Intentions in WordPad

Kevin Backhouse and Bernard Sufrin*

Draft Working Paper of 7th July 1999

Abstract

In this note we summarize the findings of the preliminary investigation of WordPad that we have been conducting in anticipation of reconstructing that program in an intentional setting. Three important kinds of intentional construct have been identified. Intentions of the first kind appear as a consequence of the style of implementation of the Microsoft Foundation Classes framework. Intentions of the second kind appear as a consequence of the requirement (widespread in GUI programming) that an invariant relation hold between one or more components (visible or otherwise) of the state of a composite structure. Intentions of the third kind are fairly ubiquitous in C++ programs – they appear in extensional form as idioms only because they have not been institutionalised as part of the notation. We give examples of all three kinds of intention. Eventually we will suggest intentional constructs that might be used to capture them more concisely.

1 Introduction

Intentional Programming (IP) is a style of programming in which a general-purpose programming language base is augmented with constructs (called Intentions) which represent domain-specific abstractions and actions. The idea is that a program for use in a specific problem domain (*e.g.* music, text editing, DNA sequencing, compiler construction) can be described using forms of language which are more closely related to the normal discourse of experts familiar with the domain in question. New notational forms may also represent common design patterns, again with the goal of more closely matching the programming notations to the task at hand.

An IP system must provide a way of transforming a program that uses domain-specific notations into runnable form. During the course of this translation such a system should be also able to apply any domain-specific optimizations of which it is aware, and part of the purpose of an intentional metalanguage is to provide a means of describing such optimizations.

We are exploring several potential sources of intentions, in new application domains as well as in existing software – where the exploration process has been dubbed *intention mining*. One of our subjects is Microsoft's WordPad program – distributed both as a useful tool and as an example of how to exploit the Microsoft Foundation Classes (MFC) application framework through the Visual C++ programming environment. The source text of WordPad consists of about 11,000 lines of C++.

* Programming Tools Group, Oxford University Computing Laboratory.

Nearly all of this is concerned with describing the user-interface to the *RichEdit* control – an MFC library component that provides most of the core functionality of WordPad. Our goal is to identify a collection of intentions suitable for presenting a higher level and more concise description of the program. Anyone who tries to read the present source text will soon see the benefits that such a description could bring in terms of understandability and maintainability. We hope eventually to be able to build a version of WordPad using an intentional programming system.

Class Wizards

Superficial evidence in support of the conjecture that a higher level description is feasible comes from the fact that a good deal of the code was written by Visual C^{++} *Class Wizards*. A Wizard is an agent, built in to the Visual C^{++} programming environment and knowledgeable about the structure of MFC, that writes boilerplate code for the programmer, who specifies its basic structural and functional characteristics by form-filling. The programmer has to edit the boilerplate code, and add application-specific code to it. We have been investigating the extent to which the design space spanned by Wizards can be characterized by intentional forms.

Visual Invariants

A form of intention which is frequently found in the implementation of a direct-manipulation style of user-interface to a system is embodied in code that maintains some kind of invariant relation between two or more aspects of the system state, one of which is available for display. In the WordPad interface the main examples of this kind of intention are provided by the formatting toolbar and the ruler.[1]

The *ruler* is used to set and to show the positions of the margins and tab stops in the paragraph which currently contains the insertion cursor. Manipulation of the pictorial representations of tabs and margins on the ruler changes their settings in the current paragraph; moving the insertion cursor to a new paragraph causes the ruler to show the state of the margins and tabs for that paragraph.

The *formatting toolbar* plays a similar role in relation to the font and justification settings in the current selection, but its actual behaviour is more complicated. It contains several components: a font dropdown, and various formatting and styling buttons. When there is no selection it shows either the font details of the text immediately to the left of the insertion cursor or the most recent format setting made since the cursor moved. When there is a selection it shows the formatting aspects the characters in the current selection have in common. When text is inserted, its format is determined by the current state of the toolbar, and the format of the current selection can be changed by changing the state of the toolbar.

The code which implements this behaviour is distributed amongst several program components: this makes it both difficult to understand and difficult to maintain. We have been investigating the extent to which the same behaviour can be expressed at a higher intentional level.

[1] The RichText control itself maintains the other main visual invariant, namely the consistency of the document being edited and the view currently being shown on the screen.

Ubiquitous Intentions

The classes of intention we have so far introduced are domain-specific or application-framework-specific. Another class of intentions, the *ubiquitous intentions* appear in many more programs. When a ubiquitous intentional form is not provided for by a particular programming language, it usually emerges as an idiomatic form. But the compiler can only check idioms for soundness with respect to the rules of a host programming language, and although this is often sufficient, sometimes there are more stringent soundness criteria for the idiom, and these *can* be checked statically only if the intention is institutionalised in some way. The traditional way of doing so is exemplified by the various preprocessors for extended dialects of Java , but as Simonyi points out, this mode of extension doesn't factor or scale well [**?**, **?**, **?**].

Some examples of ubiquitous intentions which are present in the C^{++} code of WordPad are outlined below:

- Callbacks: in a language in which all procedures are first class values there is no problem in passing procedural objects as callbacks. Any variable which is in scope at the point of abstraction of the callback may appear free in its body, and the underlying language implementation constructs closures via which such free variables are accessible.

