Windowtools

Tools for equipping programs with graphical user interfaces

Bjørn Kirkerud
Department of Informatics, University of Oslo, Norway
email: bjornk@ifi.uio.no

1 Introduction

An introduction to a special section on graphical user interfaces in the April 1993 issue of the Communications of the ACM [7] opens as follows:

> *The Graphical User Interface (GUI) is one of the most revolutionary changes to occur in the evolution of modern computer systems. In the space of less than 10 years the expectation of what the interaction between human and computer would be like has changed from a terse, character-oriented exchange modeled on the teletypewriter to the now familiar Windows, Icons, Menus and Pointing device WIMP interface.*

If this is right, most programs that are written should be equipped with graphical user interfaces. This is, however, not the case. The education in programming is in particular lacking in this respect: Most courses in programming that I know of, do not teach the students anything about the design and implementation of graphical user interfaces, and the students do not learn how to design programs that satisfy the recently evolved expectation of how programs should interact with users. An example is my own textbook [3] which does not mention graphical user interfaces; my excuse is that the book was written five years ago.

Partly to ameliorate these matters at least for myself and my students, I have developed and implemented Windowtools. My goal was to construct tools – classes and procedures – that are so simple that they could be used in introductory courses in programming where Simula is used as teaching-language, but also so powerful that they would be suitable for equipping "real" programs with graphical user interfaces.

This paper contains two main sections. In the first of these, section 2, I claim that it is difficult to build graphical user interfaces and present three reasons why I believe this is the case. Section 3 contains some examples that illustrate the use of some of Windowtools. The paper ends with a short section on the availability of Windowtools.

2 Why it is difficult to build graphical user interfaces

After years of experience in constructing programs with graphical user interfaces, I have reached the following conclusion:

> *It is difficult to design and implement graphical user interfaces!*

As far as I can see, there are three kinds of difficulties:

2.1 Complexity of window-packages

Many (most?) available packages of tools for creating graphical user interfaces are large, complex and generally badly structured. Such packages, often called *window systems* or *toolkits*, frequently give the impression of having been created by and for people who have intricate programming in general, and window-programming in particular, as a lifestyle. They are, in my opinion, not well suited for those of us who wish to construct programs in which the user interface is just one part among many, and who do not want to spend, unnecessarily, a major effort in trying to understand the often arcane intricacies that the designers of these systems must mean should be mastered by anyone attempting to implement a graphical user interface.

A well-known (and, I fear, characteristic) window system, commonly used on UNIX-systems, is X (described for instance in [6]), which contains a very large and bewildering set of functions, macros, subroutines, structures, so-called widgets, protocols, codes and names. [6] consists of 8 volumes, each containing many hundred pages. The first program described in volume 1 (a programming manual, with 635 pages, for what is called Xlib, which is the basic part of X) is an "hello-world" program. More than 6 pages are necessary just to print the code of this program! It is, of course, possible to make working X-programs that are shorter than 6 pages. When I nonetheless mention this program, which – not surprisingly – is quite difficult to understand, it is because it is the first and also most simple program presented in the manual. A reader will therefore quite certainly get the impression that such complexity is necessary in any program with a graphical user interface, and that the concepts and subroutines used in the program comprise a minimal or at least recommended set.

One of my goals in making Windowtools has been to make it possible for programmers to equip their programs with graphical user interfaces without having to understand very many new concepts nor to remember how to use a very large number of subroutines and other tools.

2.2 Lack of experience in designing graphical user interfaces

Conventional interaction between user and program takes place in the command window, which in this connection may be thought of as a single half-duplex communication channel used to exchange characters. Programs with graphical user interfaces do normally not use the command window for interaction, but rather a collection of specialized windows of various kinds. Some of these may be used by the program to display text or figures, others by the user to input data and control the program-execution (by moving the mouse, clicking mouse-buttons and hitting keys). The interaction between a user and a program with graphical user interfaces may hence be considered to take place through a set of communication channels, some of which are simplex and others half-duplex. Some channels (windows) may even be full duplex (i.e., used simultaneously by program and user).

