Cross-Platform Development: Software that Lasts

Judith Bishop, University of Pretoria

Nigel Horspool, University of Victoria

The design of software that is easy to port or deliberately targeted for multiple platforms is a neglected area of software engineering. A promising solution is to link components and toolkits through XML and reflection.

As the Greek philosopher Heraclitus purportedly said, "The only constant is change." This famous saying should be engraved on software engineering's temple walls because its followers know only too well that requirements are fluid and that functions ebb and flow with the user tide. The whole point of the practice is to design a system that will withstand unpredictability.

But the process of handling change requires answering some hard questions: What happens to the software when the new hardware arrives? And, even more challenging, how does a small software house develop for a variety of platforms, thus maximizing its customer base and profit? We believe the answer to both these questions is to build a high degree of platform independence into the software and thus alleviate the problems of creating multiple versions and porting to new platforms.

DEFINING THE PROBLEM

This is a software engineering problem, but not a well-known one. One of the more compelling definitions of software engineering is, "the multiperson construction of multiversion software."[1] The popular view of software engineering focuses on the first part of this definition—managing teams to produce a large product. Most people subscribe to this view because they see software engineering as a discipline that tackles computing in the large, elevating tools and techniques from the level of craft to a position that lets developers harness them efficiently and reproducibly for the successful completion of large projects. But just as important is the view inherent in the second part of the definition—identifying specific parts of a product so that experts can design them and organizations can mass-produce them free of language and environment dependencies.

Component-based software engineering has made tremendous strides toward satisfying both parts of the definition. The emergence of Web services as a way of delivering up-to-date functionality over the Internet is now accepted as the norm, and major software suppliers seem to be establishing the practice of automatically sending upgrades to any machine using their software. From a user's perspective at least, software engineers might have finally got it right: Software to support all walks of life—from business to entertainment, education to research—is ubiquitous, up-to-date, and mostly reliable.

Perhaps most striking is the larger than ever choice of serious platform options that are increasingly compatible. As the definitions in the "A Platform by Any Name" sidebar imply, the lines between hardware and software are blurring—so much so that investment in new hardware no longer necessarily requires buying and installing new software.

Unfortunately, the same sense of having arrived is not true for the developers and users of in-house, customized, or research-based software. Faced with all these choices, if anything, their task has become harder. Although maintenance and upgrading still account for more than

Published by the IEEE Computer Society
0018-9162/06/$20.00 © 2006 IEEE

two thirds of a project's lifetime cost,[2] developers of these narrower applications seem unmoved to address the issues. More often than not, the product's final delivery also marks the end of its developers' involvement. From this point, the software's owners and users must either work within the confines of the software as is or employ new developers to adapt it.

We see no reason for developers of any type to be victimized in this way. Through the use of middleware constructed using reflection and controlled through XML specifications, it is possible to give the components in a large software system a high degree of platform and even language independence. The result is long-lived software that can migrate gracefully as platforms improve and change. For small developers, this would mean change that does not have to be onerous and costly.

Our solution particularly benefits developers of graphical user interfaces, since GUIs are a pervasive and critical part of most software and are well represented with cross-platform toolkits and libraries. Indeed, we tested our approach on an application in which one component is a GUI library. Making such components platform independent has challenged cross-platform software development for many years. We believe our solution will meet this challenge and that it can address problems in other domains with platform-dependent system components, such as mobile devices.

TYPES OF CHANGE

Understanding how to manage change and adapt software requires first understanding how platform, functional, and nonfunctional changes differ:

- *Functional* changes add facilities to the software, retaining the existing platform base and performance.
- *Nonfunctional* changes address performance issues—better throughput, response times, security, power consumption—but assume the same platform base.
- *Platform* changes require moving the software to new or additional languages, operating systems, hardware, or devices.

Our focus is on the third change category, particularly on a system's low-level, platform-dependent parts. An organization invests in developing software that is based on a given platform, and then finds that it cannot migrate to a new platform because it relies on features that are no longer available. If many parts of the software rely on these features, as is true of graphics or networking, for example, then any migration approach is desirable, given that the organization otherwise faces complete reimplementation.

Functional and nonfunctional changes respond to the outside world—society, trends, laws—as Figure 1 shows. The system's owners and users have a say in the timing and extent of such changes, judging which are essential and profitable. Conversely, platform changes come about mainly because of technical advances—hardware, operating systems (OSs), networking—and are not easy to ignore or postpone.

Changing the OS in a major way can cause parts of the system to fail. As the figure suggests, the resulting discontinuity between software and platform is much more dramatic. Similarly, a hardware change will inevitably affect some parts of a system.