  There is, of course, a natural translation[2] of the callback intention into strict C^{++} but it is impossible to apply this translation to a single program without applying it uniformly to the whole of MFC.

  In *C* and C^{++}, not all procedures/methods are first-class values, and various idioms have emerged to deal with the problem of accessing the values of (what would be) free variables. The most common idiom is to abstract over a single callback by using two parameters: one for a procedure address, the other for a structure which captures the values of the appropriate free variables. Static type-safety is compromised by these idioms, since free variable structures will not be uniformly typed, yet the contexts in which they are stored must have a fixed type.

  Where identifiable, such idioms should be replaced with intentional forms whose type-safety can be guaranteed statically.

- Singleton Callbacks: a procedure is defined which is going to be passed as a callback in exactly one place. The potential for error arises because the definition and use sites must be textually separate. The remedy is to realize the callback as an anonymous procedure (lambda abstraction).

[2] Using an intermediate "closure" class.

- Record Expressions: We frequently find fragments of the following form in Windows and Visual C^{++} programs:

```
{ Record r;
  r.field1=value1;
  r.field2=value2;
  ...
  procedure(... &r ...)
  ...
}
```

  The intention is to specify the values of a large collection of named parameters to a procedure. Such idioms could be replaced with intentional forms to support record construction by field specifications, especially if there were related intentional forms to support record type definitions with default field specifications.

2 Class Wizards

The code that is generated by the Class Wizards of Visual C^{++} is not simple, and in order to understand what they do, and the forms of intention that we are going to propose, we shall need to take a brief look at some of the implementation details of MFC. In MFC, as in many other application frameworks, general-purpose classes provide the essential architecture of the application. They handle the administration of implementation details (such as way in which a particular window is rendered, or the response to a particular kind of event) by default methods. An application programmer's design decisions, particularly those concerning the user interface, are realized by subclassing general-purpose classes and overriding their default methods with application-specific ones.

MFC handles event distribution by a technique, known as *message-mapping*, in which each control has a table that maps the events in which it is interested to the methods, local to the class which implements that control, that will handle them. Part of the function of the *Class Wizards* is to shield the programmer from the complexity introduced by the C^{++} implementation of message mapping.

Programmers add their own application-specific code to the boilerplate code provided by the Wizard, and the resulting program texts can be inscrutable. In particular, it can be difficult to identify those parts of the program text which express individual design decisions and even more difficult to rectify an erroneous implementation decision once the program text has started to evolve.

One source of confusion is that MFC uses both message mapping and public member invocation to communicate between objects in a program. It can sometimes be hard to see why a particular mode of communication has been chosen.

Another source of confusion arises from the fact that user interface elements are sometimes referred to directly, sometimes by pointers, and sometimes by symbolically named integer handles.

2.1 A Small Example

In order to illustrate the abovementioned difficulties concretely, we present here extracts from the code of a very small dialogue-based application built with Visual C^{++}. It maintains a display of the form:

$$text_1 \;+\; text_2 \;=\; text_3$$

Each of the *texts* represents a number and is editable. The application ensures that the arithmetic equality displayed is true by changing either $text_1$ or $text_3$ whenever editing actions by the user make it necessary to do so.

All but a few lines of the code of the application were written by Visual C^{++} Wizards. After laying out three text-edit controls using the Visual C^{++} resource editor we invented names for three variables, informed the class Wizard that the variables were to be associated with the text-editing controls, and asked that the dialogue interface be informed when any of them changes its value. We then completed the program by writing the bodies of the methods which are invoked when the text in an edit control is changed.

The whole construction process took only a few minutes, and this reflects well on the useability of Visual C^{++}. But, as we shall discuss later in some detail, the application which results has a number of flaws: some of them will be of concern to the end-user whilst others affect its maintainability and generaliseability.

The C^{++} code for the dialogue component class `CalcDlg` of this program is in three files:

- the class declaration: defines the interface and storage requirements of objects of the class.
- the class implementation: defines the methods of the class, and a static dispatch table which associates events with methods.
- the resource declaration file: `#define`s the mnemonic names for the integer handles which denote the visual elements of the interface.

First let us consider the class declaration in Figure 1. The variable declarations speak for themselves. The protected method `DoDataExchange` is present as a consequence of the variables being linked to the controls: it will be invoked whenever a request to synchronize the controls with the variables is made. The protected methods, declared below the `//Implementation` comment, are invoked in response to significant events in the life of the dialogue. The methods `OnPaint()`, `OnQueryDragIcon()`, and `OnInitDialogue()` are provided as standard, and, as we shall see in Figure 5, are given standard implementations by the Wizard. The `OnChangeEdit...` methods are invoked when the corresponding edit controls change. Their headings are written by the Wizard at the programmer's request, but it is the programmer's responsibility to implement them.[3]

`DECLARE_MESSAGE_MAP()` is a macro which expands into the declaration of a private dispatch table, which will be used to map events to the methods which handle them.

