Multi-platform User Interface Construction – a Challenge for Software Engineering-in-the-Small

Judith Bishop
Department of Computer Science
University of Pretoria
Pretoria 0002
South Africa

jbishop@cs.up.ac.za

ABSTRACT

The popular view of software engineering focuses on managing teams of people to produce large systems. This paper addresses a different angle of software engineering, that of development for re-use and portability. We consider how an essential part of most software products – the user interface – can be successfully engineered so that it can be portable across multiple platforms and on multiple devices. Our research has identified the structure of the problem domain, and we have filled in some of the answers. We investigate promising solutions from the model-driven frameworks of the 1990s, to modern XML-based specification notations (Views, XUL, XIML, XAML), multi-platform toolkits (Qt and Gtk), and our new work, Mirrors which pioneers reflective libraries. The methodology on which Views and Mirrors is based enables existing GUI libraries to be transported to new operating systems. The paper also identifies cross-cutting challenges related to education, standardization and the impact of mobile and tangible devices on the future design of UIs. This paper seeks to position user interface construction as an important challenge in software engineering, worthy of ongoing research.

Categories and Subject Descriptors

D.2 SOFTWARE ENGINEERING: D.2.6 Programming Environments Graphical environments, Integrated environments, D.2.7 Distribution, Maintenance, and Enhancement, portability, D.2.13 Reusable Software, Reusable libraries.

General Terms

Performance, Design, Reliability, Human Factors, Languages, Standardization, Languages

Keywords

Graphical user interfaces, GUI library reuse, platform independence, portability, reflection, mobile devices, tangible user interfaces, .NET, XUL, XAML, Views, Mirrors

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICSE'06, May 20–28, 2006, Shanghai, China.
Copyright 2006 ACM 1-59593-085-X/06/0005...$5.00.

1. INTRODUCTION

1.1 Software engineering

Software engineering as a discipline is perceived as tackling computing in-the-large. It elevates tools and techniques from the level of a craft, to where they can be efficiently and reproducibly harnessed for the successful completion of large projects.

Thirty years ago in 1975, Fred Brooks introduced us to the mythical man month of software development [Brooks 1975] and followed this with the "no silver bullet" paper, in which he talked about software engineering as being a process of *building software* with "specifications, assembly of components, and scaffolding" [Brooks 1987]. Boehm, too, found in 1976 that software engineering was, encouragingly, concentrating on the "requirements analysis design, test, and maintenance of *applications software* by technicians in an economics-driven context [Boehm 1976]. In the same era, [Basili 1986] concurs with the notion of software engineering being concerned with "*development* not production", yet emphasising the importance of being able to experiment with and replicate the process.

The common thread in these early seminal papers by pioneers of software engineering is the desire to manage the complexity of building something big. Although software engineering is a many faceted discipline, the facets are most often viewed as forming a progression of phases from requirements through construction, testing and quality control. Fast forwarding to the SWEBOK report [SWEBOK 2004], we find the latest definition from the IEEE Computer Society: "Software engineering is the application of a systematic, disciplined, quantifiable approach to the *development*, operation, and maintenance of software; that is, the application of engineering to software."

A more unusual definition from a text book [Ghezzi 1991] states that "Software engineering is concerned with the multi-person construction of multi-version software". The popular view of software engineering focuses on the first part of this definition, that of managing teams of people to produce a (implied) large product. The second part is as important: successful software engineering has come from identifying specific parts of a product, so that they can be designed by experts, and mass-produced, free of dependencies on language and environment.

1.2 Graphical user interfaces

Graphical user interfaces are an example of such a specialization. In this paper, we consider how GUIs can be successfully engineered, not in multiple versions (reminiscent of product lines), but so that it can be portable across multiple

platforms and on multiple devices, now and in the future. Our research has identified the structure of the problem domain, and we have filled in some of the answers. This paper seeks to position user interface construction as an important challenge in software engineering, worthy of ongoing research.

The key issues are that for a given program (existing or planned) it is desirable to decouple the following four elements:

- programming language
- graphical toolkit or library
- virtual machine or runtime environment
- operating system.

In this way, we achieve separation of the GUI specification from the program so it can be reused in other programs, possibly written in other languages; and a portability path for the whole system onto other existing, or future, operating systems, with possibly different virtual machines. We investigate promising solutions to these problems, by comparing XML-based specification notations, evaluating multi-platform toolkits, and describing our new work on reflective libraries.

Throughout the paper, we note the challenges that arise in terms of the technical, production and professional aspects of GUI constructions. The focus is primarily on the technical issues of programming, particularly on the obstacles raised by portability and adaptability. However, there are also cross-cutting concerns relating to education, standardization and acceptance by the community, as well as the impact of mobile and tangible devices. It is the message of this paper that GUIs are seen to be operating behind the forefront of computing. They are not getting the attention they deserve, and they need to be brought firmly into the software engineering fold[1].

