design Goals

3.1 Deficiencies of Views for Professional Applications

The Views system was originally designed as a companion to *C# Concisely* [2], which is an introductory C# textbook. Since Views was primarily designed for use in this introductory environment it lacks some features that professional developers may wish to use. Views has a simple XML format for specification and if Visual Studio .NET were unavailable, professional developers might find it useful for prototyping. However, in the long run most professional developers would probably choose to hand code WinForms components rather than use Views because of two major drawbacks in the system.

First, the controls and properties that are supported are hard coded into Views. There is a list in appendix F of *C# Concisely* that includes almost all of the controls and properties supported by Views and, compared to the WinForms library, this list is quite small. It is likely that a professional programmer would want to use some control or property that is not supported in Views. The subset of supported controls and properties was selected in such a way that a novice programmer could make good

use of the system after an introductory programming course. While Views does not lack in functionality, it does lack in selection and breadth of control choice that would be desirable in professional application development.

The second major drawback comes in the event handling system. The WinForms API in C# provides a fairly large and diverse set of supported events that a programmer might add callback event handlers to. In Views each supported control has a specific event associated with it that is then registered with the event handler. The events used in Views are not programmer selectable. There is a certain set of events registered for the various controls and the Views system cannot be used to add more. The events that are registered in Views are in general the most obvious action associated with the given control type. For example, most buttons only make use of the click event handler, which is registered by Views. Although the events Views chooses to register will be sufficient in most cases, the fact that there is no method for adding extra events will likely be a drawback to professional developers. While 'click' is the event most commonly needed for buttons, it is fairly common for a professional application to use others as well. For example, many buttons have some functionality when the mouse is hovering over them, which there is no way to accomplish in the original Views system. Views also uses a wait loop style event handler. The programmer makes a call to the `GetControl()` method which blocks until an event is raised and returns the name of the control that raised the event. The program will not register another event until another call to `GetControl()`. There is no way in the regular Views system to handle multiple events independently or concurrently with regular program operation. If two events are raised the first will be passed to the user program and the second event will have no effect until the program calls `GetControl()` again.

The Views system offers a route out of all of these problems by providing an interface to retrieve the actual C# WinForms API objects being used in the GUI. Once the designer has a reference to the actual object, he can of course do anything

with them that he can do in normal C# code or using Visual Studio .NET. It may be assumed that the program complexity of using the C# objects is not a problem because we are dealing with drawbacks to a professional programmer. However, even if it is assumed that the programmer knows how to use C# code to set the properties of the GUI components to the desired values, this circumvents a great deal of Views' usefulness. Views still retains its function as a dynamic *layout manager*, which saves designers from statically setting the coordinates for all the controls. However, the simple XML tag that specifies a control has now been replaced by an XML tag and some C# code. It is impossible to avoid using C# code to set objects' properties completely, as we may need dynamic changes to the GUI, which can not be easily captured in XML. However, the initial state of the program can be captured in the XML and the programmer should not need to fall back on setting C# properties for initial program state setup. Programmers should also not be restricted to the set of controls that are hard coded into Views or to the default set of events that are included in the Views system.

The Views system has some drawbacks to professional programmers. Though there are workarounds to these drawbacks, their use would defeat much of the original purpose of using Views. Views serves a great purpose as an educational supplement to *C# Concisely*. However, the principle of a simple clean XML specification for the static initial state of a GUI has many benefits to professional development, but it is too restrictive. On the other hand, with some enhancements, the Views system could be very useful in this area as well.

3.2 Enhancements made to the Views System

Although Views has some drawbacks, particularly to professional developers, it also has many strengths. Views offers a simple intuitive XML format to specify the initial state of the program. Using XML to specify controls and parameters of the control is

simple, elegant and powerful. It is much easier to read and write this XML than C# property assignment code. Views also offers layout manager functionality, which is not a new idea, but is an important one. Many GUI creation toolkits require statically placed components. Perhaps they are placed by a visual designer or, even worse, by hand coding coordinates. Offering a layout manager is a significant benefit.

Between the layout manager and the XML specification, Views allows creation of high quality, highly maintainable GUIs in a relatively short amount of time. Even though it has limitations, Views serves as a solid foundation for an excellent GUI creation toolkit. With some additions, the Views system could be an excellent toolkit, particularly if it could remain backwards compatible with the current system. The resulting system would have the strengths of Views, overcome its weaknesses and still allow the simple style of the current Views programs. This would not only allow professional developers to make use of the benefits Views offers, but it would also allow new developers to learn from the simple model and to build up to industrial strength application design.

The system can be further improved, not just by overcoming the apparent weaknesses of Views, but by adding new features as well. One strength of Views is that the XML specification is much more maintainable than complex blocks of repetitive C# code. This is already a strong point, but it can be further expanded. It is possible to redesign the XML to make design even easier and provide even higher maintainability. Software design principles tell us to separate design concerns and XML is flexible enough to let us do just that. An XML based GUI design system could allow a breakdown of the XML into major design concerns. For example, there could be control list, layout information, events handled and format information. Each concern can have its own separate XML file that can be designed and altered independently. In fact, if the format of the specification for these concerns is standardized enough, the system could allow designers to select their own level of separation of concerns. The XML files could then all be woven together into an appropriate canonical input

format. XML is easily flexible enough to do this and GUI design could make good use of it.

Given Views as a conceptual starting point and continuing to leverage its strengths, the Views 2 system makes four major improvements over the original Views. Two of the improvements are to address those drawbacks mentioned previously: the event handler has been overhauled and the list of components is now less restrictive. In addition to addressing these two potential problem points, Views 2 adds two additional features: an improved XML format and CSS support. The following sections expand on these areas of improvement, mostly in so far as related to features seen by the user. Chapter 4 discusses the design rationale for these ideas.

3.2.1 Event Handling in Views 2

The event handling model in Views was discussed in section 3.1 and was contrasted to the more common callback style event handlers. Views 2 adds support for these callback style event handlers. Views selected a few key events to support and then provided a single event handler for all of them. The GetControl() method of the Views.Form class simply returned the string name of the control that had raised an event. No information was returned about the event, just a string name. This simplification is beneficial to novices and probably harmless for intermediate programmers. However, some more advanced programmers may require the information that is lost in this process or require more information about the events than is given implicitly in this system.

Views 2 exposes the full set of WinForms' available events. Event handlers in C# have a standard signature, so users must provide a method with that signature to handle the event they wish to capture. This is identical to what would be required by a regular C# program (including those written in Visual Studio .NET, which simply provides a stub). Once these methods exist the designer can simply add an attribute-value pair to the XML tag for the control the event is raised by, where the attribute

name is the name of the event and the value is the name of the appropriate event handler.

3.2.2 Supported Controls

Views translates XML to C# objects using static tables and switch statements. This results in the major drawback that if a control is to be supported then it and all of its supported attributes must be hard coded into the Views system. Views 2 does not use hard coded tables and switch statements to translate the XML. Instead, it makes extensive use of reflection to query the underlying API for controls and properties that match the names of the tags in the XML and then reflectively instantiates them. Therefore, the entire set of controls in the underlying API is supported by Views 2. Thus Views 2 automatically picks up new controls if they are added to the underlying API.

3.2.3 XML Format

Views' XML format uses a tag named for the Windows control with attribute value pairs for the properties the designer wishes to set.

For example: `<Button Name="Button1" Text="OK" Width="75" />`

This is a good format for simple values (i.e. primitives and a select few object types). However, in some cases, the designer may want to provide an arbitrary object type as a value for a property. For example, an image might be displayed on a button, so the designer needs to specify an image object in the XML. The XML specification in general and the Views style notation in particular are flexible enough that arbitrary object types could be specified. However, there is a problem: arbitrary XML cannot be specified as the value of an attribute-value pair. Views 2 uses a rearranged XML notation so that properties of a control are XML children tags, with their values being the children of those tags. It is easy to convert XML using the attribute-value

pairs into nested tags. In addition to allowing arbitrary object types to be specified by this notation, it also allows for more convenient string values because the kind of characters that can be included in an XML node is much less restrictive than the kind of characters that can be included in a value.

The Views 2 style of the tag above would look like:

```
<Button>
  <Name>Button1</Name>
  <Text>OK</Text>
  <Width>75</Width>
</Button>
```

3.2.3.1 Separation

In addition to using this more standardized XML format, Views 2 allows the separation discussed previously. The attributes that apply to one control can occur in more than one XML location. The creation of this XML is very simple — the designer just creates a new control tag and gives it the same name as a previous one. Views 2 will collect all the relevant XML about this control from all the XML passed in and weave it together to form one final XML block for the control.

For example, consider the following two XML fragments:

```
<Button>
  <Name>Button1</Name>
  <Text>OK</Text>
</Button>
```

and

```
<Button>
  <Name>Button1</Name>
  <Width>75</Width>
```

```
</Button>
```

Because these two fragments of XML are the same type (`Button`) and have the same name value (`Button1`), they would be merged together to create one button identical to that created by the single XML block in the previous example.

This is not the most compelling example because these two fragments contain only one property each and the overhead seems to outweigh the benefits. But this is just a simple conceptual example. If there are dozens of properties being set for dozens of components this breakdown becomes more beneficial, allowing designers to separate the XML into whatever relevant concerns they see fit.

3.2.4 CSS Support in Views 2

In a graphical application, it is quite common for groups of controls to have several related properties. As an obvious example consider a set of radio buttons. They probably should all have the same colours and font and likely the same width and height. The programmer should not have to duplicate all this information in the XML or in regular C# property assignments. CSS provides an easy way to capture exactly the desired functionality. It allows values to be properties for sets of objects to be given the same values. Considering the separation of concerns discussed previously, it is quite likely that there will be an XML file for formatting information. But this can be even better accomplished by use of CSS.

Views did not offer CSS support, it is an added feature in Views 2. HTML and XUL both support CSS, and so it is logical to apply it to an XML specification such as Views 2. The CSS blocks simply specify in CSS format children nodes to be added to control specifying tags in the XML source. In Views 2, like Views, most controls are required to have a Name attribute, so where relevant, this can be used for CSS id type selectors (i.e. #abcd), which use the id attribute in HTML. Aside from this minor variation in id selectors all standard CSS uses apply in a very natural way to

Views 2.

There is no requirement to use CSS in Views 2, it simply replaces the XML for one possible independent concern. A designer could still use XML for this concern if they preferred. In fact, the CSS is woven into the XML just as another XML file would be. The back end of Views 2 does not process CSS. CSS is offered because it is very appropriate for formatting in this situation and applies in a very natural way.