A programmer may encounter two kinds of difficulties when designing a graphical user interface. The first lies in deciding which components to include in the interface and in determining what functions each component should have; the second in designing and understanding the often complex patterns of communication that may occur between a program and its users in the channels that constitute the interface. [1], which is an IBM-publication about the design of graphical user interfaces, puts forward two principles for such design. The first of these says that *the user should be able to develop a conceptual model of the interface.* To achieve this goal, the designer must select suitable components – with appropriate capabilities – for inclusion in the interface. The second of the design principles says that *the user should be in control of the dialog.* One of the problems in achieving this, lies in designing suitable patterns of communication between user and program.

Many of today's programmers have constructed only character-oriented user-interfaces modeled on the teletypewriter. They may, of course, have *used* programs with graphical user interfaces, but often have little or no experience in *designing* such interfaces. They may therefore find it quite difficult to follow the principles put forward in [1] and to achieve the goals that justify these principles.

One of my goals with Windowtools has been to make it easy for a designer of graphical user interfaces to experiment with different kinds of components and different patterns of communication; another goal has been to enable students to get varied experience in designing and using such interfaces. Much experience in these matters seems to be necessary before one is able to design good interfaces.

2.3 Lack of experience in designing event-driven programs

The imperatives of conventional programs are executed sequentially, one after another in the sequence given in the program-text. This is usually not the case for programs with graphical user interfaces, which may be *event-driven*. This means that certain of their actions may be executed as a result of externally generated events (an *event* is what occurs for instance when the user clicks a button, moves the mouse or presses a key). Event-driven programs must be able to handle any relevant event that may occur, in the order – often quite unpredictable – determined by the user. It may even be the case that the handling of one event may be interrupted by the occurrence of another event and also that several actions may execute simultaneously. A very simple example of the latter is a program that allows the user to set up several simultaneous stopwatches.

That programs with graphical user interfaces normally are non-sequential, entails a partial breakdown in the simple sequential model for program-execution which most programmers are used to and which they depend upon when trying to understand what happens in their programs, in particular when debugging. It should therefore not come as a surprise that many programmers with long experience in constructing sequential programs, may encounter great difficulties when trying to construct or even understand event-driven programs.

```
begin
   external class Windowtools;
   inspect new MainWindow("Windowtools") do
      begin
         SetSize(200, 100);
         Show;
         DrawCenterText("Hello world!");
         MakeButton("Quit").Wait;
      end;
end
```

Result:

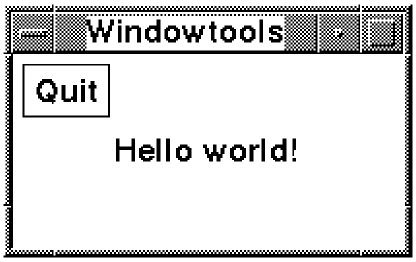

Figure 1: A program that waits for a click in a button

One reason for my desire to include design and implementation of graphical user interfaces in early courses in programming – and hence also a goal in the design of Windowtools – is to give the students, as early as possible, experience in designing, implementing and debugging event-driven programs in general. At a recent conference Allan Kay claimed [2] that the ability to construct event-driven programs will be much more important in the future than it is today. I believe this is right, and that it is important that students learn about the design and implementation of such programs quite early; – before their minds are molded in sequential folds. Programming event-driven graphical user interfaces seems to be a fruitful way to learn general ideas about the design and implementation of event-driven programs.

3 Some simple examples

This section contains some examples that show how to use a few of the classes and procedures in Windowtools to construct programs with graphical user interfaces. The examples, which are intended to illustrate different kinds of interaction between a program and its users, have quite simple data models and use very few components in their graphical user interfaces. It should, however, not be difficult to use the examples as starting-points for constructing programs with more complex data models and user interfaces.

3.1 Waiting for a click in a button

The first example, shown in figure 1, creates a window, writes some text in it, creates a so-called button and waits until the user clicks in the button. The following tools from Windowtools have been used in this program:

- The effect of **external class** Windowtools is to import every class and procedure in Windowtools and make them available for use in the program.
- MainWindow is one of the classes in Windowtools. The effect of executing **new** MainWindow(<*window-name*>) is to create a new window-object. A visible mani-

festation of this object will appear on the screen as soon as it receives the message Show. A *title-bar* containing the specified window-name is placed above the window. (The titlebar is included in figure 1, but not in the remaining figures.)