2. BASIC UI PRINCIPLES

2.1 Reference models

In the 1990s, considerable work was done on reference models for UI design, with mappings to graphical and multimedia interfaces following. As far back as 1991, Coutaz [1991] proposed a multi-agent framework applicable to the software design of interactive systems, and Dewan and Choudhary's framework tackled flexible multi-user interfaces [1992]. The issues that had to be addressed at that time were more prominently those of cognitive psychology and hardware performance. Most of the architectural research at the time refers to the baseline 1985 Seeheim Reference Model [1985] which presaged the model-view-controller (MVC) architecture with which we are now familiar.

By the mid 1990s, the advent of Apple's superb window interface, split the research community into those concerned with human computer usability and those who developed and promoted toolkits. One early such kit is Interviews from Stanford [1989], which was notable in that it already ran on four hardware platforms, all on top of one operating system, X Windows. We wish to take all of that a step further and abstract from the operating system as well. In this paper we are not focusing on GUI design or toolkit functionality, but on the engineering principles involved in the construction of modern, adaptable GUIs.

Our premise is that it should be a software engineering tenet that a GUI can be specified as a separate component in a system, and then linked in a coherent way to the rest of the computational logic of the program. The current practice is for GUIs to be specified by creating objects, calling methods to place them in the correct places in a window, and then linking them to code that will process any actions required. If hand-coded, such a process is tedious and error-prone; if a builder or designer program is used, hundreds of lines of code are generated and incorporated into one's program, often labeled "do not touch". Either approach violates the software engineering principles of efficiency and maintainability.

One of the problems of extracting a set of principles for GUIs is that many of the terms are already language specific. However, this confusion is not confined to GUIs: consider that we have case versus switch statements and base vs super classes. We will choose and stick to one set of terms, initially giving the alternatives. We start off by considering conventional screen-keyboard-mouse GUIs. Such a GUI consists of the following five parts:

- controls (or components or widgets)
- a window (or form or frame)
- a menu bar
- layout
- interaction.[2]

A GUI is seen as a *window* on the screen and contains a number of different *controls*. Controls can, for example, be labels, buttons or text boxes. The GUI very likely has a *menu bar* across the top that typically contains the name of the program or window and buttons for hiding, resizing and destroying the GUI display. Additional options can be added to the menu bar. The menu bar offers options similar to controls, but it is different in the way in which it is created and displayed.

The controls are said to have a certain *layout* that defines their position relative to each other and/or directly in the window. Both the choice of the controls and their correct layout are the concern of the programmer, even if a GUI design is specified.

Once the controls are on the screen, the user *interacts* with some of them via devices such as the mouse and keyboard. Moving the mouse over a control, and then clicking can make things happen (as with a button). Other controls allow typing from the keyboard, defining areas for outputting data, including text, images, video and sound. It is this area of interaction that is the most challenging in GUI development, and to which we shall return in Section 2.3.

2.2 Controls

Most GUI libraries[3] or toolkits offer upwards of 30 or 40 controls, each with numerous options or attributes. These attributes can be size, color, font type and so on. The attributes

[1] In this regard, it is interesting to note that Microsoft has recently coined the new acronym UX for User Experience, which is intended to address this gap.

[2] It is fascinating to note the close correspondence between this list and that of the 1989 toolkit, Interviews [16].

[3] Library and API (application programmer interface) can be used interchangeably: for clarity we use library consistently in this paper.

are set at the time the control is created, but in most systems, they can be changed dynamically. For example, a text box created as

```
TextBox password = new TextBox();
```

can later have its background set to yellow by:

```
password.BackColor = Color.Yellow;
```

Enumerating all the controls and all their attributes, even for one language, is a large task, but the appropriate choice of a control is important. For example, if we want to have an image and be able to click on it, we could use a button and set the image attribute, or we could use a picture box, which can respond to double clicking on the image specified. The picture box has extra scaling facilities, so the developer would have to choose.

> Challenge 1. Indexing and navigating through the control library.

What is needed is a compendium of controls that would be indexed not only by control names, but cross-indexed by attributes, such as clickability. Then the programmer can see what controls have this attribute, and can make an appropriate choice. Not all languages will have the same set of matches for controls and attributes, but at least, having learnt the principles, the developer knows what to look for.

2.3 Layout

When developing a GUI, we can just add controls to the window as we think of them, but that would not usually make for a very pleasing arrangement. A key feature of GUI design is to group similar controls together. Thus a designer could stipulate that all the buttons be at the top, or at the bottom of the window. We can split the screen in half, and have input boxes on one side, and output on another, and so on. The question is, how does the developer manipulate controls like this? There are two options:

2.3.1 Drag and drop tools

We can use a tool which allows us to create the code for a GUI by dragging a control from a list showing all the possibilities and dropping it onto the spot which looks right visually. We can change the placement of controls by dragging them with the mouse, and change their sizes and other attributes just as easily.