3.3 Improvements over Visual Studio .NET

Views 2 offers a improvement in Visual Studio .NET in two key areas. Visual Studio .NET is a good tool if everyone in the development team is using it, however, it does have drawbacks. One issue is that Visual Studio .NET generates a large quantity of code that is difficult to read. In principle, as long as the designers have the Visual Studio .NET files and everyone in the development team is using Visual Studio .NET there should be no need to edit this code by hand — although it is sometimes unavoidable.

Should someone need to read through the code for any reason, it is not easy code to read. It has been generated by Visual Studio .NET to fill out all the properties the visual designer has specified, not to be read by human programmers. If the Visual Studio .NET project files are ever lost or outdated by a new version of Visual Studio .NET and only the C# source code remains available, any future maintenance will require the maintainer to read the source code generated by Visual Studio .NET. The XML of Views 2 is easy to write, easy to read and designed with state-of-the-art Software Engineering principles in mind. It will improve the ease of performing maintenance tasks in these circumstances. All the maintainer needs to read is the XML and have a basic understanding of the format to make changes to the program. There is no need to read hundreds of lines of machine generated C# code.

In addition to the problem of maintenance, Visual Studio .NET does not allow

programmers to develop an understanding of the underlying API. It flags any code it generates with "Do Not Modify" comments so people using it to generate GUIs do not get a chance to learn what it is doing. Designers using Views (1 and 2) will usually hand code XML tags to generate the GUI. The XML tags are much simpler than the API calls to set the objects up. However, the XML uses the same names as the class and field names in the API. By doing this programmers gain understanding of the libraries as they build their applications, instead of just dragging and dropping components into place. Visual Studio .NET is a powerful tool but in many ways it encourages dependency on itself. Views 2 allows designers an easy way to specify GUIs and at the same time builds their knowledge of the APIs. This way, a designer will learn enough about the WinForms API that he would be able to create WinForms programs with no GUI creation tools.

Views 2 offers significant advantages over Visual Studio .NET in the two key areas of maintenance and programmer education. Having programmers learn about the APIs they are using becomes less important the more experienced they are and the older the system is, however, maintenance becomes more important in just these areas. Between these two key benefits Views 2 offers advantages over Visual Studio .NET in the entire life cycle of a software system.

3.4 Comparisons with other GUI toolkits

Views 2 is not the only XML based GUI toolkit available. The previous section highlighted some of the advantages of Views 2's XML specification in comparison to Visual Studio .NET. However, Views 2 is not the first or only XML based GUI toolkit. XUL has been in use for some time now and many of its features have been incorporated into some of Views 2's design. Microsoft's XAML is not yet officially available, although, there are third party designed systems available for it. While there are other XML based GUI toolkits available, comparing Views 2 in detail to all

of them is unnecessary. XUL and XAML will be used as a representative sample.

3.4.1 Views 2 compared to XUL

Mozilla's XUL is a fairly powerful application development tool and, like Views 2, uses a combination of XML and CSS to specify a GUI. In many ways XUL builds upon HTML and, in Netscape and Mozilla, is rendered by the same engine as HTML. This is a great strength of XUL and offers a path for designers skilled at HTML and CSS to grow into more complicated application design. However, because it was based on HTML it maintains an event model similar to HTML and uses JavaScript as the code behind the events of the application. JavaScript is a good at what it is intended for, but it is only a scripting language, and having a fully featured programming language like C# behind the application allows for much better code and more complicated functionality for the application over all.

XUL may even serve as a gateway into Views 2 for people growing from HTML, CSS and JavaScript into more powerful programming languages for application design. The step from HTML to XUL expands the XML to include more powerful application features for controls, layout and design. And the step from XUL to Views 2 replaces JavaScript with C#, to complete the transition into designing GUIs with XML and having a fully featured modern programming language for application logic and functionality.

3.4.2 Views 2 compared to XAML

Microsoft's XAML and the Windows version of Views 2 (the Windows version is, at present, the only version of Views 2) have a great deal in common at a first look. They both grew out of the present .NET Framework class library and properties, so many tag and property names are similar. However, Views 2 excels in three main areas. First, Views 2 supports CSS, and so far there is no indication that XAML

will support CSS for assignment of properties to classes of controls. Second, Views 2 supports layout managers. Once again this is missing from XAML and there is no indication that they are going to be included as XAML nears final release. Layout managers, even simply the horizontal and vertical ones supported by XUL, are very useful in creation of the XML specification.

The third benefit stems from the new XML format mentioned earlier. XAML also needed some way to address the problem of specifying arbitrary C# objects as property values. In Views 2 this was addressed by making the structure of property assignments the XML node's children instead of attribute-value pairs. XAML uses attribute-value pairs, which are still accepted in Views 2 for primitive type properties. XAML requires the designer to place the XML node that specifies the object he wants in another section of the file, which is not part of the main GUI specification. The designer then gives this XML fragment a name and uses its name as the value of the attribute-value pair for the appropriate property. The end result is similar to Views 2's style, but this separates the object specification from where it is used, and requires a reader to jump around in the XML to figure out what XML is attached to the name they are reading. The Views 2 style has the appropriate XML grouped together logically so it does not interrupt the thought process while reading or writing the XML specification.

Chapter 4

The Design of Views 2

4.1 Introduction

This chapter introduces how Views 2 works. Chapter 3 described the drawbacks and benefits of Views when used in professional applications and why it serves as a good basis for a more powerful toolkit. That chapter also briefly addressed some of the enhancements made, but from the perspective of why a designer might choose this system. This chapter begins an overview of why things are done the way they are. Chapter 5 contains an extended example of a working Views 2 program to demonstrate how these ideas fit together.

4.2 Overview of the approach

Views 2 is a GUI creation toolkit. It takes an XML input that specifies the initial state of the GUI of a program and then generates that GUI using the native components of the underlying API. The XML is accepted in several formats, but before processing by the main component of Views 2 it is transformed into an internal format. See section 3.2.3 for an outline of the XML format and appendix A for details on the XML specification, particularly section A.1 for details on the back end canonical form.

The XML is then passed to an engine that recursively uses reflection to build the

object structures specified. The engine is passed a tag that represents a control to be constructed. Its children are named for properties that can be set on that control. The children of these properties are then the values of those properties, which may themselves be object types so a similar type of construction is carried out for the appropriate object type. Any type that can be adequately represented by a single string may be a simple text child that contains the value, which then needs to be converted to the appropriate type. A control's children may instead be other controls to be nested in the current control. For these the model simply recurses on itself. With a static parser this scheme would cause problems because of the dual nature of a control's children. However, because Views 2 uses reflection to generate objects it can determine at runtime if a node is a property or a sub-control. This serves as the core model for the Views 2 system. There are some additional complications that need to be handled, but the core model used by most of the system follows this design style (section A.3 gives a detailed description of the generator algorithm).

One of the great strengths of the original Views system was its capability as a layout manager, and Views 2 does not give that up, although there are small differences. In the WinForms framework any control can contain other controls, although it is not always sensible to do so. Views 2 builds on this by allowing users to associate a layout manager with any control. The layout manager is a simple interface with one function responsible for positioning all the controls that are contained within itself. In general this does not seem to make sense; for example a button containing controls and having a layout manager associated with it seems senseless, and it is. But `Panel` and `GroupBox` are also controls, and they do contain other controls. Since the members needed to contain controls are part of the abstract parent class `Control`, all controls can in principle contain other controls. Likewise, all controls may have layout managers associated with them. Views 2 does not restrict this in any way; it leaves it up to the user to decide if having a layout manager for a particular control makes semantic sense. Beginning from the top level Form layout manager, the layout

managers recursively apply to their respective sections of the GUI until all controls have been positioned and laid out appropriately.

The *event dispatcher* in Views 2 is a class designed to attach objects created with reflection to event handlers that programmers write in regular C# code. The user writes his or her event handlers in a class, creates an instance of that class and then passes it to the Form constructor (often just the 'this' reference). The XML then may simply specify some string that is the name of a method to be used as an event handler. This method must exist in the class of the object passed to the form constructor and have the proper signature for a WinForms event handler. Views 2 links the event to the method found in exactly the way expected so that it fully supports the regular underlying event model of WinForms. This allows designers to register any events they wish to be handled by a function they write, and it is all captured with a simple XML node of the form `<EventName>HandlerMethodName</EventName>` added as a child of the control.

The last major piece of Views 2 is what is known as the *special case handler*. This module attempts to deal, as elegantly as possible, with the fact that not all objects a designer may want to construct will fit nicely into the Views 2 construction model. The "Construct and Set" style of object creation (refer back to figure 2.1 for an example of "Construct and Set") works for nearly all the objects in the System.Windows.Forms namespace. It also works for many other objects in related namespaces, but it does not work for *all* objects. Sometimes a designer will wish to use objects that do not work exactly the way the model needs them to. For example, an `Image` object is created using the `FromFile()` method of the `Image` class, rather than creating an object for the image and setting some filename property, as the "Construct and Set" scheme would require. To successfully integrate this into the XML, the special case handler provides a method to check for the presence of a special case based on information known at the present location in the XML. The Views 2 system can then decide if a special case handler is available and how to act

in this situation. The special case handling system was carefully designed to be as regular as possible, as well as being fully extensible so that new special cases can also be handled. Views 2 has a powerful model for using reflection to build objects from the API based on an XML specification, but it depends on certain parts of the API being public. If they are not public, or do not exist at all, the model breaks, and this is dealt with by the special case handler.

4.3 Rationale of the design

4.3.1 XML Format

The XML Format of Views 2 has been designed to allow software designers the most flexibility in designing their applications. It is designed to give them choice and flexibility of design, without being overly confusing in the options presented or forcing the designers to conform to predetermined design rules. After all, a toolkit should assist users in performing a task, not constrain how they do that task. The XML has been designed to be powerful and consistent with modern software design techniques but not force certain design constraints on the designers.

The simple and logical basics of the XML format are repeated throughout nearly every system that uses XML to specify GUIs. Names of the types of Controls — specifically class names in C# — are used as the name of the tags. This seems like the most natural way to handle the basic controls. If the designer wants a button then the XML tag `<Button>` is used. Views 2's back end structure requires that properties of the control be specified as child nodes of the control with their values as their respective children. Many XML packages use attribute-value pair notation for properties of the controls; this is how the original Views handled it. In some ways this seems to make more sense, as having one tag for each control and its properties as attributes is a logical model. Views 2 deviates from this because the child-node

model gives more flexibility in specification.

Using attribute-value pairs only allows a string for the value. This string can adequately represent all primitives (given trivial conversion routines), but it cannot represent more complicated object types; for example, an Image object might be desired. XAML provides one way to solve this problem by defining nodes elsewhere in the XML tree and then referring to them by an assigned name. However, this seems unnecessary since XML is fully capable of providing this sort of object specification simply by using child nodes for attributes and their children for values. These values can still be strings to cover all the same values available in an attribute-value pair style and in addition they could be XML nodes to specify more complex types. As a side benefit, the kinds of characters allowed in an XML text node are less restrictive than those in an attribute-value pair value.