- The class MainWindow (and other Windowtools-classes that define windows) contains procedures that may be used to specify the size and position of windows, to write text and draw lines in windows, to create subwindows of various kinds, etc. Three of these procedures are SetSize, Show and DrawCenterText, which work as follows: SetSize(<*width*>, <*height*>) defines the size of a window that receives this message (the specified width and height are in pixels), Show is sent to a window to make it visible, and DrawCenterText(<*text*>) writes the specified text in the center of the window that receives the message.

- A *button-window* (often shortened to *button*) is a specialized window – usually containing a text – in which the user may *click*, i.e., press and release one of the buttons on the mouse after having used the mouse to move the cursor into the button-window. Most programs with graphical user interfaces create one or more buttons, each of which is associated with an action the program is able to perform. The association is such that a click in a button has the effect that its associated action is executed.

  Button-windows are defined by a *Windowtools*-class named Button. An instance of this class may be created by using the **ref**(Button)-function MakeButton, which is available in MainWindow (and other window-classes). The effect of executing wnd.MakeButton(<*button-text*>) is to create a button with the text <*button-text*> and place it in the window wnd, – which becomes the *owner* of the button.

  One of the procedures in Button is Wait. The effect of sending the message Wait to a button is that the program waits passively until the user clicks the button.

3.2 Choosing by clicking in buttons

The program in figure 1 only allows the user to decide when the program shall terminate. In the next example, which may be found in figure 2, the user is given greater control: he or she may repeatedly choose which of several actions the program shall execute. The following new parts of Windowtools are used in this program:

- The message ClickInButton(<*button*>) is sent – by the Windowtools-system – to a window whenever a button owned by that window has been clicked. ClickInButton is a procedure which is virtual in the Windowtools-classes that define windows (MainWindow is one of these classes). This means that if a subclass of one these window-classes contains its own version of ClickInButton – as is the case for LineControl – then it is this version which will be invoked when a button owned by a window in the subclass is clicked.

- A *prompt-window* is a special window which may be used to prompt the user for text and numbers. The class PromptWindow defines such windows, and contains various procedures that may be used to ask for and read input from the user. One of these procedures is the function ask_for_int. The effect of evaluating

```
begin
  external class Windowtools;
  MainWindow class LineControl;
    begin integer seed;
      ref(PromptWindow) prompt_wnd;
      ref(Button) draw_button, clear_button;
      procedure ClickInButton(b); ref(Button) b;
        if b == draw_button then Drawlines else
        if b == clear_button then Clear;
      procedure Drawlines;
        begin integer count, line;
          count := prompt_wnd.ask_for_int("How many lines?");
          for line := 1 step 1 until count do
            DrawLine(width//2, height//2,
                    randint(1, width, seed), randint(1, height, seed));
        end;
      seed := 123873;
      SetSize(450, 250); Show;
      prompt_wnd :- MakePromptWindow;
      clear_button :- MakeButton("Draw lines");
      draw_button :- MakeButton("Clear window");
      MakeButton("Stop").Wait;
    end LineControl;
  new LineControl("Drawlines");
end
```