Such tools are not in themselves very complex, but when they are integrated with development environments, such as JBuilder or Visual Studio, they have a large footprint. Fortunately for the large cohort of programmers who do not favour IDEs, standalone GUI Builders are available, such as Glade. These builders output both program code (not recommended) and XML (which is discussed in Section 4.1)

2.3.2 Positioning

Without using drag and drop, we can parameterise calls to library methods that precisely position a control down to the last pixel. The code to place a button at position x=300, y=150 would be something like the following:

```
Button submit = new Button();
submit.Location = new Point(300, 150);
```

In itself, this is not difficult to write or understand, but working with many controls in absolute coordinates brings with it many extra small details to specify, and can become very long-winded. Another problem is that the size of screens varies with devices and with time, in which case absolute positioning has to be done most carefully to obtain results that will endure.

A more flexible and enduring method is relative positioning, as popularized in Java's layout managers. They offer standard arrangements such as flow, border and grid, into which components can be placed, with the system handling the actual positioning of components with suitable gaps between them. For most GUIs they work very well, though they can have the same scaling problem mentioned above.

With both methods, there is a lack of abstraction, too much hard-coding, and little commonality across libraries.

> Challenge 2. Encouraging standard and device independent positioning techniques for controls

2.4 Interaction

The definition and layout of controls is mostly a static exercise. Interaction is dynamic, in that it will continue through the life of the program, and requires careful programming to cover all possibilities. Active controls such as buttons are linked to event handlers early in the life of a program through *listeners*. For example, our submit button could be linked to an actionPerformed method by:

```
submit.Click += new
    System.EventHandler(actionPerformed);
```

Listeners essentially watch one or more controls and, when an action takes place, activate all registered event handlers. Once in a handler, we can interact with a control by calling methods defined to get data from it and put data into it. The selection of methods available depends on the language and library.

The interplay between listeners and handlers is the most complex part of GUI programming because it usually involves higher-order programming constructs, such as delegates or callbacks, and the previously assumed sequential ordering of statements is disrupted. The confusion caused in the mind of the developer can be alleviated by a better division of responsibility, and by shifting some of the work to a common engine, as discussed Section 3.1.2.

> Challenge 3. Reducing the complexity of the event-handling paradigm

3. THE PROBLEM STATEMENT

Given that writing a GUI can be a messy and labour-intensive task, a sound software engineering principle should be brought into play: re-use. However, GUIs are very seldom reused because they are closely coupled to a specific GUI library or toolkit, and these have proven to be not very portable. Let us introduce some definitions to support this statement.

Consider Figure 1. The GUI code is embedded in program P, written in language LA. The calls (to create, layout and handle controls) are to methods in the library or toolkit TK_G. The toolkit can be written in a language other than LA, but will be compiled down to the same code as LA, running on virtual machine VM_N. There is also a direct link between TK_G and the underlying operating system OS_X for the rendering of controls on output

devices and the catching of events. For example, program P in C# (=LA) using System.Windows. Forms (=TK_G) would run on the CLR (VM_N) on Windows (OS_X).

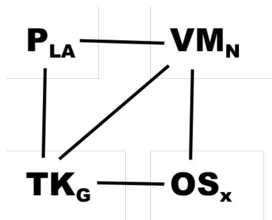

Figure 1. Normal operation of a GUI

The objective is to decouple these four elements in the following ways:

- GUI separation. Have the GUI specification separate from the program, so that it can be reused across programs;
- Language independence. Be able to link a GUI specification into a program written in a different language;
- Platform portability. Be able to move the whole system onto a different operating system platform.

Altering TK and VM is not as critical, but we shall see that they play important intermediary roles.

Example scenarios:

1. Re-using a GUI designed for a C# program in a VB program. With the traditional method-oriented GUI, the code would have to be identified, extraced and re-tooled in VB. However, the library (Windows.Forms), run-time system and the operating system could remain the same.
2. Moving a given program running on Windows to Linux. This is more difficult: it depends on whether the toolkit being used is implemented on Linux, and a runtime system exists. For C# and Windows.Forms, no such bidings existed before our work.

Thus the problem is one of having invested in developing a program based on a particular library, and then finding that the program cannot migrate to a new platform, because of the library's reliance on hardware or low-level software. If the library is a large and critical one, such as a GUI, then any alternative to a complete re-implementation of the GUI or the library would be desirable.

4. PROMISING SOLUTIONS

4.1 Reuse across languages

With the advent of Java and .NET, the possibilities for cross platform and cross language computing have increased greatly. However, libraries that support GUIs have, for the most part, not been included in this advance. For example, in the shared source version of .NET, the Windows.Forms library is specifically excluded [Stutz 2003]. There is therefore an opportunity to provide something that will travel better. Rather than design a new library – which would suffer from the same problems as other libraries – an alternative is to use XML for the specification. XML has the advantage that it is universal, and can be written by hand or emitted from a tool very easily.