Views 2 does not abandon the attribute-value pair style altogether, since it is more concise than the child node style for primitive values. The attribute-value pair style may still be used; in fact, the attribute-value pair and child node styles can be mixed for one control. Views 2 processes either style and produces the same effect. Ultimately designers are given the flexibility to specify more complex object types within the XML, but are still free to use the attribute-value pair style if they prefer it.

Views 2 also allows XML to be split across multiple input files. The way this splitting occurs is up to the designers. All of the properties of all controls can be specified in one file if the designers prefer, which may be the case for a very small form; for example, an options pane with a few check boxes and 'OK' and 'cancel' buttons. Alternatively, the properties may be split up into multiple files. However, rather than imposing a predetermined scheme for how users should divide their properties, Views 2 simply offers this functionality. The designer can put any properties into any XML file and pass in all the files and Views 2 will merge them. Section 3.2.3 gives a simple example of this, and chapter 5 contains a large scale Views 2 XML

example. Views 2 makes no logical distinction between the files passed in to it (with the obvious exception of CSS files). This splitting of XML is designed to promote separation of design concerns, although it is designed only to *encourage* it, not *force* it upon users who do not desire it. Designers may decide for themselves which logical concerns to separate their programs into. For example, a designer might desire a control list XML, a layout XML, an events XML and a formatting CSS file. They could design these four separate files and Views 2 will merge them and generate one XML. However, this set of files is not prescribed; because, if a designer is producing a form with a very simple layout he may not want to be bothered to generate a whole layout XML. He or she may just combine that information with the controls list, for example. This flexibility was very important to Views design. It allows the level of design principles being used to be customized the project.

4.3.2 Layout Managers

The Layout Managers in Views 2 are relatively simple concepts. In the .NET Framework, each control has a property `Controls` that points to a group of controls contained within it. It does not always make sense to contain controls within a control, but it is syntactically uniform. In Views 2 each control can also have an associated layout manager for laying out those controls. Many systems use tags to denote simple layouts — like `<vertical>` and `<horizontal>` in Views or `<vbox>` and `<hbox>` in XUL. These tags contain controls and group them into logically related units. However, the .NET Framework already provides control types for this purpose, namely Panel and GroupBox. Panel in particular is a lightweight control that has no purpose except to logically group controls; a Panel has no visible presence in the GUI (a GroupBox is similar, but has a border and title). Since these concepts already exist, it makes sense to simply apply a layout manager to a Panel or GroupBox, rather than to introduce a new concept of vertical and horizontal layout groups. The layout manager system is designed to be extensible from both sides. There may be new

control types that could contain controls, and there may be new layout managers.

There is no harm in accepting `<vertical>` as a synonym for a Panel with a vertical layout manager — which Views 2 does do — but to have separate logical concepts makes less sense. Panel already exists as a logical grouping of related controls; Views 2 simply adds a layout manager to that logical grouping. Layout managers are supported for all controls so if a new grouping control is added it can automatically have layout managers and likewise any new layout managers can be easily added by extending the layout manager interface. Views 2 makes use of the existing .NET Framework method of logical control grouping, but retains the usefulness of the original Views' layout managing and expands on it. Views 2 expands on the basic layout managers of Views by adding additional layout managers and allowing designers to provide their own, which requires programming one relatively simple method in a lightweight class. Designers can easily use their custom made layout managers with other applications they write; they do not need to write a function for layout logic every time. Each control may contain controls and may have an associated layout manager. Starting from the top level form they are simply called recursively to end up laying out the entire form in a simple to understand and easy to extend manner.

4.3.3 Event Handlers

The event handling subsystem of Views 2 is designed in a fairly obvious manner and why it is done in that manner does not require much explanation. The goal was to support any event the .NET Framework provides in a consistent XML syntax. As mentioned previously, the XML node `<EventName>HandlerMethodName</EventName>` added as a child of a control allows the named event to be handled by the named method. This syntax is the same as property setting XML (even in that it can be specified in attribute-value pair notation). It is not difficult to use reflection to determine if a particular name is a Property or an Event. Views 2 does exactly that. Given that it is an Event, Views 2 checks for a method appropriately named and

appropriately typed and connects it as an event handler.

C# uses *delegates* to provide event handling. The relevant event handler delegate has a method added to it, which can then be executed through the delegate. Views 2 makes use of this by using reflection to create a delegate to serve this purpose. An object reference must be passed to the constructor[1] when a Views 2 `Form` is created. When Views 2 encounters an event property it searches this argument object for a method of the appropriate name (and the regular event handler signature) to invoke for this event. It then uses the reflection framework to instantiate a delegate that will invoke this event and links it to the appropriate event hook in the `Control` object. Event handling then proceeds as normal for any C# program.

4.3.4 Special Case Handler

The special case handler is a subsystem of Views 2 that attempts to localize exceptions to the Views 2 model. In terms of the abstraction for the model, this piece of the Views 2 system offers nothing. Software design principles are the only rationale for the special case handler. It gives one interface to a bunch of non-standard routines and allows the primary code base to be simple and conform to the model that has been laid out. This gives modularity for the special cases and keeps it clear where Views 2's model deviates from the .NET Framework. The special case handler is an implementation issue, which highlights the fact that the Views 2 model does not perfectly match the WinForms classes.

[1] If there are no events in this application the argument may be 'null', since with no events to create handlers for it will never be referenced

4.4 How Reflection Improves XML GUI specification

4.4.1 XML for GUI Specification

XML is an ideal medium for GUI specification for many reasons. Views is not the only, or even first, GUI system to use XML for its specification. Despite it becoming something of a standard a review of why it is good for this purpose follows. XML is by design portable, flexible and extensible. For Views these are very desirable qualities as it is designed to be multi-platform. It is also useful for Views 2 because the control set is not static as it is in many other GUI systems. Any format that limits the control set specification would be unsuitable. It seems obvious, but the extensibility of XML is very important for Views 2. A particular XML data block is structured as a static tree format, which is very good for GUI specification. The top level window or frame is the root of the tree and then it can be subdivided into sections (panels in WinForms) for the internal nodes, and the controls are the leaves. This is a slight oversimplification, but the general metaphor works well for specifying the set of controls to appear on a GUI form. XML of course specifies only a static tree, but since it is only used to specify the initial state of the GUI, a static specification is all that is required.

4.4.2 How Reflection Improves the Situation

Many toolkits use XML to specify their initial GUI layout. The extra step Views 2 takes is to use reflection to convert the XML into controls. This step does not make a huge impact to the end user program, but it makes a great difference in the implementation. Views 2 does not require a hard coded list of supported controls, which makes the code shorter and more regular. The use of reflection means that Views 2 does not need to be updated if there are new controls added to the .NET

Framework. As long as they are subclasses of System.Windows.Forms.Control, like every other control is, Views 2 will immediately recognize them and users can use the new controls names as tags in their XML specification. This is made possible because Views 2 uses reflection to generate controls dynamically. This is just an overview, since reflection has little impact on the end user and detailed understanding is not required to see the benefits for Views 2. Detailed information on this subject is in the appendix section A.3 that details the algorithmic logic Views 2 uses to create objects from the XML.

Chapter 5

Views 2: A Detailed Example

5.1 Introduction to Views 2

Chapter 4 discussed how the Views 2 system works. In order to show how all of the concepts discussed previously work together, this chapter offers an example of the XML and CSS portions of a working Views 2 program. We will use a slightly modified version of a long running Views example program: the Colour Picker applet. This applet has been used to demonstrate a complex Views sample program, and then used to illustrate how Views could be improved. It was also developed as the first major program for the first partially working version of the Views 2 code. Now it will be used as an example for evaluating Views 2.

As discussed previously, developers are able to separate the concerns of their software design project in any way they see fit. This particular example is split into four sections: the first is the layout specification, the second is the general properties specification, the third is the events specification, and the fourth is a small CSS file for formatting information. Refer to figure 5.1 for to see what the resulting program will look like. Most of the ideas have been discussed already, so this chapter will be fairly brief. It is meant only to illustrate the ideas and serve as a basis for comparison and evaluation in chapter 6.