A Platform by Any Name

As these definitions show, the line between hardware and software platforms is rapidly becoming harder to discern.

platform. Although not strictly defined, the term generally constrains language, operating system, computer, or some combination. Platform examples include

- Java 2: Language and set of libraries that run on most operating systems (OSs) and computers.
- Windows: An OS that runs almost exclusively on Intel processors and supports many languages.
- Intel: Processor that can support many OSs, including Windows, Linux, and Mac OS X.

cross-platform or multiplatform. Software that exists in different versions so that it is available on more than one platform (in the same dimension). Examples are

- Microsoft Office or iTunes, which are available for both Windows and Mac OS X.
- Mac OS X running on Intel and Motorola platforms.
- Common Object Request Broker Architecture (CORBA) implementations for Java, Ada, C++, Python, and many other language platforms.

platform-independent. Software regarded as having few or no platform dependencies, but that is actually multiplatform. Describing Java as platform-independent means that it runs on many platforms.

portable or retargetable. Software with qualities that would enable all or most of it to be moved to another platform. Similar to "platform-independent."

port, retarget. To move software to a different platform, rewriting parts as necessary. Software designed in layers is generally easier to retarget.

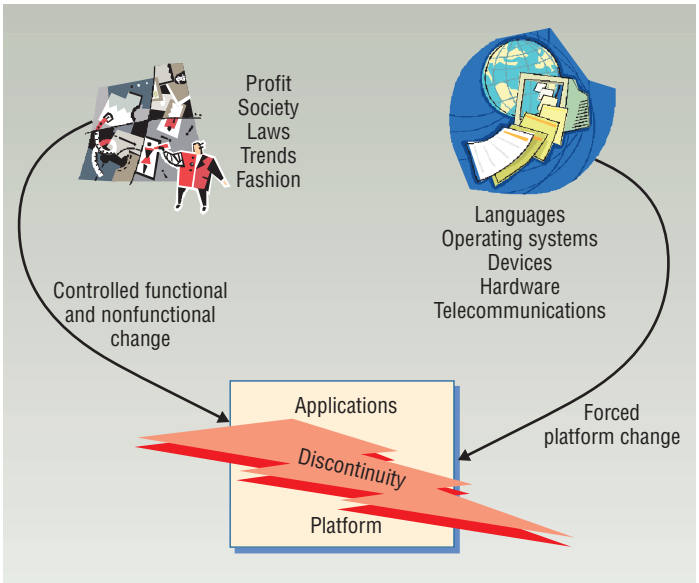


Figure 1. Causes of software and platform change. Functional and nonfunctional changes respond to the outside world, but platform changes are usually mandated and hard to ignore or postpone. Consequently, time pressures to move the change along create a discontinuity that is hard for small developers to manage.

It is the stress induced by platform change that we seek to neutralize or, at least, control. In response to such stresses, an organization could regard the crisis as an opportunity to completely redevelop the software's platform-dependent parts, perhaps exploiting the opportunity to refine them. Regrettably, this requires developers who know both the software and the new platform, a skill combination in short supply in most organizations.

Moreover, retuning and adapting the software is no guarantee that it will live longer because with the next platform change comes a fresh change cycle. In our approach, the idea is to anticipate and build for platform changes from the beginning.

EXPLOITING DEVELOPMENT ADVANCES

Our approach brings together key software development advances over the past decade and provides a way to address the migration of overlooked platform-dependent parts, such as GUI libraries. Figure 2 shows some of the dramatic changes in the software development landscape over the past five years, including

- advances in Sun's Java Internet-based programming platform (begun in 1996);
- Microsoft's .NET with its common language runtime support of many languages, as well as the shared-source Rotor and open-source Mono versions;
- the rise of Linux as a popular Windows OS rival;
- increased use of Eclipse as a Java-based cross-platform development tool, with IBM support;
- Apple's initiatives to enable multiple OSs (Unix then Windows) on its Macintosh computers; and
- a complete upgrade in the Windows platform to Vista, including a new virtual machine (.NET Framework 3.0) and a new GUI implemented in the Windows Foundation Classes.

These advances alone do not guarantee that software can survive a platform change. For example, if an organization is developing GUI-based software using Windows and wants to port it to Mac OS X, the experiment will most likely fail since none of the Windows .NET platforms on the Mac support the Windows GUI application programming interface (API). The shared-source version of .NET (distributed as Rotor), for example, specifically excludes the System.Windows.Forms library.[3] Thus, a major change in the Windows platform will mean that much software will face migration to take advantage of Windows Vista's new features.

Our solution is to exploit certain software development advances to build a new era of software that can last—even in the face of numerous platform changes. These advancements include APIs, cross-platform toolkits, reflection, virtual machines (VMs), and the Extensible Markup Language (XML).