The first step towards gaining independence of the operating system, is to introduce a language-independent GUI specification notation to specify the function of the library, in other words the programmer's interface. The theory for such specifications is called *declarative user interface models* [da Silva 2000], which describes user interfaces in terms of the controls displayed to the user, their composition and their layout. Two examples of the genre of declarative user interface models are IUP/LED [Levy 1996] and CIRL/PIWI [Cowan 1993]. In both cases, a declarative language (LED and CIRL) was provided to describe the user interface in terms of its controls and layout. On the library front, they contain functions for hooking events signaled by the interface to call-back methods defined in the user's application, and functions to query and alter attributes of the controls displayed. The call-back event model is used so that the usual native windowing toolkit's events are filtered down to those relevant to the application.

Such declarative user-interface models are not new and XML is broadly being adopted as the favourite notation for these models. Our work on the Views[4] project has also defined a notation that draws on the underlying Windows.Forms toolkit, but exercises freedom in improving the layout structure, more in the line of Java.

For example, instead of the label, textbox and button code:

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

we would write an XML specification such as:

```
<Label text="Password"/>
<Textbox name=entry/>
<Button name=button text="Submit"/>
```

Using XML in this way was proposed almost ten years ago, and it is gaining in acceptance. We will look at the XML-based systems XUL, XAML, and Views. For comments on Glade see 3.1.4.

4.1.1 XUL

XUL is the model used by the Mozilla family of browsers. XUL has a rich notation for creating widgets, and uses Box, Grid and other layout models. An example of a the specification of a simple currency calculator in XUL is:

```
<window onload="loadWindow();"
    id="win" title="Currency calculator"
    orient="horizontal"

    <script src="calculator.js"/>
    <grid>
      <rows>
        <row>
          <label id="l1" class="small"
            value="Paid on hols"/>
          <textbox id="eurobox"/>
        </row>
        <row>
```

[4] Throughout this paper, projects and products whose primary source of information is a website are listed at the end of the paper, but the sources are not cited in the text.

```
            <label id="l2" class="small"
             value="Charged"/>
            <textbox id="GBPbox"/>
          </row>
          <row>
            <label id="l3" class="small"
             value="Exchange Rate is"/>
            <textbox id="ratebox"/>
          </row>
          <row>
            <button label="="
             oncommand="equals();"/>
            <button label="Reset"
             oncommand="clear();"/>
            <button label=">>"
             oncommand="next();"/>
          </row>
        </rows>
      </grid>
    </window>
```

that will produce the window:

By virtue of its being embedded in html. the XUL code is inherently cross-platform. The scripts for the handlers are not in the program, but written in Java Script, 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 program, written in its language.

XUL has a sister language, XBL (Extensible Binding Language) in which additional customization of the widgets can be written.

4.1.2 Views

Views is the result of a research project with Microsoft Research to extend GUI functionality to the shared-source Rotor version of the .NET platform, which did not include the GUI library [Bishop 2004]. The Views system consists of an XML-notation and a runtime engine which initiates the rendering of controls and handles events.

By means of 10 standard methods, the engine is able to trap events that come from the operating system and passes them on to the program. Unlike XUL, no code is included with the GUI specification. The calculator shown before would be entered as a string in the program (or read from a file) as:

```
static string specEn =
    @"<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>";
```

The handlers can be set up with callbacks, or can use a simple event loop, part of which is

```
  case "clear":
    euro=1; GBP=1;
    f.PutText(
      "eurobox",euro.ToString("f"));
    f.PutText(
      "GBPbox",GBP.ToString("f"));
   break;
```

A recent extension of Views Views 2.0, allows the inclusion of style sheets and other customisable information, similar to XUL's XBL [Mason 2005].

4.1.3 XAML

XAML is the declarative markup language Microsoft is making available with Version 2 of the .NET Framework, as 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 in XAML, in a similar way 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 Windows.Forms for events and rendering, and thus remains closely couple to the Windows operating system.

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, making use of partial classes (a .NET 2.0 feature) and connected by name to the XML, e.g.

```
   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");
   }
```

4.1.4 Comparison

All the notations achieve the goal of re-usable GUI specifications. The design of the XML, in each case, is somewhat tied to the expected underlying library, but the commonality among widget names and behaviours is considerable.

The big difference between the two commercial notations and Views is that both XUL and XAML allow (but do not compel)

the programmer to embed event-handling code (JavaScript, and any .NET language, respectively) within the user interface declaration. The Views model, on the other hand, provides an engine that intercedes on behalf of the GUI to signal events to the host application. The implementation language of the engine is irrelevant to the Views user, who can be programming in VB, C++ etc.

While the functionality offered by XUL and XAML is attractive, we contend that the separation of concerns evinced by Views' engine-based approach is cleaner and offers greater maintainability and ease-of-use to the programmer and designer.