In Figure 2 we see the implementation of the dialogue constructor and its message map, together with the method `DoDataExchange` which is used to synchronize the values of the variables `m_int...` with the displayed values in the text-edit controls. The user-interface elements of the dialogue are known by mnemonics `IDC_EDIT1`, *etc.* – integer "handles" whose definitions are shown in Figure 3, and which are bound to the corresponding user interface elements by declarations made in the resource script excerpted in Figure 6 and discussed later.

[3]The prefix `afx_msg` is a null macro, presumably used as a marker for Wizards which may later read the generated code. Some of the comments inserted by the Wizard (but omitted here for brevity) contain information which can also be used to differentiate the Wizard-written code from the programmer's code.

```
class CCalcDlg : public CDialog
    {
    // Construction
    public:
            //  constructor
            CCalcDlg(CWnd* pParent = NULL);

            // Dialogue variables
            enum { IDD = IDD_CALC_DIALOG };
            long    m_int1;
            long    m_int2;
            long    m_int3;

            protected:
            virtual void DoDataExchange(CDataExchange* pDX);

    // Implementation
    protected:
            HICON m_hIcon;
            virtual BOOL OnInitDialog();
            // Dialogue messages
            afx_msg void OnPaint();
            afx_msg HCURSOR OnQueryDragIcon();
            afx_msg void OnChangeEdit1();
            afx_msg void OnChangeEdit2();
            afx_msg void OnChangeEdit3();
            // Message dispatcher
            DECLARE_MESSAGE_MAP()
    };
```

Figure 1: The Dialogue Class Declaration (`CalcDlg.h`)

```
CCalcDlg::CCalcDlg(CWnd* pParent /*=NULL*/)
        : CDialog(CCalcDlg::IDD, pParent)
{
        m_int1 = 0;
        m_int2 = 0;
        m_int3 = 0;
        m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
}

//
//      Synchronizes the controls' data with the dialogue's variables
//
void CCalcDlg::DoDataExchange(CDataExchange* pDX)
{       CDialog::DoDataExchange(pDX);
        DDX_Text(pDX, IDC_EDIT1, m_int1);
        DDX_Text(pDX, IDC_EDIT2, m_int2);
        DDX_Text(pDX, IDC_EDIT3, m_int3);
}

BEGIN_MESSAGE_MAP(CCalcDlg, CDialog)
        ON_WM_PAINT()
        ON_WM_QUERYDRAGICON()
        ON_EN_CHANGE(IDC_EDIT1, OnChangeEdit1)
        ON_EN_CHANGE(IDC_EDIT2, OnChangeEdit2)
        ON_EN_CHANGE(IDC_EDIT3, OnChangeEdit3)
END_MESSAGE_MAP()
```

Figure 2: Construction, data exchange, and message map for the dialogue (`CalcDlg.cpp`)

```
#define IDD_CALC_DIALOG                 102
#define IDR_MAINFRAME                   128
#define IDC_EDIT1                       1000
#define IDC_EDIT2                       1001
#define IDC_EDIT3                       1002
```

Figure 3: Integer handles for interface elements: (`resource.h`)

```
void CCalcDlg::OnChangeEdit1()
{       UpdateData(TRUE);
        m_int3 = m_int1 + m_int2;
        UpdateData(FALSE);
}

void CCalcDlg::OnChangeEdit2()
{       UpdateData(TRUE);
        m_int3 = m_int1 + m_int2;
        UpdateData(FALSE);
}

void CCalcDlg::OnChangeEdit3()
{       UpdateData(TRUE);
        m_int1 = m_int3 - m_int2;
        UpdateData(FALSE);
}
```

Figure 4: The essence of the program (`CalcDlg.cpp`)

In Figure 4 we see the essence of the dialogue: thebodies of the `OnChange...` methods written by the programmer. Each method is invoked when the corresponding edit box receives a change message, whereupon it sets the values of the variables from those in the edit controls, recalculates the appropriate variable's value, and synchronizes the edit controls again[4].

The remaining Wizard-generated code is shown in Figure 5. It defines the standard method used to initialize the dialogue class, and the method `OnPaint()`, which is used to repaint the real-estate it occupies when the dialogue becomes iconic.[5]

[4]The `UpdateData` method invokes `DoDataExchange` in the appropriate direction

[5]It is hard to see why these methods should be defined explicitly. Surely their standard definitions should be provided by the superclass `CDialog`, and overridden in the subclass only if necessary. `OnPaint()` is interesting, nevertheless, because it demonstrates (on its fourth line) the direct use of message passing as a means of communication between objects.

```
BOOL CCalcDlg::OnInitDialog()
{       CDialog::OnInitDialog();
        SetIcon(m_hIcon, TRUE);                 // Set big icon
        SetIcon(m_hIcon, FALSE);                // Set small icon
        // Add extra initialization here
        return TRUE;
}

void CCalcDlg::OnPaint()
{       if (IsIconic())
        {       CPaintDC dc(this); // device context for painting
                SendMessage(WM_ICONERASEBKGND, (WPARAM) dc.GetSafeHdc(), 0);

                // Center icon in client rectangle
                int cxIcon = GetSystemMetrics(SM_CXICON);
                int cyIcon = GetSystemMetrics(SM_CYICON);
                CRect rect;
                GetClientRect(&rect);
                int x = (rect.Width() - cxIcon + 1) / 2;
                int y = (rect.Height() - cyIcon + 1) / 2;

                // Draw the icon
                dc.DrawIcon(x, y, m_hIcon);
        } else CDialog::OnPaint();
}

HCURSOR CCalcDlg::OnQueryDragIcon()
{
   return (HCURSOR) m_hIcon;
}
```