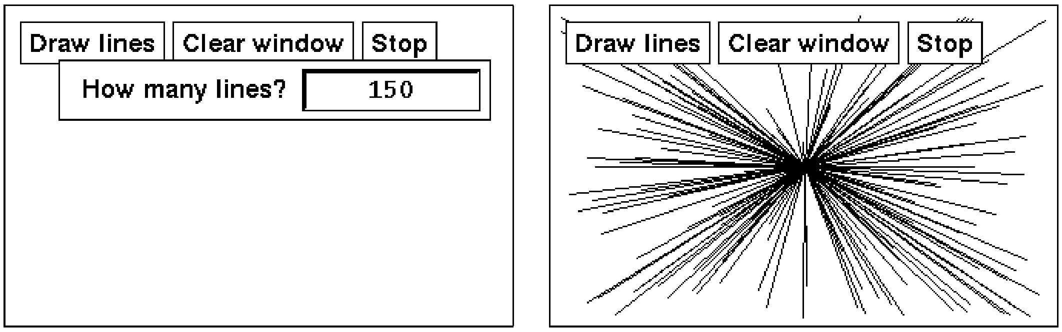

Figure 2: A program controlled by button-clicks

prompt_window.ask_for_int(<*prompt-text*>), is to make the prompt_window visible with the specified text written in its left-hand part and an empty *answer-field* in its right-hand part. The user may then type his or her input in the answer-field, and hit the return-key to indicate that the input is ready.

A prompt-window may be created by evaluating wnd.MakePromptWindow, where wnd is a window that becomes the owner of the prompt-window.

- The effect of sending the message DrawLine(x1, y1, x2, y2) to a window is that a line is drawn in the window from (x1, y1) to (x2, y2). The integers x1, y1, x2 and y2 are given in pixels, with (0, 0) in the upper left-hand corner of the window.
- The integer functions width and height may be used to ask a window for its size.
- When a window receives the message Clear, any lines or text drawn or written in the window are removed.

The program in figure 2 is a very simple example of an event-driven program (see section 2.3).

3.3 Interrupting actions

Assume that a user of the program in figure 2 commands the program to draw a very large number of lines. The program will then keep on drawing lines for quite a while, possibly putting an unendurable strain on the patience of the user, who may therefore wish to terminate the drawing. The program does not, however, give the user any means by which to achieve this: The loop in Drawlines will not terminate before every line specified by the user has been drawn, and the only way the user may stop the drawing, is by brutally terminating the program completely (this may achieved by sending a command to kill the job to the operating system, for instance by typing a special character – control-c on some systems – in the command window). But there is no way the user of the program in figure 2 may interrupt and stop the action specified in Drawlines and then bid the program to continue with something else. Clicking a button is of no avail: the click will be registered, but ClickInButton will not be invoked before Drawlines has finished its task.

The problem of not being able to interrupt an action during its execution is a general problem that may be encountered when using any program that employs a mode of interaction similar to the one used by the program in figure 2. Such programs execute one action at a time, and will always finish each action before they allow the user to start another. It is not, however, difficult to use Windowtools to construct programs with *interruptable* actions. An example is given in figure 3, which employs the following new Windowtools-concepts:

- Whenever the user presses a key, moves the mouse or clicks a mouse button, an *event* occurs. All such user-initiated events (and some other events not directly initiated by the user) entail that suitable *event-descriptors* are put in an *event-queue* where they stay until handled, one by one, by special Windowtools-procedures.
- The most important of these procedures is handle_event. The result of invoking this procedure, is that the event at the head of the event-queue is handled in an appropriate manner. Should the queue be empty, handle_event waits passively until an event occurs.

  What handle_event does with an event, depends upon what kind of event it is. For most kinds, handle_event sends a suitable message – it invokes a virtual procedure that handles events of that kind – to the window in which the event occurred. If, for example, the user clicks a button-window, handle_event invokes ClickInButton in the window that owns the button.

```
begin
  external class Windowtools;
  MainWindow class LineControl;
    begin
      ref(Button) draw_button, pause_button, clear_button, stop_button;
      procedure ClickInButton(b); ref(Button) b;
        begin
          if b == draw_button  then Drawlines       else
          if b == pause_button then paused := true else
          if b == clear_button then Clear           else
          if b == stop_button  then stopped := true;
        end;
      Boolean paused, stopped;  integer seed;
      procedure Drawlines;
        begin
          paused := false;
          while not (paused or stopped) do
            begin
              DrawLine(width//2, height//2,
                       randint(1, width, seed), randint(1, height, seed));
              handle_pending_events;
            end;
        end;
      SetSize(450, 250);  SetFancy;  Show;
      draw_button  :- MakeButton("Draw lines");
      pause_button :- MakeButton("Pause");
      clear_button :- MakeButton("Clear window");
      stop_button  :- MakeButton("Stop");
      seed := 123873;
      stopped := false;
      while not stopped do handle_event;
    end LineControl;
  new LineControl("Menu-choice and interrupts");
end
```