Application programming interfaces

The key to managing complexity has always been abstraction, the oldest form being the subroutine. Coherent sets of subroutines, or classes when object-oriented, are further grouped into APIs—also known as libraries or foundation classes. They are now a very large part of any language, almost dwarfing it in sheer number of terms and functions. For example, the API for Java 2 comprises 166 packages, while the API for .NET 2.0 is organized as 195 namespaces. The APIs are very similar and cover features that range from accessibility to imaging, cryptology,

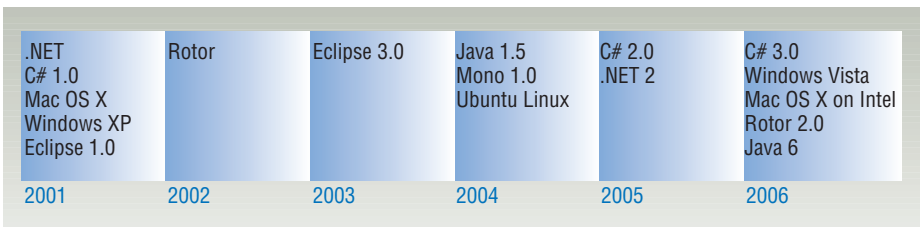


Figure 2. Platform development since 2001. Platforms like Java and .NET increase the possibility for cross-platform and multilanguage software.

logging, mail, printing, and security to zip files.

Although APIs provide immense functionality, having so many of them presents a real challenge in identifying the right tool for the task and keeping up with modifications. The first GUI API provided for Java in 1996 was the Abstract Windowing Tool, which went through two revisions. In 2000, Swing replaced AWT, bringing some quite different elements. The corresponding API in Windows is System.Windows.Forms, which has remained stable since 2001 with the major Windows language platforms (C/C++, C#, Visual Basic, and so on). With Windows Vista, however, the Windows Presentation Foundation API will replace System.Windows.Forms. Even if WPF is upwardly compatible with System.Windows.Forms, its new features will affect existing software.

Ultimately, the API provides the interface to low-level functionality, and if the API itself is not available on the new platform, the only current choice is redevelopment.

Cross-platform toolkits

Cross-platform toolkits are integral to GUI development. Like APIs, they provide a set of classes that define the widgets (or controls or components) that developers can create and manipulate in developing an event-driven GUI. But unlike APIs, they define the widget set independently of any platform and then provide multiple implementations for many widely used platforms. (In this context, "platform" can be a language or OS.)

Because of its platform-independent qualities, a toolkit raises the odds of successful migration to a new platform. However, there are two caveats: First, the original software does not use the native API, which might make programming awkward. Second, the GUI's look and feel might differ from that of an application that uses the native GUI facilities.

The language the toolkit relies on is also a consideration. Two popular toolkits, Trolltech's Qt and Gnome's Gtk+, rely on C/C++, for example. Using them with other languages requires creating additional code and interfaces to perform various mappings and bindings and could require enclosing many classes in wrapper code. All this can be daunting to a maintenance programmer. Both Qt and Gtk+ define their own sets of widgets, which are then platform independent. Unfortunately, they are incompatible with each other, as well as with the native widget sets that the language API provides.

Reflection

Reflection is a program's ability to observe, and perhaps alter, its own structure. In the context of mainstream object-oriented programming languages, reflection is the ability to collect complete information about an object from the metadata associated with it and then to use these fields, properties, and methods in allowable ways.

For example, a program might use reflection to obtain a list of the fields and methods that an object possesses and to obtain the fields' data types and methods' signatures. Once a program ascertains that a field or method exists, it might use reflection to access that field's value or to invoke a method.

```
using System.Reflection;

public class Reflect {
 public int x;
 public bool b;
 public void Print (object obj) {
    Type t = obj.GetType();
    Console.WriteLine
      ("Object has type {0}", t.Name);
    if (t.IsClass) {
    Console.WriteLine
      ("Fields and values are");
      foreach(FieldInfo fi in t.GetFields()) {
        Console.WriteLine ("  {0} = {1}",
        fi.Name, fi.GetValue(obj));
      }
    }
  }
  static void Main() {
    new Reflect().Print(new Reflect());
 }
}
```

(a)

```
Object has type Reflect
Fields and values are
 x = 0
 b = False
```

(b)

Figure 3. Reflection. The example shows the use of reflection in C# for a general print function that displays the names and values of all fields in an object. Without knowing any details of the classes declared in the program, the Print method (a) will display the names and values of any fields declared as public in any instance of any class. In this example, it is working on itself with corresponding output (b).