Figure 5: Initialization and Painting (`CalcDlg.cpp`)

The remaining significant object generated by Visual C^{++} is the *resource script* that describes (amongst other things) the visual elements of the user interface. This is transformed by the resource compiler into a structure which is interpreted by Windows when constructing the interface. A fragment of the script is shown in Figure 6.

```
#include "resource.h"

IDD_CALC_DIALOG DIALOGEX 0, 0, 287, 36
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
EXSTYLE WS_EX_APPWINDOW
CAPTION "Calc"
FONT 8, "MS Sans Serif"
BEGIN
    EDITTEXT        IDC_EDIT1,59,13,40,14,ES_AUTOHSCROLL
    EDITTEXT        IDC_EDIT2,123,13,40,14,ES_AUTOHSCROLL
    EDITTEXT        IDC_EDIT3,187,13,40,14,ES_AUTOHSCROLL
    LTEXT           "+",IDC_STATIC,107,16,8,8
    LTEXT           "=",IDC_STATIC,171,16,8,8
END
```

Figure 6: A fragment of the resource file (`Calc.rc`)

Resource scripts are also used to describe the interfaces of programs with menus and toolbars. In general they (are thought to) make life easier for the developer because they separate descriptions of the appearance of user interface elements from descriptions of their behaviour. A change in the appearance of an interface element (for example, the legend on a button, the language of an error message, or the position of an icon) is easily accomplished without changing the behaviour of the program.

2.2 Analysis

The places where the program described above departs from the informal specification we gave may at first sight seem superficial. But sometimes *superficial symptoms* are indicators of more fundamental problems. In this case they will lead us to problems of the kind which can dramatically affect the maintainability and generaliseability of more complex applications.

Symptom: The text controls behave inappropriately. If any of the texts is completely erased, then the application puts up a (modal) dialogue box warning that it no longer represents a number, but the empty text is treated as if it represents zero until the situation is remedied. Thus a user who wants to change one number to another that has no digits in common with it may not do so by simply erasing the first.

What seems to be wrong here is that the standard validation provided for the number entry text control is too eager – the text is inspected whenever it changes. One remedy (in the context of MFC) would be to use our own code, instead of the stock DDX code, to maintain the association between the text of each control and the number it represents. Leaving aside the logistical difficulties in doing this[6], the fact remains that the feature the three text controls have in common (that they represent integers

[6]The programming environment needs to be locally customized.

when they represent anything at all) would not be articulated concisely and explicitly in the program text.

What's needed here, surely, is a component whose text must be numeric when it is nonempty. So why could we not design such a component as an independent entity, and put three instances in the application? Such a component, though easy to build, has to be made available to the programming environment at design time if this is to be done. Although this isn't particularly hard, it seems rather inelegant to have to customize the programming environment simply in order to be able to make a small number of copies of (what should be) a very lightweight component.

Diagnosis: The fundamental problem here is that there is no lightweight and compositional notation for describing the *behaviour* of composite controls in terms of their components.

Symptom: The dialogue has a fixed geometry. The precise position and size of everything in the dialogue box is fixed at *dialogue design time* (relative to the main font size used in the dialogue, which is also fixed at design time). Should the user wish to work with enormous numbers, then there is no way that the dialogue can be stretched at runtime to provide room for more digits – the user simply has to put up with whatever scrolling facilities the basic text control provides. Transpose the nonstretchability to (say) a file-choice dialogue box opened in a well-populated directory and you can see why this should not be dismissed as a merely superficial annoyance.

Diagnosis: In this case the fundamental problem is that there is no lightweight and compositional notation for describing the *appearance* of components, let alone doing so in a way that supports late binding of layout and/or dimensions.

3 Towards an Intentional Framework for User Interfaces

In this section we sketch informally, and by example, an *intentional framework*[7] which we hope will be useful for describing user interfaces of the kind that are presently built using MFC through Visual C^{++}. Although it was "mined" from Windows/Visual C^{++}/MFC, we have tried to design the framework so that interface descriptions can be decoupled from the specificities of the operating system and user-interface platform on which they are to be implemented. Our aim is to isolate and treat such platform dependencies as a fixed part of the implementation of the framework, rather than leaving them ubiquitous and implicit. The feasibility of this goal remains to be tested in practice, and we are currently constructing and evaluating prototype implementations.

We begin by giving a brief account of our model of a user interface. Next we describe the intended behaviour of the calculator dialogue described above by providing an explicit and lightweight description of a numeric field editor control, and showing how the calculator dialogue can be described by gluing three of these components together. We continue by giving an account of some ways in which the appearances of components can be described, and lastly we give examples of appearance- and behaviour- preserving transformations which could be used to transform composite components described in the style outlined here into components more directly in keeping with the style of MFC.