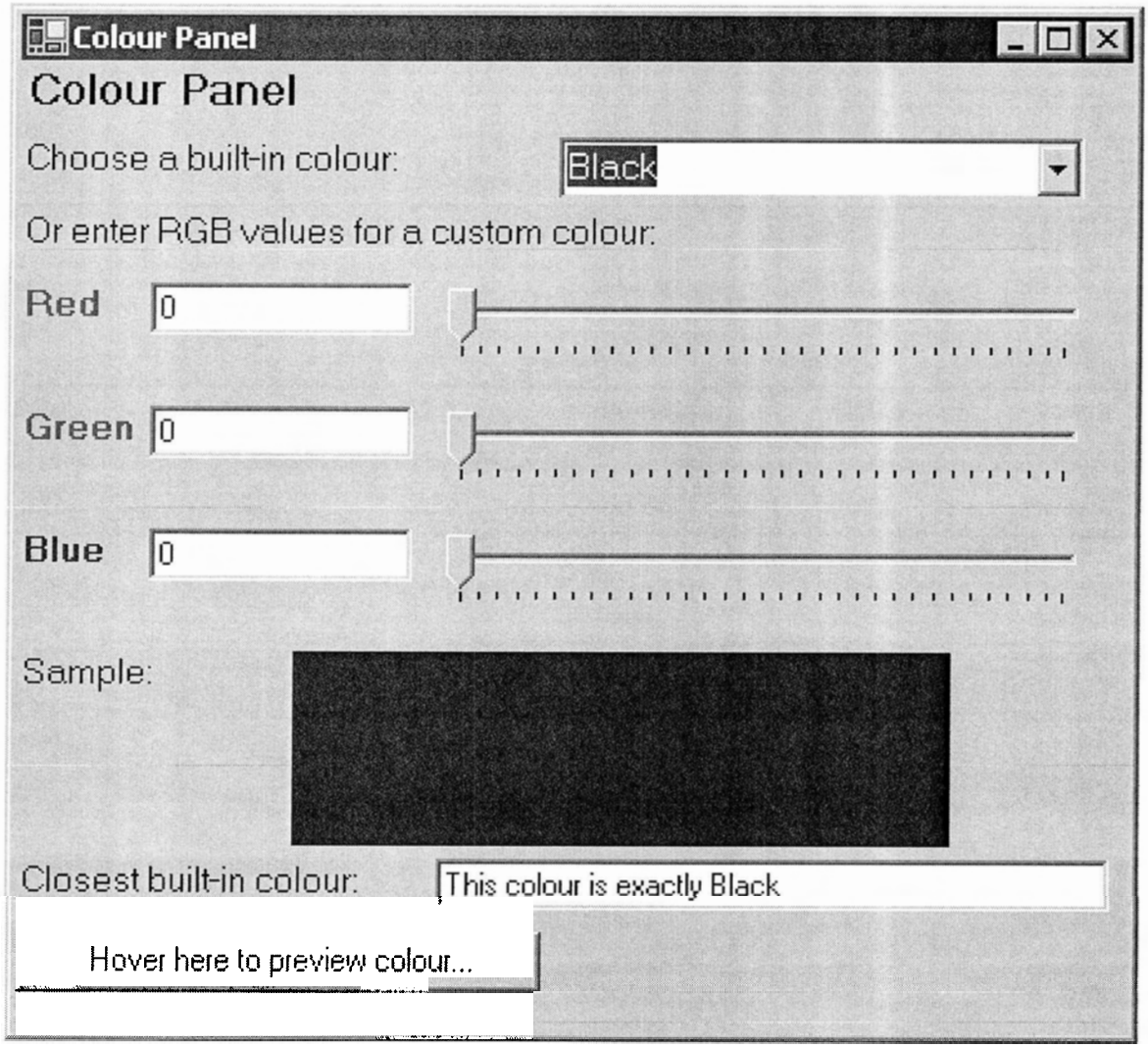

Figure 5.1. *The (reduced) Colour Picker Applet*

5.2 Control Specification and Layout Managers

Figure 5.2 shows the first piece of the XML for specifying this program. It specifies which controls exist in the program, which other controls they are nested inside of (primarily for panels) and which layout manager is associated with which control. When this XML is processed it will create controls and place them as children of the appropriate parent controls. It will also set their Name attributes which can then be used to refer to these controls later. As is shown, layout managers are added by supplying the attribute `layout-manager` for a control and giving it the value of a recognized layout manager. The set of recognized layout managers can easily be extended to include custom layout managers.

5.3 Merging Multiple XML Files

Figure 5.3 shows the second XML component for specifying the Colour Picker Applet. It revisits many of the controls specified in the previous section by opening a new tag of the same control type with the same name. Views 2 will recognize that these two pieces of XML are for the same control and will add the attributes specified here to those previously specified for this control. The layout and nesting of controls need not be specified again. Because names are required to be unique in Views 2, every control can be flatly specified within the Form tag on its second (and subsequent) occurrences. If the value of this node's `Name` property has been encountered before, Views 2 will find the control in the right nested location. The `ComboBox` control demonstrates the child node property style, all other controls use the attribute-value style because they do not have any attributes that require the child node style.

There is another minor feature of Views 2 illustrated here: the `<primitive-array>` tag that is used to specify an array of primitives (in this case strings). An array of primitives is generated by splitting a string into substrings and converting them into the elements of the array. A `<primitive-array>` has two properties `separator` and

```
<Form layout-manager=vertical>
  <Label Name=TitleLabel/>
  <Panel Name=ColourListPanel layout-manager=horizontal>
    <Label Name=ColourListLabel/>
    <ComboBox Name=ColourList/>
  </Panel>
  <Label Name=RGBValuesLabel/>
  <Panel Name=RedPanel layout-manager=horizontal>
    <Label Name=RedLabel/>
    <TextBox Name=RedBox/>
    <TrackBar Name=RedSlider/>
  </Panel>
  <Panel Name=GreenPanel layout-manager=horizontal>
    <Label Name=GreenLabel/>
    <TextBox Name=GreenBox/>
    <TrackBar Name=GreenSlider/>
  </Panel>
  <Panel Name=BluePanel layout-manager=horizontal>
    <Label Name=BlueLabel/>
    <TextBox Name=BlueBox/>
    <TrackBar Name=BlueSlider/>
  </Panel>
  <Panel Name=SamplePanel layout-manager=horizontal>
    <Label Name=SampleLabel/>
    <PictureBox Name=Sample/>
  </Panel>
  <Panel Name=ClosestColourPanel layout-manager=horizontal>
    <Label Name=ClosestLabel/>
    <TextBox Name=SystemBox/>
  </Panel>
  <Button Name=Select/>
</Form>
```

Figure 5.2. *The Layout Specification XML*

```
<Form Text='Colour Panel'>

<Label Name=TitleLabel Text='Colour Panel' Font=BoldSans12 Width=350/>

<Label Name='ColourListPanel.ColourListLabel'
  Text='Choose a built-in colour:' Width=200/>

<ComboBox Name='ColourListPanel.ColourList'>
  <Font>Sans10</Font>
  <Width>200</Width>
  <Items>
    <primitive-array>
      <separator>;</separator>
      <values>Black;Red;Green;Blue;Custom</values>
    </primitive-array>
  </Items>
</ComboBox>

<Label Name='RGBValuesLabel'
  Text='Or enter RGB values for a custom colour:' Width=350/>

<Label Name=RedLabel class=RGBPanel Text=Red/>
<TextBox Name=RedBox class=RGBPanel Text=0/>
<Label Name=GreenLabel class=RGBPanel Text=Green/>
<TextBox Name=GreenBox class=RGBPanel Text=0/>
<Label Name=BlueLabel class=RGBPanel Text=Blue/>
<TextBox Name=BlueBox class=RGBPanel Text=0/>

<Label Name='SamplePanel.SampleLabel' Text='Sample:'/>
<PictureBox Name='SamplePanel.Sample' BackColor=Black
  Width=250 Height=75/>

<Label Name='ClosestColourPanel.ClosestLabel'
  Text='Closest built-in colour:' Width=155/>
<TextBox Name='ClosestColourPanel.SystemBox'
  Text='This colour is exactly Black' Width=256/>
<Button Name='Select'
  Text='Hover here to preview colour...' Width=200/>

</Form>
```

Figure 5.3. *The Detail Specification XML*

values. The array is constructed in a straightforward manner by splitting values at every occurrence of separator[1].

5.4 Handling Event Handlers

Figure 5.4 shows the section of the XML that specifies the event handlers that will be used by this GUI. Though the properties are events rather than visible parameters of the controls, this section is handled in the same way as the section in figure 5.3. This section of the XML is only included in the interest of completeness. Refer back to section 4.3.3 for details on how event handlers are managed by Views 2.

5.5 CSS Specification of Properties

In addition to the multiple XML inputs, the Views 2 system can also accept CSS style input. The CSS support at this time is not true CSS; it is a CSS syntax style specification of C# properties. The CSS blocks are interpreted in an obvious way, with the properties and values shown having precisely the same meaning as if they were added as XML children of all nodes to which that CSS block applies. For example, the second CSS block in figure 5.5 applies to all Labels and sets their font to Sans10. This is exactly the same as adding <Font>Sans10</Font> as a child node of every <Label> tag. Of course, the benefits for maintenance and modifiability are obvious. If it is decided that all fonts should be increased to 12 point, the change needs to be made in only one place.

[1]These are preprocessed by Views 2 the same way as any other tag, so the attribute-value pairing syntax may be used instead of the child elements demonstrated above

```
<Form Text='Colour Panel'>

<ComboBox Name='ColourList'>
  <SelectedIndexChanged>
    ColourList_SelectedIndexChanged
  </SelectedIndexChanged>
</ComboBox>

<TextBox Name=RedBox>
  <TextChanged>RedBox_TextChanged</TextChanged>
</TextBox>

<TrackBar Name=RedSlider>
  <Scroll>RedSlider_Scroll</Scroll>
</TrackBar>

<TextBox Name=GreenBox>
  <TextChanged>GreenBox_TextChanged</TextChanged>
</TextBox>

<TrackBar Name=GreenSlider>
  <Scroll>GreenSlider_Scroll</Scroll>
</TrackBar>

<TextBox Name=BlueBox>
  <TextChanged>BlueBox_TextChanged</TextChanged>
</TextBox>

<TrackBar Name=BlueSlider>
  <Scroll>BlueSlider_Scroll</Scroll>
</TrackBar>

<Button Name='Select'>
  <Click>Select_Click</Click>
  <MouseEnter>Select_MouseEnter</MouseEnter>
  <MouseLeave>Select_MouseLeave</MouseLeave>
</Button>

</Form>
```

Figure 5.4. *The Event Specification XML*

```
TrackBar {
  Width: 256;
  Maximum: 255;
  TickFrequency: 8;
  LargeChange: 32;
}

Label {
  Font: Sans10;
}

.RGBPanel {
  Font: BoldSans10;
  Width: 45;
}

#RedLabel {
  ForeColor: Red;
}

#GreenLabel {
  ForeColor: Green;
}

#BlueLabel {
  ForeColor: Blue;
}
```

Figure 5.5. *The Formatting CSS*

Chapter 6

Evaluation of Views 2

6.1 Introduction

This chapter compares Views 2 to both XAML and XUL to highlight areas where improvements to the state of the art have been made. In chapter 5 the colour picker application (see figure 5.1), was presented. This chapter continues with that example in order to evaluate how well Views 2 compares to other similar toolkits. The application (as near as possible) has been reproduced in both Microsoft XAML and Mozilla XUL to see how the specification in those toolkits compares to Views 2.

XAML can support the same controls as Views 2, since it is designed by Microsoft to be the main design tool for the next generation of Windows and the .NET Framework. XAML supports some new controls as well, but for this example XAML will be referencing the same WinForms API that Views 2 uses. When comparing Views 2 to XAML, the focus will be on what each toolkit can do using the same underlying API.

XUL has its own set of controls, and, as a result, does not support controls such as the trackbar (sliding arrow). XAML has a slightly wider range of control selection than XUL, but XUL CSS support as well as basic layout managers. Since Views 2 also has support for CSS-based style sheets and layout managers, XUL serves as a comparison on the basis of these features.

These two are taken as a reasonably representative sample of XML GUI toolkits

because most others are small variations on the same theme. XAML shows what can be done with the WinForms control set and XUL demonstrates extra utility features. Therefore, it is unnecessary to evaluate every possible toolkit. XAML and XUL will be used to give a detailed look at what the current state of the art looks like compared to the Views 2 example in chapter 5. This should allow a relatively concise, fair evaluation without including dozens of very similar examples.

6.2 Views 2 compared to XAML

6.2.1 XAML's Colour Picker

Using Xamlon, an (approximate) XAML version of the Colour Picker from chapter 5 has been produced. The first thing to note upon looking at figure 6.1 is that it looks almost identical to the GUI produced by Views 2 (figure 5.1). This is to be expected since the XAML is specifying controls from the current standard set supported by WinForms, in the exact same namespace Views 2 operates in. A few minor spacing differences in the layout and a few slightly different fonts are the only differences in the resulting GUIs.