Hybrid systems worth mentioning that use the XML approach are Glade and OpenLaszlo. Glade is tied to the Gtk+ toolkit and Gnome desktop environment. Its features bear much in common with the multi-platform toolkits discussed in 3.2.1. OpenLaszlo is aimed at the web market for a rich client experience and works with Eclipse and Macromedia's Flash Player and is now supported by IBM.

4.2 Platform independence

Referring to Figure 1, the more difficult and less traveled step is to change the virtual machine and/or operating system, and thus the platform of a GUI based program. These low-level components are responsible for the rendering of widgets and the catching and redirecting of events.

4.2.1 Multi-platform toolkits

A popular way of achieving platform independence at the GUI level is to base a program on a multi-platform toolkit which already has several bindings for common platforms. Three commercially available toolkits are Tcl/Tk, Gtk+ and Qt. If a program is written using one of these toolkits, then it can move among the supported platforms without alteration.

Multi-platform 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 [Carlisle 1999], FranTk for Haskell [Sage 2000] and SMLTk for ML [Lüth 2000]. Because these languages have no UI capability of their own, they adopt the interface of the toolkit, and the programmer inserts code to interact with the toolkit directly.

However, the problem is that languages with a GUI capability, the programming interface presented by these toolkits is essentially "non-standard" for a programmer trained in a particular language GUI builder front-ends and XML notations can alleviate this situation, but in general with these toolkits, one can gain platform or language independence, but not both.

In the .NET world, there have been projects similar to Views to port GUI toolkits onto the CLI. Gtk# is a translation by the Mono project of the Gtk+ toolkit into C# [Bernstein 2004]. The programmer familiar with Gtk will feel comfortable calling the well-known methods, but a .NET programmer with a Windows program to port could be at a loss. As a simple example, creating a label, textbox and button in Gtk# is done with:

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

which is quite different to the Windows equivalent of:

```
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 via the CLI to the Linux platform. Qt# is a similar project intended to provide a binding of Qt to C#, and is still under development.

4.2.2 A targeting methodology

Clearly there is a gap here. What we want is to achieve is the front-end flexibility of XUL and the back end portability of a Gtk. in order to achieve our goal of re-usable GUIs for multiple platforms. Glade almost has the best of both worlds, but does not tackle the important issue of retaining the programmer interface of the original language library.

Taking a long-term software engineering approach, we developed a methodology for removing all platform dependent parts of our Views system, and replacing them with a platform independent GUI, in this case Qt. ViewsQt is a conversion of an XML-based GUI library to support a retargetable back-end. The project involved extracting the common front-end elements of XML checking, parsing, and abstract control creation from the original Views engine, and replacing references to the Windows Forms library classes with calls to a C# interface. This interface hides the toolkit-specific back-end components behind a small (and easy to learn) set of methods. Finally, we created an implementation of this interface for the Qt windowing toolkit, and provided a set of classes to delegate calls from the C# objects to their counterpart C++ objects. [Bishop 2005].

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) required to compile and execute the code on the Linux and Mac OS X operating systems. On the Windows platform, ViewsQt works well with both the .NET Framework and Rotor.

The two research systems mentioned earlier, CIRL/PIWI [Cowan 1993] and IUP/LED [Levy 1996], were designed from the start to abstract the GUI description from the underlying platform's toolkit, and to provide a similar look-and-feel across the various platforms. The creators of both projects, however, lamented the absence of an existing toolkit that provided a common look-and-feel across various platforms (both projects were born in the pre-Java and before any widely-accepted platform-independent toolkits, such as Qt and Tcl/TK, were available). Our work on the ViewsQt project was not hindered by these concerns because of the high-quality, platform independent toolkits available to us today.

In summary, we have been able to transform the system depicted in Figure 1 to that of Figure 2:

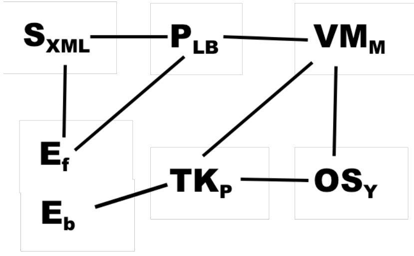

Figure 2. Retargetted GUIs

The GUI is specified completely independently in XML. The XML is fed into the program, where identifiers are matched with controls and handlers there at compile time. The handlers are embedded in the program. The XML is interpreted by the engine (in this case ViewsQt) and requests for rendering GUI controls are passed to the backend, from there to the platform independent toolkit (Qt) and over to a operating system such as Linux. The toolkit and program also interact with the virtual machine (if present). Once this system is in place, the specification, language, virtual machine and operating system can all change independently. The pre-existing engine and toolkit are hidden from the programmer, and just go along with the system.

4.3 Reflective libraries

Mirrors is a project which began as an investigation into the power of reflection, and became feasible with the advent of .NET 2.0 and its new language features, such as generics, anonymous methods, and the extended Windows.Forms layout. From the user's point of view, the first case study of Mirrors (described here) shares the same objectives as Views, XAML and XUL: to provide an XML notation for GUI building and some means for connecting up event handlers for runtime use. However, the primary mechanism is reflection of the library that is to be used, and therefore the Mirrors methodology is not at all restricted to a GUI library. In short, Mirrors can

- instantiate any control within any library, and
- make use of all the defined properties and events for the specified control.

None of this information is hard-coded within the system, as it is all available at runtime through extensive use reflection. Mirrors is capable of finding the relevant classes, events and properties at runtime without generating any code or assemblies that need to be added at compile time.

Mirrors' basic function is to provide a way of creating the GUI, and allowing the programmer easier access to the controls contained within it. This approach allows complete control over the GUI once it has been built, and the user may operate on the generated object as though the Mirrors system was not there.

4.3.1 The specification phase

Using a notation similar to that of XAML, Mirrors is able to generate a GUI from a simple or complex specification to give an industrial scale GUI that is portable. The XML file can be re-used to generate the same GUI on other systems, if the control names are the same.

Mirrors has been written using C# 2.0 which now has support for a wider set of controls, including layout managed panel controls. Additional layout managers may be bound using the "Layout" event defined within the control super-. For backward compatibility with Views specifications, the two additional layout tags <horizontal> and <vertical> are specially recognized.

As an example, consider a specification for a GUI with a label, textbox and button.