Consider, for example, a completely general print function that displays the names and values of all fields in an object (a class instance). With reflection, this function need not include the names and types of all the fields of all classes in the program even if the programmer knows all this information in advance. As Figure 3 shows, reflection permits a much more concise and elegant solution.

In general, reflection lets the programmer defer implementation decisions to runtime and thus provides a new form of program abstraction. It is extremely powerful when used in conjunction with APIs and toolkits.

Virtual machines

VMs provide an abstraction level just above the OS layer. A VM implements a machine architecture that is both computer and OS independent. To implement a

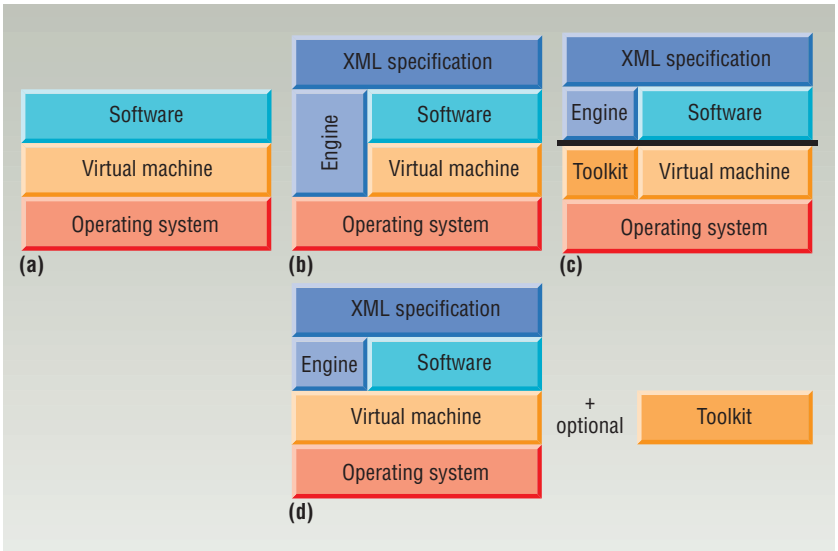


Figure 4. Development of a GUI that relies on platforms at different levels. (a) Normal software, (b) language-independent software with engine-based access to libraries, (c) platform-independent software with cross-platform toolkits, and (d) platform-independent software with reflective libraries or toolkits.

programming language, a compiler translates a program written in that language to the VM's instruction set. Using a VM has several important advantages:

- A compiled program distributed as instructions for the VM is independent of all platforms that run the VM.
- The VM controls the program's access to files and other system resources, thus providing a high level of security.
- Using just-in-time technology, the VM can dynamically translate the program's VM instructions into the host computer's native instructions. With direct access to the host computer's resources, performance is much higher than if the host computer interpretively executes VM instructions.

Two widely used VMs are the Java VM and the .NET Common Language Runtime. JVM executes Java byte code that Java compilers generate. The CLR executes Common Intermediate Language instructions generated from any .NET language. A VM can enable code written in different languages to interact. For example, the C#, VB, and C/C++ compilers provided with .NET generate CIL code. An important feature of .NET is that multiple CIL modules from different compilers can be combined in a single program. In this role, the CLR can help lengthen the software's life in the face of new language development.

When the VM has been implemented on many platforms, as is the case with Java (and to a much lesser extent with .NET), compiled programs are essentially platform independent. Java applets fully exploit this platform independence, enabling Web browsers that run on almost any platform to download and execute them.

Extensible Markup Language

XML is a platform-independent notation for representing structured data. Both XML and the ubiquitous Hypertext Markup Language (HTML) notation derive from the Standard Generalized Markup Language (SGML). As its name implies, XML is extensible through the addition of tags, and programmers have developed standard tag sets for an extraordinarily diverse set of applications, including spreadsheet data, calendars, mathematical equations, stock market data, and geographical mapping.

Although HTML notation looks similar to XML, it breaks many XML rules. A newer notation that *does* follow the rules, XHTML, could one day replace HTML.

XML's platform independence and the availability of standard tools on all platforms for manipulating XML files make the language an excellent format for data exchange. Even if two programs run on different platforms, they can exchange data in XML form as long as they agree on the set of tags in use. Thus, using XML as the format for external data will lengthen the software system's life.

DEVELOPMENT FOR LONG LIFE

Figure 4 shows the stages of our method for developing a GUI that relies on platforms at different levels.

The code to create and manipulate a GUI is embedded in the software, which is written in a language, *L*. The calls (to create, lay out, and handle controls) are methods in the toolkit. The toolkit can be in a language other than *L*, but it will be compiled to the same VM. A direct link between the toolkit and the underlying OS helps render controls on output devices and event handling. The objective is to increase the system's overall portability.