Figure 6.2 contains the entire XAML listing for this GUI. This version of the Colour Picker does not include events. Adding events to a XAML application was covered in section 2.6. The overall complexity increase is small: a `<?Mapping?>` tag and an extra attribute-value pair for each event.

6.2.2 The Comparison

XAML uses the standard style of XML specification. Tags are control types and attribute-value pairs specify the properties. It is a concise elegant declaration of the components and controls to be included in this GUI. XAML is a great step forward compared to the "Construct and Set" standard for C#, but as powerful as it is there

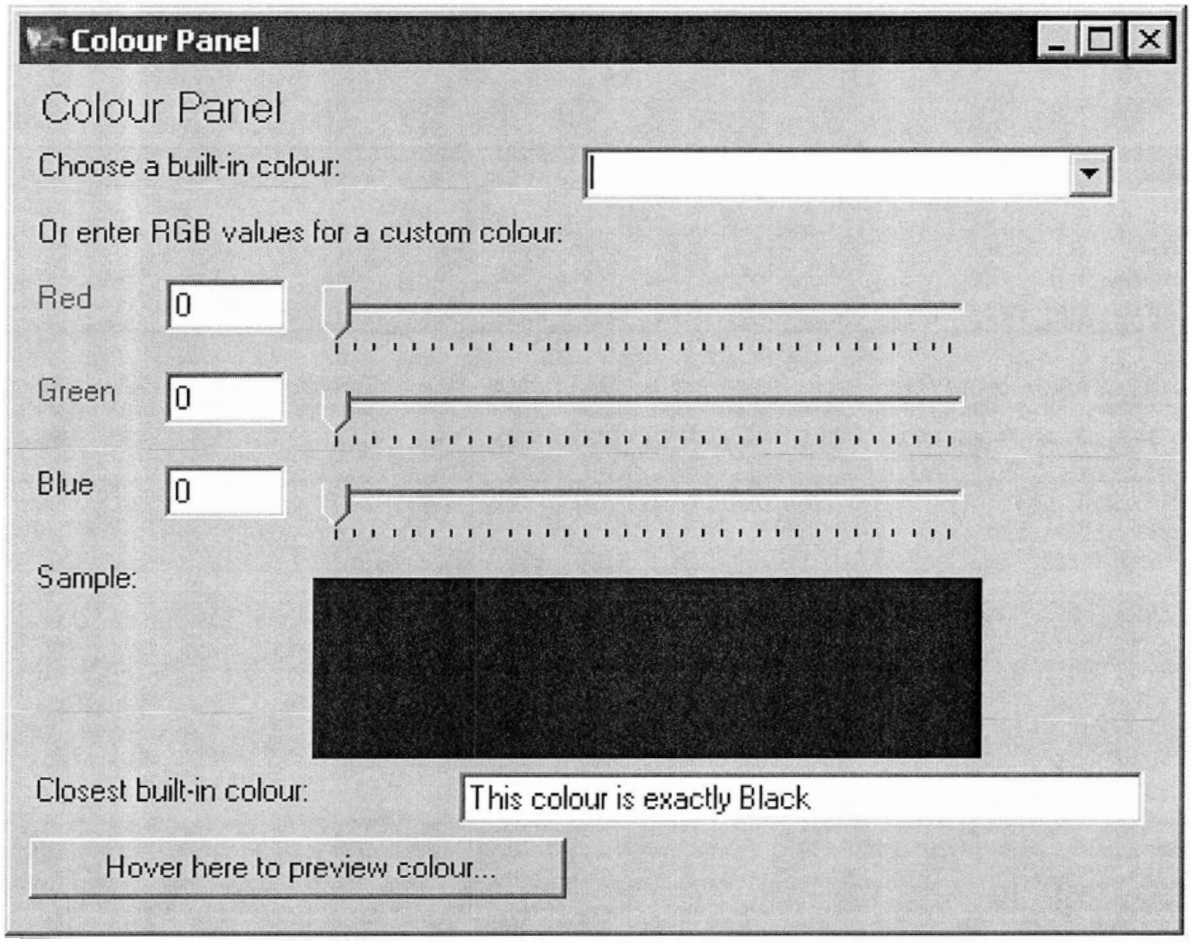

Figure 6.1. *The XAML version of the (reduced) Colour Picker*

```
<?xml version="1.0"?>

<?Mapping XmlNamespace="wf" ClrNamespace="System.Windows.Forms"
    Assembly="System.Windows.Forms"?>

<wf:Form Name="Form1" Text="Colour Panel" Width="500" Height="350"
    xmlns="http://schemas.microsoft.com/2003/xaml/"
    xmlns:def="Definition" xmlns:wf="wf">
  <wf:Form.Controls>
    <wf:Label Name="TitleLabel" Location="5,5" Text="Colour Panel"
        Width="350" Font="Microsoft Sans Serif, 12pt"/>
    <wf:Label Name="ColourListLabel" Location="5,30"
        Text="Choose a built-in colour:" Width="200"/>
    <wf:ComboBox Name="ColourList" Location="210,30" Width="200">
      <wf:ComboBox.Items>
      <String>Black</String>
      <String>Red</String>
      <String>Green</String>
      <String>Blue</String>
      </wf:ComboBox.Items>
    </wf:ComboBox>
    <wf:Label Name="RGBValuesLabel" Location="5,55"
        Text="Or enter RGB values for a custom colour:" Width="350"/>

    <wf:Label Name="RedLabel" Location="5,80" Text="Red"
        Width="45" ForeColor="Red"/>
    <wf:TextBox Name="RedBox" Location="55,80" Text="0"
        Width="45"/>
    <wf:TrackBar Name="RedSlider" Location="105,80"
        Width="256" Maximum="255" TickFrequency="8" LargeChange="32"/>

    <wf:Label Name="GreenLabel" Location="5,115" Text="Green"
        Width="45" ForeColor="Green"/>
    <wf:TextBox Name="GreenBox" Location="55,115" Text="0"
        Width="45"/>
    <wf:TrackBar Name="GreenSlider" Location="105,115"
        Width="256" Maximum="255" TickFrequency="8" LargeChange="32"/>
```

Figure 6.2. *XAML Colour Picker Source*

```
    <wf:Label Name="BlueLabel" Location="5,150" Text="Blue"
        Width="45" ForeColor="Blue"/>
    <wf:TextBox Name="BlueBox" Location="55,150" Text="0"
        Width="45"/>
    <wf:TrackBar Name="BlueSlider" Location="105,150"
        Width="256" Maximum="255" TickFrequency="8" LargeChange="32"/>

    <wf:Label Name="SampleLabel" Location="5,185" Text="Sample:"
        Width="100"/>
    <wf:PictureBox Name="Sample" Location="110,185" BackColor="Black"
        Width="250" Height="75"/>
    <wf:Label Name="ClosestLabel" Location="5,265"
        Text="Closest built-in colour:" Width="155"/>
    <wf:TextBox Name="SystemBox" Location="165,265"
        Text="This colour is exactly Black" Width="256"/>
    <wf:Button Name="Select" Location="5,290"
        Text="Hover here to preview colour..." Width="200"/>
  </wf:Form.Controls>
</wf:Form>
```

Figure 6.2. *XAML Colour Picker Source* (continued)

are drawbacks compared to Views 2.

As elegant as this declarative style is, having many attribute-value pairs for each tag can make the resulting XML difficult to read. Also many controls mean many tags, which further lowers the readability. Complicated programs are going to result in complicated, difficult to read XML. Since Views 2 splits property definitions over multiple XML blocks, each logical unit can be very concise. No control in this example requires more than two lines of attribute-value pairs, but it is easy to make a control take many more lines for all of its attributes. Even two lines is more than the Views 2 example ever requires in any one XML block. XAML is more readable than current code generated by the Visual Studio .NET visual designer, but XAML does not facilitate maintainability in the way that Views 2 does by allowing XML to be separated into easily readable sections.

The second point at which XAML proves to be inferior to Views 2 — which is related to the first in an obvious way — is that this monolithic structure does not take full advantage of the incremental nature of GUI design. For example the colour picker was designed in four distinct phases. For each of these four phases, one file was created, three XML and one CSS[1].

1. Outline which controls are going to exist in this GUI; name them; place them in the control hierarchy; and, if relevant, associate a layout manager with them.
2. Set the detailed properties of the GUI: text to appear on controls; items to appear in the lists; default values for input fields; etc.
3. Adjust the layout and formatting to achieve the desired GUI (This was primarily done in the CSS section).
4. Specify event handlers that are relevant for the application.

This basic incremental design certainly could — and would with the same designer — be done using XAML, but the designer would be incrementally adding to the same

[1]In chapter 5 the order of presentation was reversed for steps three and four to isolate the CSS after discussing the XML.

XML file, the result of which is shown above. Views 2's split style adds repetition of Control names at design time, but these duplicates can be generated quickly by making copies of the initial file, and this small amount of design time overhead leads to much better overall program structure, aiding maintainers later in the software's life. If this overhead seems too great, Views 2 can accept the entire specification as one monolithic XML input. For simple applications this may not reduce readability significantly and may be the preferred approach. There is nothing to be lost by using Views 2, because the splitting is optional. Making use of the split XML, each phase of the design can be revisited in isolation should something need to be changed. This an is obvious application of separation of concerns, an important principle of good software design.

The third, and possibly most important, benefit Views 2 offers over XAML is the provision of layout managers. XAML doest not have layout managers. As the previous example shows, all of the controls have to be positioned manually with the Location property. Views 2 supports layout managers that will arrange a set of controls in some logical way. A few basic layout managers are provided by Views 2. There is also the option to write a very simple class to create more. Using layout managers requires adding a few extra tags to the XML to logically group controls for the layout, which are a few `Panel` controls in this example. Adding these few extra tags is much easier than specifying locations manually. If a new control gets added in the middle of the Form, many others will need to be moved manually without layout managers.

There is one final point worth mentioning, which is how a programmer gets access to the objects created from the XML. In XAML, the user must use an interface called IPageConnector to get access to the objects that were created. IPageConnector has already been discussed in section 2.6.4. While using IPageConnector would not be difficult for an intermediate programmer, Views 2 simply offers a hashtable indexed by the controls' names, which is much more convenient to use.