The concrete syntax used in all these examples is somewhat tentative. Whilst attempting to preserve the notational flavour of C^{++} for describing algorithmic content, we find ourselves drawn to a keyword styled approach to describing classes. For the moment we hope this will enable us to rely on our readers' goodwill and intuition in approaching the semantics of the notation: later we will have to be more precise.

3.1 Describing Behaviours

Our model of a user interface is as an hierarchical collection of named components[8] with local state operating in an environment. Components can send directed messages to other named components or undirected messages to the environment. In the latter case the environment is responsible for discovering an appropriate component to which the message can be delivered. The way in which it does this depends on the nature of the user interface. We will discuss environments in more detail later.

As well as sending messages, a component may inspect the *public attributes* or invoke the *public operations* of other named components. A component declares its willingness to accept delivery of particular kinds of message from the environment, and also declares the names (and the type of content) of the messages which it will send. The essential difference between invoking a public operation in a named component and sending a message to that same component is that in the former case the operation is executed immediately, whereas in the latter case the message may be delayed by the environment.[9]

[7] An *intentional framework* is a coherent collection of language constructs designed to support the implementation of program components for a specific problem domain.

[8] Or, if you will, *objects*.

[9] This difference seems to be fundamental to the architecture of MFC-built user interfaces, which behave as discrete event simulations (rather than truly concurrent systems). It is worth noting the additional complexity induced by the recent grafting of concurrent threads onto this architecture.

A *composite* component (*i.e.* a component which has subcomponents) is responsible for handling *all* the undirected output messages of its own subcomponents. By convention these are simply suppressed by the environment unless they are handled explictly by the composite itself.

3.1.1 The `NumberEditor` control

The class of `NumberEditor` control is described in Figure 7. Each control of this kind consists of a `Text` control, and makes public the attributes `valid` and `value`, and the operations `SetValue` and `SetValid`. It can send the message `CHANGE` into the environment.

The two attributes `value` and `valid` are invariantly related to the text shown by `t` as follows:

- When `valid` is true, the text shown by `t` represents the `value`, and its colour scheme is normal.
- When `valid` is false, the text shown by `t` need not represent any value, and its colour scheme warns that it doesn't.

Whenever the text control component changes, the corresponding notification message is handled by the number editor, which first acts to preserve the invariant relationship outlined above, and then notifies the environment of the change.

Discussion

1. The component name `t` denotes a text editing control throughout its scope. Just *how* it represents it (by handle, by pointer, or otherwise) for a particular instance of this kind of control is a matter of implementation detail which may depend on the context in which the instance is used.
2. The scope of component, attribute, and operation names is the whole of the executable body of the class, *i.e.* its operations, its initialisation statements, and its event-handling clauses. An attribute or operation named n of a control c is referred to by the qualified name $c.n$. The components of a control may not be referred to outside its definition.
3. The public operations of the text control within a number editor may not be invoked from outside the editor. This is remedied by adding a *delegation* clause of the following form

   ```
   DELEGATES
     OPERATIONS OF Text TO t
   ```

   Likewise, if we wished to have a number editor share all the public attributes of its text control component, then we would write the delegation clause:

   ```
   DELEGATES
     OPERATIONS OF Text TO t
     ATTRIBUTES OF Text TO t
   ```

   Whether this delegation is *implemented* by true delegation or by subclassing is an implementation detail which may depend on the context in which instances of the control are used. The point of the delegation is that the `NumberEditor` control now offers the public operations (and attributes) of a `Text` editing control, and these are implemented by the (hidden) component `t`. Delegation is, incidentally, a useful intention to add to any language which supports modular descriptions of behaviour.

```
CONTROL CLASS NumberEditor
{ COMPONENTS
        Text  t
  MESSAGES
        CHANGE
  PUBLIC ATTRIBUTES
        int  value
        bool valid
  INITIALLY(int v)
        { PrivateSetValue(v) }
  PRIVATE OPERATIONS
        void PrivateSetValue(int v)
        { value = v
          t.SetText(IntegerToText(v))
          SetValid(true)
        }
  PUBLIC OPERATIONS
        void SetValue(int v)  { if (value != v) PrivateSetValue(v)  }

        void SetValid(bool validity)
        { valid = validity
          if (valid) { t.SetColourScheme(ColourSchemes.Normal) }
          else       { t.SetColourScheme(ColourSchemes.Warning) }
        }
  INTERNAL EVENTS
        WHEN t.CHANGE
        { String s = t.getText()
          if (isInteger(s))  { value = TextToInteger(s); SetValid(true) }
          else               { SetValid(false) }
          SEND CHANGE
        }
}
```

Figure 7: The `NumberEditor` Control Class

4. An important assumption we have made is that the messages sent by the components of a composite control should be handled by the control itself, and that they don't automatically percolate up the containment hierarchy.

   Here we have said nothing about messages other than `CHANGE` which might be sent by the component control `t`. The scope rules make it impossible to refer to them outside the body of the number editor, but the following clause, whose meaning is dual to the meaning of `DELEGATES`, would mandate the forwarding of messages (other than `CHANGE`) from `t` as if they came from the number editor itself.