Language independence

Our method starts by introducing a language-independent GUI specification notation to specify the toolkit's function—in short, quantifying the programmer's interface. The theory underlying this step is based on declarative user interface models,[4] which describe user interfaces as the controls displayed to the user, including their composition and layout.

XML is now widely accepted as the preferred notation for these models: Instead of using the usual object constructions for a label, textbox, and button,

```
Label request = new Label();
request.Text = "Password";
Textbox reply = new Textbox();
Button activate = new Button();
activate.Text = "Submit";
```

programmers write an XML specification

```
<Label text="Password"/>
<Textbox name="reply"/>
<Button name="activate" text="Submit"/>
```

The strategy of using separate specifications for user interfaces appears to have originated with the Extensible Virtual Toolkit [5] and has since become popular, with Eclipse's Extensible User Interface Language (XUL) being among the main proponents. The "XML-Based Specifications for a Graphical User Interface" sidebar describes three such systems. Figure 5 shows a GUI specification for a small calculator in Views, an XML-based system for GUI specification that we developed.[6]

The handlers are in the code and refer to the methods in Views to access the controls mentioned in XML, as Figure 5b shows. For example, the method call `form.PutText("eurobox","0")` would display the string "0" in the TextBox control whose name is "eurobox"and which has the label "Paid on hols." Programmers can also use the more familiar callback notation, although when we designed the system, we felt this was less desirable. The following assignment therefore can perform the same write to the textbox:

```
form["eurobox"].Text ="0";
```

where, by overloading the indexing operator, `form` provides access to the controls in the GUI. Within the Views engine, the use of implicit type conversions avoids the need for type casting.

As the sidebar describes, XUL, the Extensible Application Markup Language (XAML), and Views achieve the goal of reusable GUI specifications. The XML design in each case is somewhat tied to the expected underlying library, but there is considerable commonality among widget names and behaviors.

The big difference between the two commercial notations and Views is that both XUL and XAML let programmers—but do not compel them to—embed event-handling code (for XUL, JavaScript, and, for XAML, any .NET language) within the user interface declaration. The Views model, on the other hand, provides an engine that intercedes on the GUI's behalf to signal events to the host application.

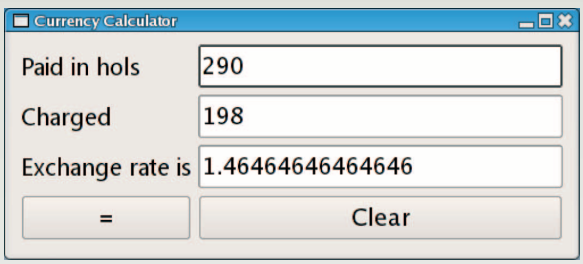
The engine's implementation language is irrelevant to the Views user, who can be programming in VB, C++, and so on. Programmers can exploit this separation of notation and toolkit to maximize the cross-platform qualities of GUIs.

```
 // XML Specification
<Form text="Currency calculator">
  <horizontal>
    <vertical>
      <Label text="Paid on hols"/>
      <Label text="Charged"/>
      <Label text="Exchange rate is"/>
      <Button name="equals" text="="/>
    </vertical>
    <vertical>
      <Textbox name="eurobox"/>
      <Textbox name="GBPbox"/>
      <Textbox name="ratebox"/>
      <Button name="clear" text="Reset"/>
    </vertical>
  </horizontal>
</Form>
```

(a)

```
// C# code to interact with the GUI form
for( ; ; ) {
  double euro = 1.0;
  double GBP = 1.0;
  string s = form.GetControl();
  if (s == null) break;
  switch (s) {
     case "clear":
       euro = GBP = 1.0;
         form.PutText
          ("eurobox",euro.ToString("f"));
       form.PutText
        ("GBPbox",GBP.ToString("f"));
       break;
     case "equals":
       euro = double.Parse
         (form.GetText("eurobox"));
       GBP = double.Parse
         (form.GetText("GBPbox"));
       form.PutText("ratebox",
         (euro/GBP).ToString("f"));
       break;
   }
}
```

(b)



(c)

Figure 5. XML specification. (a) GUI specification in XML for a simple currency calculator, (b) the C# code that uses the form, and (c) the result.

Platform independence

In Figures 4a and 4b, the more difficult and less traveled step is to decouple the software from specific knowl-

XML-Based Specifications for a Graphical User Interface

XML-based GUIs need both a specification and a set of handlers.