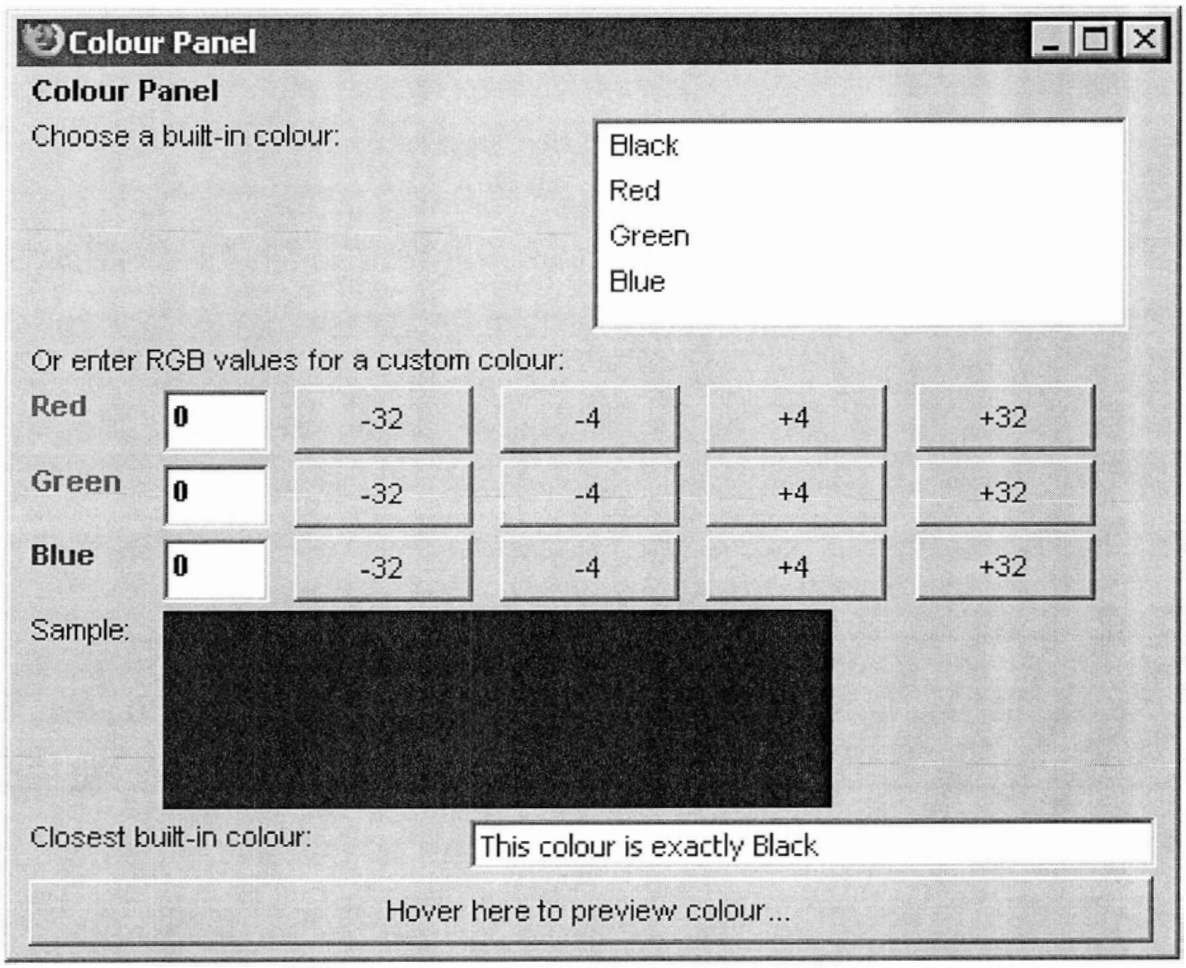

Figure 6.3. *The XUL version of the (reduced) Colour Picker.*

6.3 Views 2 compared to XUL

6.3.1 XUL's Colour Picker

Unlike XAML, Mozilla's XUL is already installed in full production form on many machines. Section 2.5 discussed that any installation of Netscape's current browser suite or any version of Mozilla Suite or Firefox include the rendering engine which is responsible for rendering XUL.

XUL's control set is not entirely compatible with Views 2 and XAML. However, the only major difference is that it does not have the `TrackBar` control which is used

for the red, green and blue sliders in the Views 2 and XAML versions. Instead we use sets of four buttons to adjust the current red, green or blue value. The end result is similar functionality with a few extra control tags, but the increase in complexity is small. Figure 6.3 shows the GUI that results from processing the XUL in figure 6.4.

Unlike XAML, XUL supports CSS and this application has a small CSS file, which is shown in figure 6.5. The CSS-variant currently used for Views 2 uses CSS syntax and selector semantics, but uses C# property names and values, just as they would be written in a Views 2 XML block. XUL follows the CSS standard so the CSS is slightly different from that presented for Views 2. There are also certain properties that have to be specified in the CSS, as there is no way to specify them in the XUL source even if they apply to only one element.

The next sections contain the entire XUL and CSS listing for this GUI. Like the XAML version, the event handlers are left out. Adding events to a XUL application is covered in section 2.5. The overall complexity increase is small: a `<script>` tag and an extra attribute-value pair for each event.

6.3.2 The Comparison

The basic format of the XML for XUL is similar to XAML. It follows the same style of tags named for control types with attribute-value pairs used to specify the properties, which is the standard style for these toolkits. Its control set is slightly different, as are some property names, but the basic format is similar to XAML. Therefore, several of the points made for XAML also apply to XUL, and they will not be repeated. We chose XUL for the comparison because it provides layout managers and the unusual feature of CSS, so this comparison will focus on these features.

Layout Managers XUL has simple layout managers, with the `<hbox>` and `<vbox>` tags. These two layout managers provide a great deal of utility for most programs. But they do not match up to the power and flexibility offered by Views

```
<?xml version="1.0"?>
<?xml-stylesheet href="colourpanel.css" type="text/css"?>

<window title="Colour Panel" orient="horizontal"
    xmlns=
    "http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul">
 <vbox>
  <label id="TitleLabel" value="Colour Panel" width="350"/>
  <hbox>
   <label id="ColourListLabel" value="Choose a built-in colour:"
     width="200"/>
   <listbox id="ColourList" width="200" height="80">
     <listitem label="Black"/>
     <listitem label="Red"/>
     <listitem label="Green"/>
     <listitem label="Blue"/>
   </listbox>
  </hbox>
  <label id="RGBValuesLabel" width="350"
     value="Or enter RGB values for a custom colour:"/>
  <hbox>
   <label id="RedLabel" class="RGBPanel" value="Red"/>
   <textbox id="RedBox" class="RGBPanel" value="0"/>
   <!-- Red TrackBar replacement buttons --> ...
   <button id="RedMinus32" label="-32"/>
   <button id="RedMinus4" label="-4"/>
   <button id="RedPlus4" label="+4"/>
   <button id="RedPlus32" label="+32"/>
  </hbox>
  <hbox>
   <label id="GreenLabel" class="RGBPanel" value="Green"/>
   <textbox id="GreenBox" class="RGBPanel" value="0"/>
   <!-- Green TrackBar replacement buttons --> ...
   <button id="GreenMinus32" label="-32"/>
   <button id="GreenMinus4" label="-4"/>
   <button id="GreenPlus4" label="+4"/>
   <button id="GreenPlus32" label="+32"/>
  </hbox>
```

Figure 6.4. *XUL Colour Picker Source*

```
  <hbox>
   <label id="BlueLabel" class="RGBPanel" value="Blue"/>
   <textbox id="BlueBox" class="RGBPanel" value="0"/>
   <!-- Blue TrackBar replacement buttons --> ...
   <button id="BlueMinus32" label="-32"/>
   <button id="BlueMinus4" label="-4"/>
   <button id="BluePlus4" label="+4"/>
   <button id="BluePlus32" label="+32"/>
  </hbox>
  <hbox>
    <label id="SampleLabel" value="Sample:"/>
    <label id="Sample" width="250" height="75" value="Black"/>
  </hbox>
  <hbox>
    <label id="ClosestLabel" value="Closest built-in colour:"
      width="155"/>
    <textbox id="SystemBox" value="This colour is exactly Black"
      width="256"/>
  </hbox>
  <button id="Select" label="Hover here to preview colour..."
    width="200"/>
 </vbox>
</window>
```

Figure 6.4. *XUL Colour Picker Source* (continued)

```
@import url(chrome://global/skin/);

.RGBPanel {
  font-family: sans-serif;
  font-size: 10;
  font-weight: bold;
  width: 40px;
}

#Sample {
  background-color: Black;
}

label, #ColourList {
  font-family: sans-serif;
  font-size: 10;
}

#TitleLabel {
  font-family: sans-serif;
  font-size: 12;
  font-weight: bold;
}

#RedLabel {
  color: Red;
}

#GreenLabel {
  color: Green;
}

#BlueLabel {
  color: Blue;
}
```

Figure 6.5. *XUL Colour Picker CSS Rules*

2's layout manager system. Views 2 provides a slightly expanded set of built-in layout managers, but the additional layout managers can be registered by name, which can then be included in the XML no differently from the built-in layout managers.

CSS Support XUL's support of the CSS standard is a strength it has over Views 2. Views 2 would benefit from supporting standard CSS. However, Views 2's style is beneficial as well. Because it uses the actual property names from the API, the name is consistent whether accessed through C# code, set in the Views 2 XML or set in the Views 2 CSS. One of the drawbacks of CSS is that in situations (like HTML) where there is a choice of where to set a particular property, the CSS name for that property may differ from the other system. This is not a problem with CSS; the CSS is simply conforming to its own standard. However, being able to use API names for things is beneficial as it keeps consistency in the system. This is also a benefit of Views 2 as an educational tool, since it allows designers to learn the API they are using, which was one of the design goals for Views 2. The final goal for Views 2's CSS support would be to keep this style as well as supporting the CSS standard as synonyms. Views 2's CSS-variant support has shown that it can be applied to this system, so a complete implementation of the CSS standard is not needed to demonstrate its usefulness. There is one major improvement to the CSS in Views 2: it is not mandatory. In XUL there are some properties that can not be set without the use of CSS. Views 2 encourages use of CSS for formatting concerns because it is so well suited to that use. However, it is entirely up to the designer how they want to separate their concerns and how much to use CSS. CSS is a powerful technology, very well suited to the task, but that does not mean it is always needed. Designers are free to make absolutely no use of CSS in Views 2 if it suits their program.

Chapter 7

Conclusions and Future Work

7.1 Future Work

7.1.1 Standard CSS Support

Views 2 makes use of CSS-based style specifications. In chapter 6, Views 2 was compared with XUL, which supports standard CSS. Views 2's CSS-variant has beneficial points as well, but CSS is a well defined standard, and adhering to the standard would be beneficial to the system. This would be a direct enhancement to the Views 2 code base.

7.1.2 Views 2 Visual Designer

One of the main attractions of a tool like Visual Studio .NET is the visual designer. Programmers may create GUIs by simple drag-and-drop interaction. Views 2 is designed to be readable and writable by hand, but that does not mean that a visual designer would not be beneficial. Work on visual designers has been done for the original Views system, so enhancing these systems to work under Views 2 would be a natural companion project to Views 2.