```
<FlowLayoutPanel FlowDirection="LeftToRight"
    Dock="Fill"
    BorderStyle="fixed3d" AutoSize="true">
    <label name="lblQuestion"
         text="Quantity" width="300"/>
    <label name="lblText" text="Quantity"
         autosize="true"/>
    <textbox name="qty" width="120"/>
    <button name="do" text="Buy"
        click="Buy"/>
</FlowLayoutPanel>
```

The strings associated with the name attributes can be used in the program to refer to the relevant control. Thus to read a quantity, the program would access the "qty" control. The button control is called "do" although the text "Buy" is displayed. The button also supplies a link to the click event called "Buy". By using implicit operator overloading of [ ], every attribute of every control can be interrogated and changed, as in:

```
private void Item(object o, EventArgs args) {
  item = (o as Control).Text;
  (xml["lblQuestion"] as Label).Text =
    "How many " + item + " would you like?";
  (xml["qty"] as TextBox).Text = "0";
}
```

Here the label question is changed from "Quantity" to "How man xxx would you like" where xxx is picked up from another control.

4.3.2 XML Parsing

Mirrors uses XPath as a flexible XML reading format, which allows us to easily traverse the nodes and their attributes within the XML document. The XML document is parsed in a single sweep, with relatively little margin for error. A fundamental drawback of XML parsing is the lack of knowledge of what may or may not be defined within a tag, or even what a tag may be, because that knowledge is only obtained by reflection when the system is run. The most basic of errors that may be displayed are

- simple XML formatting problems,
- the attribute is not defined within the tag's corresponding class, or
- the tag is not a recognized class.

Mirrors reacts to these errors either by throwing an exception, or ignoring the illegal name.

Challenge 4. Extracting better error messages during reflective parsing

In Views, the decision was taken to check the XML using a hand-written parser, so that more errors could be detected and reported at compile time. The fundamental difference is, of course, that in Views the set of controls and their attributes are already known.

Controls are specified using a basic syntax, which is easy to understand, and the attributes are those from the underlying library that is being exposed. This approach gives a gradual learning curve: a novice can work with a few obvious controls and events, and then branch into more complicated options later.

4.3.3 Reflective method binding

Each tag introduced in the XML is associated with a class in the library. The tag's attributes map onto properties of the corresponding class. For example, in

```
<textbox name="qty" width="120"/>
```

textbox is the tag associated with the control class Textbox; name and width are attributes associated with the Name and Width properties. The properties of a class are discovered via reflection, and if found to be a string, the property is simply given the value that follows. If the property is of a numeric type, such as integer or double, the value is first passed through the object's "Parse" method to generate a suitable object to be assigned `[int.Parse(string)]`. A special case is also provided for Enumeration types, where the values may be specified by a string name instead of the numeric value by using the Enum.Parse static method:

```
[(enumtype)Enum.Parse(
     typeof(enumtype), string)].
```

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

Each XML document establishes a main object called Form. Using the instantiated Mirrors instance, the developer has full access to the controls via a simple text string. For example:

```
(mirror["qty"] as TextBox).Text = "0";
```

Mirrors does not restrict the programmer to a fixed set of methods and properties exposed by the system.

4.3.4 Summary

The Mirrors approach can be applied to other libraries. One such project would be to change the GUI library to be exposed from Windows.Forms to GTK, as was done in the Views project. It is anticipated that this retargeting would be even simpler than that reported in Section 4.2. Other libraries that could benefit from the Mirrors approach would be those associated with speech recognition, or handwriting translation. The advantage of Mirrors over accessing the raw library is that the XML specification remains separate and language independent. This approach aids separation of concerns and reuse.

5. FUTURE CHALLENGES

To summarise, what we achieved with the ViewsQt and Mirrors projects was crafting of one-off systems, but they proved that the methodology was sound, and that goal of GUI independence was obtainable.

We can now identify the challenges associated with raising multi-platform user interface construction to the level of software engineering, as well as considering some of the wider issues that will arise in the future.

5.1 Converging notations

The critical component in the methodology described in 4.2.2 is the notation in which the XML specification is written. At present, each system has its own notation, examples of which were shown in Section 4.1. The methodology can be used to move a program using one notation to toolkit which supported on a new platform, but which uses a different notation. It many cases, this would be preferable to rewriting GUI code in the program.

> Challenge 5. Establishing and developing core sets of widgets

The situation calls out for the recognition of GUIs as a serious construction technology. The number of well-used GUI toolkits and libraries is now relatively small – under ten – and further convergence could be possible. The evolution of GUIs in the software engineering domain is to contrast to that of hi-performance computing. Here too, there are several competing libraries for managing communication between processes, but the overall impression is that the community is striving towards standardization as a means of obtaining portability. Thus some libraries such as MPI last for a long time, but new ones do arise, and they often have cross-compatibility to the old, such as OpenMP, a shared memory communication model, but which exists in hybrid modes with MPI.

> Challenge 6. Teaching principles of GUI programming in a notation-independent way.

In the same way as we talk about for loops or class constructors, for example, we should be able to talk about box layouts and on-focus events.

Education is essential in raising awareness of GUIs programming. Because GUIs are implemented via libraries, and not part of a language, educators have a tendency to leave them off the syllabus, or to present them in a "just do it this way" manner, while trainers and self-taught programmers naturally gravitate to only one instance of GUI-dom.

5.2 Mobile devices

Mobile devices such as phones and PDAs have restricted GUI capabilities because of their size. They also tend to run their own versions of virtual machines and operating systems, making them prime candidates for the reflective library approach described in Section 4.3. Everyone agrees that adapting a GUI to a variety of resources with different capabilities is one of the most interesting questions of today's mobile computation. Mitrovic and Mena [2002] discuss how to map a single user interface specification down to differing devices, based on mobile agents built with XUL. Eisenstein *et al* [2001] develop an abstract approach that serves to isolate those features that are common various contexts of use, and to specify how the output would adjust when the context changes. However, most of the work is at the front-end, and the adaptability is confined to a single operating system.

> Challenge 7. Adapting GUIs to a variety of output devices with different operating systems and GUI libraries.

An interesting commercial application is CrossFire [Lee 2004] from AppForge, which is a third-party product built on top of .NET. Crossfire uses a booster to the CLR to enable code in VB, C# etc to run on the compact frameworks used by a variety of mobile devices, such as cell phones and palmtops. The controls mimic the device user interface of Windows.Forms, without its large footprint. In this way, Crossfire also enhances portability.

5.3 Tangible user interfaces

Although this paper concentrated on GUI libraries, which imply a screen-keyboard-mouse interface, other emerging hardware-

oriented technologies have the same problem of portability, for example the libraries associated with

- speech recognition
- handwriting translation
- gesture recognition.

Among current research are the tangible user interfaces (TUIs) otherwise known as physical user interfaces. TUIs integrate digital information with everyday physical objects such as electronic tags and barcodes. They accept input from the natural world, and are context-aware.

Papier-Mâché [Klemmer 2004] is an open-source toolkit for building TUIs with a high-level event model to facilitate portability. As the writers say: "[Before Papier-Mâché] building these UIs required "getting down and dirty" with input technologies such as computer vision. Consequently, only a small cadre of technology experts could build these UIs."