XUL : XML User-Interface Language

XUL is the XML-based GUI specification model that the Mozilla browser family uses. It has a rich notation for creating widgets, and uses box, grid, and other layouts. Part of an XUL specification for a simple currency calculator is

```
<script src="calculator.js"/>
<grid>
<rows>
    <row>
    <label id="11" class="small"
      value="Paid on hols"/>
    <textbox id="eurobox"/>
  </row>
</rows>
</grid>
```

By virtue of being embedded in HTML code, XUL code is inherently cross-platform. The scripts for the handlers are not in the program, however, but are in JavaScript in a separate file, for example:

```
function clear() {
document.getElementById("eurobox").
   value=1.00;
document.getElementById("GBPbox").
   value=1.00;
}
```

This breaks the separation of concerns between the GUI and the computational logic, which should be in the program and in the program's language. XUL's sister language, XML Binding Language (XBL), allows additional widget customization.

XAML: Extensible Application Markup Language

XAML is the declarative markup language that Microsoft introduced in Version 2 of the .NET Framework. It forms part of the Windows Presentation Foundation (WPF) component of the upcoming Vista platform for Windows. XAML is very similar to Views in that it rides on the language interoperability of .NET. Unlike Views, there are no push-based event methods, and all handlers are also indicated as method names similar to XUL. Of course, Microsoft does not intend that anyone would actually write XAML: It is more the output notation from the GUI-builder of Visual Studio.

There is nothing intrinsically cross-platform in XAML, since it still relies on the System.Windows.Forms API for events and rendering, and thus remains closely coupled to the Windows platform. XAML is more verbose than the other notations, especially in layout. The control specification for an equals box would be

```
<Button Grid.Row="3" Grid.Column=
"0" ID="equals"Click="EqualsClicked">=
</Button>
```

The handlers are kept in a separate file, using partial classes (a feature of .NET 2.0) and connected by name to the XML, for example, private void EqualsClicked

```
    (object sender, RoutedEventArgs e) {
    double euro = 1;
    double GBP = 1;
    euro = double.Parse(eurobox.Text);
    GBP = double.Parse(GBPbox.Text);
    ratebox.Text = (euro / GBP).
       ToString("f");
}
```

Views: Vendor-Independent Extensible Windowing System

Views (http://views.cs.up.ac.za) is an XML-based library for creating GUIs that we developed so that novice programmers in C# can develop nontrivial user interfaces without having to use a professional development environment like Microsoft's Visual Studio.

Views was a project with Microsoft Research to extend GUI functionality to the shared-source Rotor version of the .NET platform, which does not include the GUI library. With minor changes, Views became ViewsQt so that it was available on any platform with .NET and Qt, notably Linux running Mono. Views forms the basis for Mirrors, a reflection-based system that offers a solution for building long-lived software.

Views consists of an XML notation and a runtime engine that initiates the rendering of controls and handles events. The engine is not a toolkit, since it does not itself define a set of widgets. Rather, it aims to expose some existing widgets in an underlying API, System.Windows.Forms.

Using 10 standard methods, the engine traps events that come from the OS and passes them to the program. Unlike XUL and XAML, the GUI specification includes no code, and the program does all event handling.

edge of the GUI in the VM, OS, or both, which are responsible for rendering widgets and event catching and redirecting. A popular way to achieve platform independence at the GUI level is to base software on a

multiplatform toolkit that already has several implementations for common platforms. A program written using one of these toolkits can move among the supported platforms without alteration.

Multiplatform GUI toolkits have long been popular for enhancing the capabilities of languages and packages lacking built-in GUI facilities. Recent examples are Rapid for Ada[7] and FranTk for Haskell.[8] Because these languages have no user interface capability of their own, they adopt the toolkit's interface, and the programmer inserts code to interact with the toolkit directly. For languages with a GUI capability, however, the toolkit's interface is essentially nonstandard for a programmer trained in that language. Front-ends for building GUIs and XML notations can alleviate this situation, but in general these toolkits provide *either* platform *or* language independence, not both.

In the .NET world, projects similar to Views have ported GUI toolkits onto the Common Language Infrastructure (CLI). The Mono project has translated the Gtk+ toolkit into C#, the result being Gtk#.[9] The programmer familiar with Gtk will feel comfortable calling the well-known methods, but a .NET programmer who must port a Windows program could be at a loss. Creating a label, textbox, and button in Gtk# consists of

```
Label label = new Label("Password");
Entry entry = new Entry();
Button button = new Button("Submit");
```

which is quite different from the Windows equivalent:

```
Label label = new Label();
label.Text = "Password";
Textbox entry = new Textbox();
Button button = new Button();
button.Text = "Submit";
```

In other words, Gtk# is not a means for porting existing Windows programs through the CLI to the Linux platform. Qt# is a similar project, intended to provide a binding of Qt to C#, but it is still under development.