7.1.3 Views 2 XML Refactoring

Because Views 2 has such flexible XML input and there are so many alternative inputs that produce the same GUI, it would be a very interesting project to produce a system that refactors the XML. The system could separate or merge Views 2 XML according to some basic rules. This would be very useful if a system started out with one XML file, but grew rapidly to the point where some separation would be desirable for improved maintainability. This project would also fit well with the visual designer, since it would allow output from the visual designer to be separated into multiple XML files, instead of one large output.

7.1.4 New Problem Domains

Views 2 makes very powerful use of reflection to instantiate objects from the WinForms API based the XML declarations. But this general technique could be applied to other problem domains as well. In the most abstract sense, the XML is specifying the initial state of some system, and then reflection is used to instantiate the objects to represent this system in the appropriate state. Many systems follow this general concept of setting up an initial state and then performing some operations on that data. The operations do not need to be user interaction as they are in the Views 2 case. These ideas of reflective instantiation could be used almost anywhere XML is being used as an intermediate language for communication between two systems.

7.2 Conclusions

Toolkits for declaratively specifying GUIs with XML are widespread in modern GUI design. While there are many such toolkits available, most offer only minor differences from one another. Views 2 attempted to advance the state of the art for XML GUI toolkits. In addition to offering new ideas for professional GUI design, Views 2

attempts to make it easy for novice programmers to build up to industrial strength design. The predecessor, Views, was primarily intended for educational applications, although it still offers many benefits to professional designers. Views 2 attempts to build on these strengths without sacrificing usability for new programmers.

The resulting system is very flexible and extensible: offering basic modes of operation that are quite simple, but at the same time having a wide range of powerful features that advanced designers can make use of to express their designs in a powerful and maintainable way. In addition to providing support for good software design principles and a high level of flexibility to developers of GUI applications, Views 2 also serves as an example of new software design techniques. Much of the flexibility that allows Views 2 to serve such a wide range of developers extends from its use of reflection in its implementation.

This thesis covered background information on middleware and reflection (chapter 2), because to fully understand the impact of Views 2 it is necessary to think of it in a wider application than a GUI creation toolkit. Most of this thesis focused on GUI toolkits because that is what Views 2 primarily is, but the idea can be applied to other areas which, in some ways, is as important as the primary application. The background material also covered other XML GUI toolkits to give a basis for the application of Views 2's ideas.

This thesis continued on with a discussion of design goals and ideas for the Views 2 system (chapter 3), comparing it to the current systems and explaining why Views was used as a basis for a professional GUI design toolkit, when it was originally targeted to be primarily educational software. Because of the flexibility and power of reflection, Views 2 can maintain much of the novice-friendliness of Views while building up to be industrial strength. The reasons and rationale behind the design of the system were then laid out (chapter 4) to complete the description of how the Views 2 system works to offer powerful GUI design techniques while maintaining ease of use and understanding. Throughout the thesis the emphasis on good software

design and flexibility is highlighted.

The thesis closes with a look at a relatively complicated program in Views 2 (chapter 5) as well as some of its competitors (chapter 6) to evaluate how well Views 2 met its design goals. Views 2's explicitly stated design goals were to improve the state of GUI design and offer a toolkit that is powerful, flexible and easy to use even for novice designers. Although there is still room for improvement in some areas, particularly CSS support, the Views 2 system has met its design goals. However, in addition to the main theme of GUI design, there has been an underlying idea of how powerful reflection is for this application and middleware in general.

Through compelling use of reflection Views 2 offers a powerful GUI creation toolkit that improves the overall state of GUI design. It also offers an excellent example of just how powerful reflection can be for middleware systems.

Appendix A

Details of the XML Specification

A.1 XML Canonical Form

Views 2 accepts XML in a few different forms; it also accepts multiple XML fragments for one control and CSS specification of some properties. Information on some of these options are found in chapter 4. However, before being passed into the generator, these are all processed into one canonical form. The canonical form is outlined in this section.

It begins with the top level tag `<Form>`, which maps to a C# control that is used for the main frame of a windowed application. Every control has three types of children tags: properties, events and nested controls.

Property tags are tags that are named for some C# property of the control; it then has a child node that is the value to be assigned to that property. For example, `Text` is a property of `Form`:

```
<Form>
  <Text>Colour Panel</Text>
  ...
</Form>
```

Views 2 will detect what type the property has and will attempt to construct a value of that type from the value node. A basic text node will cover all primitive types as Views 2 will read it in as a C# string and perform the appropriate conversion. If

necessary the value node could be an arbitrary XML node used to specify another object. For example, `Image` is a property of `PictureBox` that has a non-primitive type, so more XML is needed to tell Views 2 how to create the appropriate type of value:

```
<PictureBox>
  <Image>
    <Image filename="apple.gif"/>
  </Image>
  ...
</PictureBox>
```

Event tags are tags named for some event that can be raised by this control. The format of event tags is the same as primitive typed properties, except that the string in the child node is interpreted as being the name of a method to handle this event. For example, the method `ButtonClicked()` is to be called when this button is clicked:

```
<Button>
  <Click>ButtonClicked</Click>
  ...
</Button>
```

The final type of children tag is nested control tags. In the .NET Framework every instance of `Control` (or one of its subclasses) has a property `Controls` which can contain other controls. This is suitable for controls such as `Form`, `Panel` and `GroupBox` but not for most others. However, the .NET Framework makes no distinction, so any control may have other controls nested in it. Views 2 supports this simply by allowing a control nested inside another control and it is up to the programmer designing the XML specification to decide if it is sensible or not. For example, the top level `<Form>` tag discussed above will probably contain other controls:

```
<Form>
```

```
  <Button> ... </Button>
  ...
</Form>
```

As is suggested by the previous examples, each nested control can have its own set of children tags of the three types. This introduces recursion into the generator and the specification. This follows very naturally from using XML as the specification format. Statically it could be difficult to tell the three types of nodes apart, but because Views 2 is generating all these objects at runtime with reflection the problem is much simpler. It is possible, in principle, for there to be a naming conflict between control names and properties or events of a control. For example, `Control` has a property called `Region`. If there were also a control class named Region, then a child of a control tag named `<Region>` would cause such a conflict. Views 2 would have no way of determining if the user intended to create a Region control as a child control or to set the property `Region` of the current control. There is, in fact, a class `Region` in the System.Drawing namespace, but because it does not inherit from `Control`, the child would have to be referring to the property. As a result, this does not actually cause a conflict. No such real conflicts have been discovered in the .NET Framework yet, but it possible. Views 2 is designed to be expandable to other APIs, so some way to resolve such conflicts is needed. If a conflict does occur the controls can be nested in a tag `<Controls>` named for the property of the `Control` class that contains nested tags. A conflicting name will bind to a property or event if it is not in a `Controls` tag and, of course, to the control if it is in such a tag [1]. Note that there does not *have* to be a conflict in order to use the `<Controls>` tag; it can be used at the designer's preference even if Views 2 could resolve all names without conflict.

There is also a small selection of tags used in Views 2 that are not drawn from elements of the .NET Framework: `layout-manager`, `class` and `primitive-array`.

[1] For more information on why conflicts are resolved this way see the details of the generator algorithm in section A.3.

The property `layout-manager` is used to add a layout manager to a control (see section 5.2 for more information on layout managers and how this property is used). The property `class` can also be used; class is a reserved word in C#, but it is used in the XML for CSS classes (see section 5.5 for more information on this) and is not related to the C# meaning. There is also a tag `<primitive-array>` that is used to specify an array of primitives. As mentioned previously, all primitives are captured by a single string that is converted properly; an array of primitives is then generated by splitting a string into substrings and converting them into the elements of an array. A `<primitive-array>` has two properties: `separator` and `values`. An array is then constructed in a straightforward manner by splitting the field 'values' at every occurrence of the character specified by 'separator'. These strings can be converted into whatever primitive type is needed (which can be determined by reflection). These tags are needed by the Views 2 system but do not correspond to .NET Framework objects or properties. Most of the Views 2 control tags can be recognized by the presence of a hyphen in their names. This character can not be part of a valid C# identifier, but it can be part of an XML tag name. The exception to this rule is the `class` property; this is not a valid C# identifier because it is a reserved word.

This canonical form of the XML allows full specification of the GUIs to be created by Views 2. It begins with a top level `<Form>` tag to produce the top level `Form` control required for a windowed application. Then children, as described above, are recursively added to the XML until the entire GUI has been specified. The full range of XML and CSS input formats that are accepted by Views 2 to support greater user expressivity have been described previously in section 4.3.1, but these formats are all merged together to form a final XML specification of this format (with no CSS — the back end of Views 2 has no CSS processing capabilities).

A.2 XML Shortcuts for Ease of Use

Views 2 supports a much wider variety of input XML than the canonical form for the back end generator. The main purpose of Views 2 is to allow designers to express their design in an easy to read, easy to write and highly maintainable fashion. In addition to the features of the XML described in section 4.3.1 and the use of CSS to specify some properties of the controls, Views 2 allows some short cuts to be taken with the XML.

The first shortcut allowed is to specify properties and events as attribute-value pairs. It has been stated that this is not the primary model Views 2 uses for its processing of properties and events, however, it can accept that as input, as has been alluded to earlier. As was discussed in the previous section, most property values are simple strings that are converted into a primitive type. Likewise, events are specified by a string that is the name of the method to handle that event. In these simple cases where the value is just a string, it may be written as an attribute of the control node rather than a child node. For example:

```
<Button Click="ButtonClicked"/>
```

and

```
<Button>
  <Click>ButtonClicked</Click>
</Button>
```

are considered identical in the Views 2 system. Views 2 will translate the former into the latter so that the XML being passed to the generator is in the canonical form specified in section A.1. But as long as the value of the property or event may be contained legitimately as the value of an attribute-value pair, the programmer may use whichever form is preferred.

When using this format to specify properties and events, the double quotes may be replaced by single quotes around the values of any attribute-value pairs. If there

is no space in the value, the quotes can be omitted entirely; the value is delimited on the left by the = sign and on the right by whitespace or the end of the tag. This short cut is implemented because if the XML is included in a C# string constant, the double quote character is not legal unless it is escaped. If the designer greatly prefers the attribute-value pair style, then there could potentially be a large number of double quote characters being used, leading to a large number of escape characters. This can be avoided by making use of the single quote character at the programmer's preference. This problem can also be avoided by not using C# string constants and instead having the XML read from a file. The XML in a file can still use these short-cuts, but it has no need to escape double quotes if those are used for delimiting values.