The process of establishing a core set of widgets (Challenge 5) is currently underway. The next challenge will be to make them portable.

> Challenge 8. Making tangible user interfaces conform to high-level standards of portability from the start.

6. CONCLUSIONS

This paper sought to position user interface constructions as an important challenge in software engineering research. We identified the major issues as language independent specifications and portability of GUI libraries. Various approaches and remaining challenges to these were described. In particular, the idea of reflective libraries holds promise. Cross-cutting concerns in education and new devices will supply additional challenges for some time to come.

Overall, what was leant from this study was that the methods and ideals of mainstream software engineering do encompass the needs of user-interface construction, but that they need to be honed into a body of knowledge, accessible to all involved in the field.

7. ACKNOWLEDGMENTS

This work was supported by Microsoft Research and THRIP Grant no. 2788. I acknowledge the inspiration of Nigel Horspool of the University of Victoria (Views), David-John Miller (Mirrors), Basil Worrall (ViewsQt) and Jonathan Mason (Views 2).

REFERENCES

[1] Basili V R and R W Selby , D H Hutchens, Experimentation in software engineering, *IEEE Trans. Soft Eng*. 12 (7) 733-743, 1986

[2] Bernstein Niel M, Using the Gtk toolkit with Mono, *O'Reilly ONDotNet*, online article 2004/08/9/ August 2004.

[3] Bishop Judith and Nigel Horspool. Developing principles of GUI programming using Views. *Proc. ACM-SIGCSE*, 373-377, March 2004.

[4] Bishop Judith and Basil Worrall, Towards platform interoperability: retargeting a GUI library on .NET, *Proc. 3rd Conf .NET Technologies*, 23-33, Plzen, Czech, May 2005

[5] Boehm B W, Software Engineering, *IEEE Trans Computers*, (12) 1226-1241, 1976

[6] Brooks Frederick P, *The Mythical Man Month*, Addison-Wesley, 1975

[7] Brooks Frederick P, No silver bullet: essence and accidents of software engineering, *Computer*, 20 (4) 10-19, 1987

[8] Carlisle, Martin C and P. Maes. RAPID: A free, portable GUI designer for Ada, *SIGAda '98,* 158-164, ACM, 1998

[9] Coutaz Joelle, Architectural design for user interfaces, *ESEC* 7-22, 1991.

[10] Cowan D D et al. CIRL/PIWI: A GUI toolkit supporting retargetability. *Software—Practice and Experience*, 23(5) 511–527, 1993.

[11] da Silva Paulo Pinheiro. User interface declarative models and development environments: a survey, *Proc. DSV-IS2000, LNCS 1946*, 207–226, Springer-Verlag 2000.

[12] Dewan Prasun and Rajiv Choudary, A high-level and flexible framework for implementing multi-user user-interfaces, ACM Trans, on Information Systems, 10 (4) 355-380, 1992

[13] Eistenstein Jacob, Jean Vanderdonckt and Angel Puerta, Applying model-based techniques to the development of UIs for mobile computers, *Proc. ACM Conf on Intelligent User Interfaces* 69-76, 2001.

[14] Ghezzi Carlo, Mehdi Jazayeri, Dino Mandrioli, *Fundamentals of software engineering*, Prentice Hall, 1991

[15] Klemmer Scott R *et al*, Papier-Mâché: toolkit support for tangible input. *CHI 2004: Proc. ACM Conf. on Human Factors in Computing Systems, CHI Letters*, 6 399-406, 2004.

[16] Lee Wei-Meng, Writing cross-platform mobile applications using Crossfire, *O'Reilly ONDotNet*, online article 2004/07/12, 2004

[17] Levy C H *et al*, IUP/LED: A portable user interface development tool. *Software—Practice and Experience*, 26(7):737–762, 1996.

[18] Linton Mark A, John M Vlissides and Paul R Calder, Composing user interfaces with Interviews, *IEEE Computer* 8-24, February 1989

[19] Lüth C B, Wolff, TAS - A generic window inference system, *13th Conf on Theorem proving and higher order logics, LNCS 1869*, 405-422, Springer-Verlag 2000.

[20] Mason Jonathan, Views 2: Reflections on Views, *MSc Thesis*, University of Victoria, Canada, 2005

[21] Mitovic Nikola and Eduardo Mena, Adaptive user interface for mobile devices, *Interactive Systems: design, specification and verification, Interactive systems: desogn, specification and verification*, 9th Intl. Workshop, 47-61, Springer-Verlag, 2002

[22] Pfaff GE, *User interface management systems: proceedings of the Seeheim Workshop*, Springer Verlag 1985

[23] Sage Merig, *FranTk* - a declarative GUI language for Haskell, Proc. 5th ACM SIGPLAN conf. on Functional Programming, 106 – 117, 2000

[24] Stutz David, Ted Neward, and Geoff Shilling, Shared source CLI essentials, O'Reilly, 2003

[25] SWEBOK Guide to the software engineering body of knowledge, IEEE Computer Society, www.swebok.org, 2004

WEB REFERENCES

CLI	www.ecma-international.org
Crossfire	www.appforge.com
Eclipse	www.eclipse.org
Glade	glade.gnome.org/
Gnome	www.gnome.org/
Gtk#	gtk-sharp.sourceforge.net
Gtk+	www.gtk.org
JBuilder	info.borland.com/techpubs/jbuilder
Mono	www.go-mono.com
MPI	www-unix.mcs.anl.gov/mpi/mpich/
OpenLaszlo	www.laszlosystems.com
OpenMP	www.openmp.org
Qt	www.trolltech.com
Qt#	qtcsharp.sourgeforge.net
Tcl/Tk	www.tcl.tk
Rotor	msdn.microsoft.com/net/sscli
Views	views.cs.up.ac.za
Vista	www.microsoft.com/windowsvista
WPF	www.microsoft.com/windowsvista
XAML	www.xaml.net/
XUL	www.xul.org