Taking a long-term software engineering approach, we used the classic front-end to interface to the back-end approach to remove all platform-dependent parts of the Views engine and replace them with a cross-platform GUI toolkit, in this case Qt. We started by extracting the elements of Views that would be common across toolkits, namely XML checking, parsing, and abstract control creation. This group became the Views front end. We created a special interface consisting of a few easy-to-learn methods for event handling and replaced all references to the System.Windows.Forms library classes with calls to our interface.

So far, we had done all programming in C#. However, because Qt provides its own interface, which is in C++, we created an implementation of our interface for the Qt windowing toolkit and provided a set of classes to delegate calls from the C# objects to their counterpart C++ objects. The resulting system, ViewsQt, can run on any platform on which Qt runs, including Linux.[10]

Our experiments have shown that the ViewsQt code is portable with only a few changes to the C++ classes (related to interface inclusion and entry-point specification) to compile and execute the code on Linux and Mac OS X. On the Windows platform, ViewsQt works well with the .NET Framework, Rotor, and Mono. In essence, we were able to transform the system in Figure 4b (Views) to that in Figure 4c (ViewsQt).

Reflective libraries

The remaining dependency to be tackled was between XML, the engine, and the toolkit. The engine explicitly exposes the toolkit's classes and the methods, which are revealed in XML. Although it's possible to move between toolkits that have similar functionality, major change would be difficult, which is where reflection comes in. With the advent of .NET 2.0 and its new language features, such as generics and anonymous methods, we were able to replace the Views system with Mirrors—a system that reflects the GUI toolkit of choice, be it Views, Qt, System.Windows.Forms, or any other.

From the user's view, Mirrors shares the same objectives as Views, XAML, and XUL: to provide an XML notation for GUI building and some means of connecting event handlers for runtime use. However, the primary mechanism is reflection of the library to be used. Consequently, in theory, Mirrors can instantiate any class within any library and use all the defined properties and events for the specified class.

None of this information is hard-coded within the system; it is available at runtime through the extensive use of reflection. Mirrors can find relevant classes, events, and properties at runtime without generating any code or assemblies that need to be added during compilation. Mirrors' basic function is to provide a way of creating the GUI and giving programmers easier access to the controls within that GUI. This result is complete control over the GUI once the programmer has built it, and the user can operate on the generated object as if the Mirrors system were not there. Figure 4d shows the resulting system.

For backward compatibility with Views specifications, Mirrors recognizes two additional layout tags <horizontal> and <vertical>, as Figure 6a shows. Each tag introduced in XML is associated with a class in the library. The tag's attributes map onto properties of the corresponding class. For example, in

```
<listbox name="picturelist"
   type="pixmaplist" />
```

```
<form text="My Photo Album">
  <vertical>
    <label text="My Photo Album"
       fontsize="16" halignment="center" />
    <horizontal>
      <listbox name="picturelist"
         type="pixmaplist" />
      <label name="picture" />
    </horizontal>
    <horizontal>
      <space orientation="horizontal" />
      <button name="exit"  />
    </horizontal>
  </vertical>
</form>
```

(a)

(b)

Figure 6. Sample input to Mirrors. (a) Input that generates an image and thumbnails. (b) Using reflection, Mirrors can find relevant classes, events, and properties at runtime without generating any code or assemblies that need to be added during compilation.

`listbox` is the tag associated with the control class ListBox; `name` and `type` are attributes associated with the corresponding properties.

Class properties are discovered through reflection, and if the program finds them to be a string, it simply gives the property the value that follows. If the property type is numeric, such as integer or double, the program first passes that value through the object's Parse method to generate a suitable object. `Type` was not a property visible in our original Views system. The value of `pixmaplist` creates the thumbnails in Figure 6b. Thus, we have harnessed the power of an underlying toolkit without having it known to the engine or program.

Events are bound to methods within the user class, which pass into the Mirrors class when instantiated, thereby letting the programmer bind method handlers to the controls. The event handlers do not have to pass through a proxy layer and are bound directly onto the created controls, giving the user complete control of the object.

Programmers can use one of two methods to access the GUI from the program: They can call the methods through an engine such as the one Views provides:

```
form.PutImage("picture", photos[n, 1]).
```

Or, more generally, they can call the actual controls themselves, since the controls are available as objects through the reflective interface:

```
(xml["picture"] as PictureList).Image =
photos[n,1].
```

The first method requires knowing one of an ancillary engine's methods (in this case Views). The second requires knowing the underlying toolkit's controls. The developer can choose where to draw the line. The beauty of the method is that, with simple refactoring, the developer can switch from one to the other during retargeting.