The canonical XML form chosen is very flexible and chosen for its technical merits; however, sometimes the notation can become quite long with repetition in open and close tags for the same property that just have a single value between them. Many XML based GUI specification languages use attribute-value pairs. Views 2 has not gone this route for various reasons that have been discussed previously. There are, however, benefits to this style. As the example above shows, even for a single event the canonical form is much longer. If there were two or three events and several properties, it could make for a big block of very repetitive XML code that makes the meaning less clear relative to the attribute-value pair style. The purpose of Views 2 is to encourage expressivity and good design for GUI systems, so by accepting the attribute-value pair style where it is sufficiently flexible for the value, Views 2 allows designers the full flexibility of the child node format, without forcing that more verbose syntax on designers when it is not needed.

A.3 How Reflection is used to turn XML into C# Objects

The core of Views 2's logic lies in the generator: the class `VGenerator`. It has a primary public method `Generate()` that should be called with the Document Element (i.e. root) node of the XML tree. This will then generate and return the top level object (a Form), having recursively generated everything that should be inside it. Note that the top level should have already been constructed (as a `Views2.Form` specifically) which will have been passed into the `VGenerator` when it was constructed, so the top level call will actually just return the same `Views2.Form` that called it. This is a side effect of the fact that the generator works recursively and instantiates all the controls except the top-level `Views2.Form`.

The general algorithm of the generator is as follows:

1. Construct an object of the type given by the XML tag (or look it up in the controls table if this is a second or subsequent XML fragment for this control).
2. Iterate through the children of this XML node:
    - Check if this node represents a property, event or control:
        (a) Property: check for a special case given this type, property and the name of this node's first (should be only) child. If it exists, execute that special case; otherwise, attempt to assign the value constructed from this node's child to this property using a normal set method.
        (b) Event: register the event with the `VEventDispatcher`; the method name to execute is in a text node as this node's only child.
        (c) Control: a new control should be nested in this control. Recursively generate it and add it to the current control's contained controls set.

This algorithm seems surprisingly simple, and in fact in many ways it is very simple, because reflection is so powerful. Here are the specific details on each step:

1: First perform a simple check to determine if the current XML node is indicating that a form control should be created. If so, there should already be a `Views2.Form` object that was passed in; assign that form object to the result. Otherwise, check the controls that have already been created for one with the same name (retrieved from the `<Name>` child tag) and retrieve it, assigning the retrieved object to the result. If it does not exist, use reflection to find a Type named by the XML tag, then use reflection to invoke the default constructor and assign the instantiated object to the result. If invocation of the default constructor fails, check for a special case on the given type and child node to see if an object can be created that way. If this also fails, report an error.

2: Using reflection, check if the type of the result object has a property named for the current XML node. If it does then move to step 2a. Otherwise check for an event named for the current XML node, if one exists then move to step 2b. Finally if neither of these checks succeeded check for a type named for the current XML node, and if there is such a type move to step 2c. A particular name can not be valid for both a property and an event, but it could be valid for one of those and a control. If a conflict occurs the order these checks are done in implies a priority ordering of which path should be taken. Controls are the lowest priority so if there is a conflict, the property or event (that can not conflict with each other) will be used for a tag with the name in question. Controls can — as noted in section A.1 — be contained within a tag `<Controls>` to get a control with the conflicting name.

2a: The reflection check indicated there is a property of the relevant name. Check for a special case on this property; if there is one, build the argument from the XML subtree rooted on this node's child and apply the special case handler. Otherwise use reflection to check for a set method on this property, and if there is one, build the argument from the XML subtree rooted on this node's child and call the set method. Otherwise, report an error.

2b: The reflection check indicated there is an event of the relevant name. Check

for the presence of an event dispatcher (the event dispatcher is an optional argument to the `Views2.Form` constructor) and call the `Register()` method with the current object, the reflection `EventInfo` object and the name of the method from the child node (which is assumed to be plain text). If the method name does not exist (which should be the case if it is malformed) or if the event dispatcher is not defined report an error.

2c: The reflection check indicated there is a control of the relevant name. Recursively call the generator function and add the result to the `Controls` property of the current result `Control` object.

Appendix B

Glossary of Terms

Cascading Style Sheets (CSS) CSS, most commonly used with HTML web pages, is a mechanism for adding style to documents. CSS allows fonts, colours and other formatting and style information to be separated from the information content and recombined for display.

Common Intermediate Language (CIL) See Microsoft Intermediate Language (MSIL).

Common Language Infrastructure (CLI) The infrastructure that supports creation and execution of MSIL bytecode executables, which is the output of most C# compilers. Microsoft's .NET Framework is the primary implementation of the CLI, but there are others including Mono and Rotor.

Control A control is the simplest component of a GUI. A single button, label, textbox or other simple GUI component is a control. The term widget is often used in general for these components. The term comes from the class `Control` in the WinForms API that is an ancestor class of all of these simple components.

Delegates Delegates are a C# language feature. Informally they are similar to function pointers from C++, except that they are typed and type-safe (and a single execution of a delegate may execute multiple methods). Formally they are a mechanism for treating functions as first-class objects. They allow a method to be invoked without having an explicit reference to the receiver, and are the usual way to implement event handlers in C#.

ECMA International ECMA International is a information, communications and electronics standards association. C# has been submitted and approved as an ECMA standard.

Extensible Markup Language (XML) XML is a general-purpose markup language for documents and data containing structured information. It combines both content and structure into a single textual document using markers (called tags) to indicate structure.

Dynamic Link Library (DLL) A DLL provides functions or data that can be used by Windows applications. Many Win32 APIs are published as DLLs, as it is an appropriate format for non-executable compiled code. Programs dynamically create links to the functions or data in these libraries as needed.

Graphical User Interface (GUI) A GUI is the visible portion of a program that the user interacts with. A GUI is used for both user input and output of graphical programs.

Microsoft Intermediate Language (MSIL) MSIL is Microsoft's bytecode language that is executed by the .NET Framework as well as any other implementation of the CLI. C#, as well as many other languages, can be compiled to MSIL rather than a platform specific machine language and executed by an implementation of the CLI. Also know as Common Intermediate Language (CIL).

Mono A third-party implementation of the CLI. Unlike Rotor, Mono has a complete implementation of the APIs included in the Win32 .NET Framework. Mono is primarily Linux based, but there are versions available for Windows, Mac OS X, BSD and Solaris.

Layout manager Found in some GUI design toolkits, a layout manager is a function that sets the locations (and possibly sizes) of a group of controls in a GUI. XUL's `<vbox>` and `<hbox>` tags are an example of simple layout managers for vertical

and horizontal lists respectively.

Panel Panel is a special type of control used for logically grouping other controls in WinForms. A panel has no visible presence in the final GUI, its only task is to group other controls. In Views 2 a panel will often be associated with a layout manager. All of the controls in one panel will be the group of controls that a particular layout manager is acting on.

Reflection Reflection is a meta-programming technique. There are different degrees of reflection supported by different platforms, but it can allow inspection, invocation of methods, or altering structure of running systems. In some cases reflection can allow entirely new executable sections of code to be generated and dynamically attached to a running system.

Rotor Microsoft's platform-independent implementation of the CLI. Unlike Mono, Rotor does not have implementations of certain classes in the Win32 .NET Framework, most notably, the WinForms API. This product is officially called the Shared Source Common Language Infrastructure (SSCLI), but is often referred to as Rotor, which was the project name at Microsoft.

Shared Source Common Language Infrastructure (SSCLI) See Rotor.

WinForms API The WinForms API is the primary API for the creation of GUI programs on the Windows platforms. This is a shortened name for the API implemented in the System.Windows.Forms namespace included with the Win32 .NET Framework as well as Mono.

Bibliography

[1] Judith Bishop and Nigel Horspool. Developing principles of GUI programming using Views. In *Proceedings of SIGCSE'04*, pages 373–377, Norfolk, VA, March 2004. ACM Press.

[2] Judith Bishop and R Nigel Horspool. *C# Concisely.* Pearson Education, 2003.

[3] Object Management Group, Inc. *CORBA Component Model, v3.0,* June 2002. Avaiable online at http://www.omg.org/technology/documents/formal/components.htm.

[4] Pierre Cointe. Reflective languages and metalevel architectures. *ACM Computing Surveys*, 28(4), December 1996.

[5] ECMA International. *Standard ECMA-335 Common Language Infrastructure (CLI)*, 2nd edition, December 2002. Available online at http://www.ecma-international.org/publications/standards/Ecma-335.htm.

[6] Edd Dumbill and Niel M. Bornstein. *Mono: A Developer's Notebook.* Developer's Notebooks. O'Reilly & Associates, 1st edition, July 2004.

[7] Steve Vinoski. Where is middleware. *IEEE Internet Computing*, 6(2):83–85, March 2002.

[8] Fabio Kon, Fabio Costa, Gordon Blair, and Roy H Campbell. The case for reflective middleware. *Communications of the ACM*, 45(6):33–38, June 2002.

[9] J. Ferber. Computational reflection in class based object-oriented languages. In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, pages 317–326, New Orleans, Louisiana, 1989. ACM Press.

[10] Judith Bishop, R Nigel Horspool, and Basil Worrall. Experience in integrating Java with C# and .NET. *Concurrency and Computation: Practice and Experience*, 17:663–680, June 2005.

[11] Judith Bishop and Basil Worrall. Towards platform interoperability: Retargeting a GUI library on .NET. In *3rd Conf .NET Technologies*, Plzen, Czech Republic, May 2005.

[12] Neil Deakin. XUL tutorial. http://www.xulplanet.com/tutorials/xultu/, 2005. URL last accessed April 30, 2005.

[13] Mozilla layout engine. http://www.mozilla.org/newlayout/, September 2004. URL last accessed May 12, 2005.

[14] Lori MacVittie. Conjure up XUL for browser flexibility. *Network Computing*, 12(5):110–114, March 2001. Freely available online at http://www.nwc.com/1205/1205ws4.html.

[15] Ben Goodger, Ian Hickson, David Hyatt, and Chris Waterson. XML user interface language (XUL) 1.0. http://www.mozilla.org/projects/xul/xul.html. URL last accessed April 30, 2005 (site not complete).

[16] Alex Hildyard. Learn to write XAML applications today with Xamlon. http://www.devx.com/dotnet/Article/22341, November 2004. URL last accessed 2005-02-17.

[17] Amy Fowler. JDesktop Network Components. http://javadesktop.org/articles/JDNC2/index.html, June 2004. URL last accessed 2005-04-30.

[18] GNUe forms. http://www.gnuenterprise.org/tools/forms.php, 2003.

[19] Bindows. http://www.bindows.net/, 2005.

[20] Thierry Machicoane. http://www.chez.com/mustcorp/smawl/. http://www.chez.com/mustcorp/smawl/, 2003. URL last accessed May 19, 2005.