   ```
   FORWARDS
      EVENTS OF Text FROM t
   ```

   Experience shows that many composite components are designed to emulate virtually the complete behaviour of one (or more) of their components, but as we shall eventually explain in detail, neither delegation nor forwarding need necessarily be "wholesale".

3.1.2 The Calculator Dialogue

The calculator dialogue is described in Figure 8. Each dialogue of this kind consists of three `NumberEditor` controls and an "ok" button. Its behaviour (but not its appearance) is described below. To make the situation a little more interesting we have described the dialogue as if it were a component in its own right.

Discussion

1. The public attributes `valid` and `value` are simply those of `edit3`. The point we make with this example is simply that attributes need not be simple variables, they can also be calculated. A more concise way of achieving the same ends in this particular case would be to delegate the calculation of the attributes

   ```
   DELEGATES
     ATTRIBUTES
      int  value TO edit3
      bool valid TO edit3
   ```

2. The dialogue is prepared to accept messages of the form `INVOKED(v)`, (where `v` is an integer) from its environment. It responds by (re)setting the values of its components and making itself visible.

3. When the `ok` button is pressed, the dialogue sends a message (up the hierarchy) indicating dialogue completion and validity of the internal value, and then makes itself invisible. It would be better for the dialogue to return a value only if the number is valid, and otherwise give some feedback.

   ```
   WHEN ok.COMMAND
   { if (valid)
     { SEND DIALOGUERETURNS(value)
       MakeInvisible()
     } else { Beep() }
   }
   ```

```
 DIALOGUE CLASS Calculator
 { COMPONENTS
        NumberEditor edit1  (0)
        NumberEditor edit2  (0)
        NumberEditor edit3  (0)
        Button       ok
   MESSAGES
        DIALOGUECOMPLETE(bool, int)
   PUBLIC ATTRIBUTES
        int  value  { return edit3.value }
        bool valid  { return edit3.valid }
   INTERNAL EVENTS
        WHEN edit1.CHANGE OR edit2.CHANGE
        { if (edit1.valid && edit2.valid)
          { edit3.SetValue(edit1.value+edit2.value) }
          else
          { edit3.SetValid(false) }
        }

        WHEN edit3.CHANGE
        { if (edit3.valid && edit2.valid)
          { edit1.setValue(edit3.value-edit2.value) }
          else
          if (edit3.valid && edit1.valid)
          { edit2.setValue(edit3.value-edit1.value) }
        }

        WHEN ok.COMMAND
        { SEND DIALOGUECOMPLETE(valid, value)
          MakeInvisible()
        }
   PUBLIC EVENTS
        WHEN INVOKED(int v)
        { edit1.SetValue(v)
          edit2.SetValue(0)
          edit3.SetValue(v)
          MakeVisible()
        }
}
```

Figure 8: The `Calculator` Dialogue

In this case the MESSAGES clause of the dialogue description would be

```
MESSAGES
   DIALOGUERETURNS(int)
```

4. An alternative way of binding the COMMAND event of the ok button to the code which handles it would have been to write its declaration as follows, whilst omitting the WHEN ok.COMMAND declaration:

```
Button ok
{ EVENTS
    WHEN COMMAND { if (Calculator.valid) .... }
}
```

Notice that the body of the event handler uses the attribute valid declared in the Calculator class, and that this is stated explicitly. If Button objects were defined with a public attribute of the same name then the variable valid would refer to *that* attribute, and we need some way of making it clear (to the reader as well as to the compiler) just which valid attribute is being referred to.

We could also have omitted the WHEN edit3.CHANGE declaration, had we declared

```
NumberEditor edit3
{ EVENTS
    WHEN CHANGE
    { if (edit3.valid && edit2.valid) ....
      else
      if (edit3.valid && edit1.valid) ....
    }
}
```

The qualified form of attribute name edit3.valid is necessary in this case because both NumberEditor and Calculator classes declare public attributes with the same name.

3.2 Constant Attributes

Some component features are specified at design-time and are thereafter constant. For example a standard dialogue box has constant features which include its caption, whether its frame is to be automatically decorated with a system menu, and whether it is to be a modal dialogue.[10]

Such attributes are declared as part of an ATTRIBUTES declaration, using the type modifier CONSTANT. For the example in question, we might add the following feature declaration to the definition of Calculator:[11]

```
PRIVATE ATTRIBUTES
  CONSTANT int STYLE = DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
  CONSTANT String CAPTION = "Calc"
```

Some constant attributes of a component may be specified only when the component is declared: these are the INITIAL ATTRIBUTES. For example, adding the following clause to the definition of NumberEditor gives it initial attributes named HORIZONTALSCROLL and VERTICALSCROLL which are false by default.

[10] Of course the code bodies of operations and event handlers are also examples of design-time constants.

[11] *c.f.* Figure 6

```
INITIAL ATTRIBUTES
   bool HORIZONTALSCROLL = false,
        VERTICALSCROLL   = false
```

By changing the declaration of `Text t` within the same class we can set the scrolling bits of its `FLAGS` initial attribute appropriately.

```
Text t
{ ATTRIBUTES
        FLAGS = HORIZONTALSCROLL?ES_AUTOHSCROLL:0 + VERTICALSCROLL?ES_AUTOVSCROLL:0
}
```

The detailed layout and appearance of a component may also be a design-time constant attribute, though in some cases it may be better to prescribe layout *strategy* at design time, and have the strategy implemented as the components run. We deal with these matters at greater length in the next section.