Through middleware built using reflection and controlled with XML-based specifications, we achieved a high degree of platform and language independence for a GUI library. But other technologies would also benefit from greater portability: for example, the libraries associated with speech or gesture recognition and handwriting translation. A growing trend is wearable computing devices, which require tangible user interfaces. TUIs integrate digital information with everyday physical objects such as electronic tags, bar codes, and even clothing. The software that drives these devices will undergo the same rapid development cycle that we saw with cell phones, and new versions will constantly emerge for newer models. In this domain, our solution would encapsulate the essence of the TUI and enable the drivers or toolkits to migrate in time.

The size of mobile devices such as phones and PDAs restricts GUI capabilities. Such devices also tend to run their own VM and OS versions, making them prime candidates for our reflective approach.

Adapting a GUI to a variety of resources with different capabilities is one of the most interesting issues in mobile computation. One approach[11] maps a single user interface specification to differing devices using mobile agents built with XUL. Another approach[12] isolates features that are common across various use contexts and specifies how the output would adjust when the context changes. Most of this work, however, is at the front end, where adaptability is confined to variations of a single OS, such as Java and JavaME, or Windows and Windows CE. Our approach, in contrast, targets backend portability as well.

Reflection is a software mechanism that transcends change. Coupling it with XML and toolkits gives a brighter future for software developers.

Acknowledgments

This work was supported by Microsoft Research, a South African Department of Trade and Industry (THRIP) grant, and a Discovery grant from the Natural Sciences and Engineering Research Council of Canada. We acknowledge the hard work and inspiration of David-John Miller and Hans Lombard (Mirrors), Basil Worrall (ViewsQt) and Jonathan Mason (Views 2).

References

1. C. Ghezzi, M. Jazayeri, and D. Mandrioli, *Fundamentals of Software Engineering*, Prentice Hall, 1991.
2. J. Koskinen, "Software Maintenance Costs," 2003; www.cs.jyu.fi/~koskinen/smcosts.htm.
3. D. Stutz, T. Neward, and G. Shilling, *Shared Source CLI Essentials*, O'Reilly, 2003.
4. P.P. da Silva, "User Interface Declarative Models and Development Environments: A Survey," *Proc. DSV-IS2000*, LNCS 1946, Springer-Verlag, 2000, pp. 207-226.
5. M. Rochkind, "XVT: A Virtual Toolkit for Portability between Window Systems," *Proc. Usenix Winter Conf.*, Usenix, 1989, pp. 151-163.
6. J. Bishop and N. Horspool, "Developing Principles of GUI Programming Using Views," *Proc. ACM SIGCSE*, ACM Press, 2004, pp. 373-377.
7. M.C. Carlisle and P. Maes, "RAPID: A Free, Portable GUI Designer for Ada," *Proc. SIGAda 98*, ACM Press, 1998, pp. 158-164.
8. M. Sage, "FranTk—A Declarative GUI Language for Haskell," *Proc. 5th ACM SIGPLAN Conf. Functional Programming*, ACM Press, 2000, pp. 106-117.
9. N.M. Bernstein, "Using the Gtk Toolkit with Mono," Aug. 2004; ondotnet.com/pub/a/dotnet/2004/08/09/gtk_mono.htm.
10. J. Bishop and B. Worrall, "Towards Platform Interoperability: Retargeting a GUI Library on .NET," *Proc. 3rd Conf. .NET Technologies*, Union Agency-Science Press, 2005, pp. 23-33.
11. N. Mitrovic and E. Mena, "Adaptive User Interface for Mobile Devices," *Proc. 9th Int'l Workshop Interactive Systems Design, Specification and Verification*, Springer-Verlag, 2002, pp. 247-261.
12. J. Eisenstein, J. Vanderdonckt, and A. Puerta, "Applying Model-Based Techniques to the Development of UIs for Mobile Computers," *Proc. ACM Conf. Intelligent User Interfaces*, ACM Press, 2001, pp. 69-76.

Judith Bishop *is a professor of computer science at the University of Pretoria. Her research interests are programming languages, distributed systems, and Web-based technologies. Bishop received a PhD from the University of Southampton. She is a fellow of the British Computer Society, the Royal Society of South Africa, and the South African Academy of Science and is a member of the IEEE Computer Society and the ACM. Contact her at jbishop@cs.up.ac.za.*

Nigel Horspool *is a professor of computer science at the University of Victoria. His primary research interests are in compiler construction and programming language implementation. He received a PhD from the University of Toronto. He is a member of the ACM. Contact him at nigelh@cs.uvic.ca.*