3.3 Layout and Appearance

The layout and appearance of a composite is specified by the `LAYOUT` section of its description. This consists of one or more clauses, each of which specifies the layout for a particular locale by associating the name of the local with a *layout expression*.

We favour a declarative style for the specification of layout and appearance. The key idea is that the subcomponents of a composite are located in containers of various sorts, and that containers may be grouped within other containers. Some containers merely provide some sort of decoration for their content, whilst others specify, either implicitly (by constraint) or explicitly, the detailed positions of the components or containers they hold.

For the sake of concreteness in this note we have adopted a rather basic repertoire of container types, but new types of container behaviour can easily be specified. For example, the appearance of the Calculator dialogue might be specified by the following `LAYOUT` section

```
LAYOUT
{ US.English =
  Column {Font="TimesRoman", Size=12pt}
  (     Row (edit1 "+" edit2 "=" edit3)
        Row (FILL  ok{caption="OK"})
  ),
  UK.English =
  Frame  {linewidth=.01pt}
  (
   Column {Font="ChurchillSansCigar", Size=14pt}
   (     Row  (edit3 "=" edit1 "+" edit2)
         Row  (ok {caption="Yes, please!"} FILL)
   )
  )
}
```

Discussion

1. A *layout expression* may be *simple* – in which case it is a component specification, a glue specification, or a text specification, or it may be *composite*, in which case it consists of a container

type, a (possible empty) list of container properties specified by keyword, and a (nonempty) sequence of layout expressions (written in parentheses), which constitute its content.

A *component specification* is the name of a component, together with a possibly empty list of its properties (specified by keyword).

A *text specification* is a literal string, together with a possibly empty list of its properties (specified by keyword).

2. Our basic layout model is similar to the familiar boxes and glue model popularised by Donald Knuth. Each container has natural dimensions (which may be overridden by specifying them explicitly as properties). Siblings within the same container are separated by blobs of *glue* which have a minimum size and a "stretchability".[12]

   Unless otherwise specified every container takes on its natural dimensions, and in any case may never take on smaller dimensions. For example, the natural width of a column is the largest of the natural widths of its content. Its natural height is the sum of the natural heights of its contents and the associated glue. Symmetrically, the natural height of a row is the largest of the natural heights of its contents, and its natural width is the sum of the natural widths of its contents and the associated glue.

3. A container (or component) may be expandable vertically, horizontally, or in both dimensions. When it is stretchable it grows in the appropriate direction to fit its enclosing container: the extra space is distributed amongst any blobs of glue it contains in proportion to their stretchability.

   Unless otherwise specified (as one of its properties), a row is expandable horizontally but not vertically, and a column is expandable vertically but not horizontally.

3.4 Menus and Toolbars

We have extended the intentional framework to support the description of menu-bars, menus, and toolbars. We will report on this part of the work in the next edition of this note.

3.5 Behaviour-Preserving Transformations

The intentional framework we have described above could be implemented (for the most part) by translating component class descriptions into classes, implementing delegation literally (*i.e. by delegation*), and translating layout descriptions into a combination of resource files and invocations of geometry management tools. But there may be situations in which to do so would lead to what some might consider unacceptable inefficiencies in the resulting implementation – consequences either of the length of delegation chains or of vtable bulk.

In this section we present a collection of simple transformations which may be used to make a composite component (called the host) more monolithic and more efficient, whilst preserving its appearance and observable behaviour. The main transformation unfolds the variables of each of the host's components into the host class. Two "rectifying" transformations then lift into the host the methods and

[12] `FILL` is the kind of glue which has highest stretchability.

message handling code (private and public) of each distinct component class which is used within the host.

These transformations yield components which will eventually translate to C++ code similar in style to that which might be produced by a Visual C++ Wizard. The outcome of applying them to the code of the calculator dialogue of Figure 8 is shown in Figures 9 and 10.

Discussion

1. The variable-lifting transformation replaces each instance of `NumberEditor` with an analogously-named instance of its component `Text`, and introduces new variables which corresponds to each of its variables.

2. The methods of `NumberEditor` are lifted into the host, as private methods with an extended parameter set with reference parameters corresponding to each subcomponent and variable of the component. Their bodies are retained intact, and calls to them are transformed appropriately.

3. The `NumberEditor CHANGE` event body is lifted into the host as a method, and appropriate calls are then used to prefix the event bodies of the host. Notice that the disjunctive specification of the host's first event trigger has to be expanded into to distinct events in order to do this.

The simplistic transformations discussed above can lead to code duplication, a problem which can be avoided by using more sophisticated *object-centred* representations of components in the host language.

```
DIALOGUE CLASS Calculator
{ COMPONENTS
       Text    edit1, edit2, edit3
       Button  ok
  MESSAGES
       DIALOGUECOMPLETE(bool, int)
  PUBLIC ATTRIBUTES
       int  value  { return edit3_value }
       bool valid  { return edit3_valid }
  PRIVATE ATTRIBUTES
       int  edit1_value, edit2_value, edit3_value
       bool edit1_valid, edit2_valid, edit3_valid
  PRIVATE OPERATIONS
       void NumberEditor_PrivateSetValue(int v, Text t, int& value, bool& valid)
       { value = v
         t.SetText(IntegerToText(v))
         NumberEditor_SetValid(true, t, value, valid)
       }

       void NumberEditor_SetValue(int v, Text t, int& value, bool& valid)
       { if (value != v) NumberEditor_PrivateSetValue(v, t, value, valid)
       }

       void NumberEditor_SetValid(bool validity, Text t, int& value, bool& valid)
       { valid = validity
         if (valid) { t.SetColourScheme(ColourSchemes.Normal) }
         else       { t.SetColourScheme(ColourSchemes.Warning) }
       }

       void WHEN_BODY1()
       {
         if (edit1_valid && edit2_valid)
         { NumberEditor_SetValue(edit1.value+edit2.value, edit1, edit1_value, edit1_valid) }

         else
         { NumberEditor_SetValid(false, edit1, edit1_value, edit1_valid) }
       }

       void WHEN_BODY2()
       {
         if (edit3_valid && edit2_valid)
         { NumberEditor_SetValue(edit3.value-edit2.value, edit1, edit1_value, edit1_valid) }

         else
         if (edit3.valid && edit1.valid)
         { NumberEditor_SetValue(edit3.value-edit1.value, edit2, edit2_value, edit2_valid) }

       }

       ...
```

Figure 9: The Transformed `Calculator` Dialogue (Part 1)

```
     ...

 INTERNAL EVENTS
     WHEN edit1.CHANGE
     { NumberEditor_CHANGE_EVENT(edit1, edit1_value, edit1_valid)
       WHEN_BODY1()
     }

     WHEN edit2.CHANGE
     { NumberEditor_CHANGE_EVENT(edit2, edit2_value, edit2_valid)
       WHEN_BODY1()
     }

     WHEN edit3.CHANGE
     { NumberEditor_CHANGE_EVENT(edit3, edit3_value, edit3_valid)
       WHEN_BODY2()
     }

     WHEN ok.COMMAND
     { SEND DIALOGUECOMPLETE(valid, value)
       MakeInvisible()
     }

 PUBLIC EVENTS
     WHEN INVOKED(int v)
     { NumberEditor_SetValue(v, edit1, edit1_value, edit1_valid)
       NumberEditor_SetValue(0, edit2, edit2_value, edit2_valid)
       NumberEditor_SetValue(v, edit3, edit3_value, edit3_valid)
       MakeVisible()
     }
}
```

Figure 10: The Transformed `Calculator` Dialogue (Part 2)

4 Maintaining Visual Invariants

When a relation between one or more components of a program is to be maintained invariantly true, it is sensible to group together those aspects of the program's code which are responsible for re-establishing the invariant when one of the components changes.

Grouping code in this way can cut across the *natural* process structure of the program. Consider, for example, the Model-View-Controller design pattern. The essence of this pattern is that a controller process is responsible for co-ordinating the visual invariant between an underlying model and one or more views of that model. One might expect to find model and view processes designed independently of the controller process, but in our experience considerations of efficiency and functionality frequently give rise to modifications (by subclassing) of the idealised model and view processes. These modifications have to be specifically designed to accomodate the needs of the controller process that maintains the invariant. In these circumstances the original intention of maintaining a visual invariant between model and view gives rise to code in three different classes. It's hard to discover the original intention by inspecting the classes, and program maintenance becomes correspondingly difficult.

Our goal is to build an intentional framework to support the gathering in one place of all code related to a single visual invariant. We have been influenced here by work on *Subject Oriented Programming*[**?**] and *Aspect Oriented Programming*[**?**].

This part of the work is incomplete.

5 Discussion

There has been a tension in our work between, on the one hand, the desire to restructure "intentionally" an existing program, built on top of an existing framework and language (MFC and Visual C^{++}), and, on the other hand, the desire to discover domain-specific intentions which are sufficiently abstract to be useful outside that framework.

It is arguable that the explicit distinction between events, and operations in the framework outlined in the previous section is artificial – an uneccessary throwback to its origins in MFC which confuses both readers and writers.

We are presently investigating a more abstract framework in which this distinction is removed. At the top-level of description of a component in this more abstract setting there are just *entries*. An entry may eventually be realized as an event-handler or as an operation, but we would like to defer making that decision. The way we make it in any particular case may depend on details of the underlying implementation language and/or the underlying user-interface toolkit.

Appendix: Implementation Considerations

The present document leaves many details unspecified, particularly details of the environment (which deals with the routing of messages).

We are presently clarifying our ideas by experimenting with translation schemes which yield "Petzold style" C implementations of components.

Acknowledgements

Our principal debt is to Charles Simonyi, whose manifestos on Intentional Programming inspired the project of which this work is just a small part. Our colleagues in the Programming Tools Group (Quentin Miller, Oege de Moor, Ivan Sanabria, Ganesh Sittampalan, Mike Spivey, Eric van Wyk) have provided a challenging and supportive environment for us.

Kevin Backhouse is recipient of a scholarship generously funded by Microsoft Research.