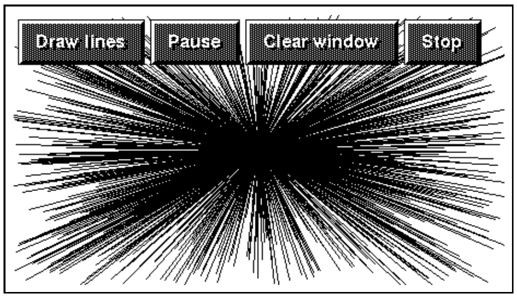

Figure 3: A program that handles interrupts

```
begin
  external class Windowtools;
  MainWindow class MouseWindow;
    begin
      procedure handle_pointer_motion(x, y); integer x, y;
        quit_button . PlaceAt(x + 10, y + 10);
      procedure handle_button_click(button); integer button;
        begin pointer_motion_desensitize; Clear end;
      ref(Button) quit_button;
      SetSize(250, 150);
      quit_button :- MakeButton("Quit");
      SetBackGround("gray25");  Show;
      DrawCenterText("Fix quit-button by clicking");
      pointer_motion_sensitize;
      button_sensitize;
      quit_button . Wait;
    end;
  new MouseWindow("Mouse-demo");
end
```

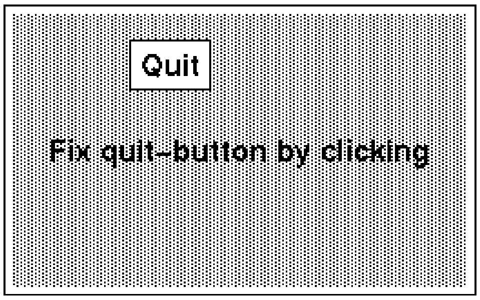

Figure 4: A program that handles events

- The procedure handle_pending_events is programmed as follows:

  ```
  procedure handle_pending_events;
    while not <the event queue is empty> do handle_event;
  ```

  This means that if the queue contains no events, handle_pending_events returns immediately. Otherwise all events in the queue are handled, one by one.

- The effect of sending the message SetFancy to a window is purely cosmetic: The buttons that are created in the window will get a three-dimensional look.

3.4 Handling other kinds of events

The program in figure 4 shows how to use Windowtools to handle events of other kinds than button-clicks. The following new concepts are used in this program:

- Windowtools contains tools which may be used to make a window *sensitive* or *insensitive* for certain kinds of events. If, for instance, the message pointer_motion_sensitize is sent to a window, the window becomes sensitive to movements of the mouse. This means that whenever the mouse is moved with the cursor inside the window, an event occurs and an event-descriptor with data about the event – in particular the coordinates of the cursor and an identification of the window – is placed in the event queue. When handle_event reaches this event, it sends the message handle_pointer_motion – with the coordinates as parameters – to the window identified in the descriptor. The procedure handle_pointer_motion is virtual, which means that the version of handle_pointer_motion declared in the class of the window will be executed, as is the case in the program in figure 4.

When the message pointer_motion_desensitize is sent to a window, the window becomes insensitive to mouse movements. This entails that events no longer will occur when the user moves the mouse inside the window.

- The effect of sending button_sensitize to a window, is to make it sensitive for mouse-button events. This means that an event occurs whenever the user presses or releases a mouse-button with the cursor inside the window.

  If the user presses and then – without moving the mouse – releases a mouse button with the cursor inside a button-sensitive window, handle_event sends the message handle_button_click(<*button_number)*> to the window.

- The effect of sending the message PlaceAt(x, y) to a window, is that the window is placed with its upper left hand corner in point (x, y) in its owner.

- The effect of sending SetBackGround("gray25") to a window, is that it will appear with a light gray background.

3.5 An interface to a very simple database-system

The program in figures 5 and 6 may be used to read and write data about persons from and to a file, and let the user change some of the data or introduce data about new persons. The program uses the following new parts of Windowtools:

- The classes TextItemWindow and IntItemWindow define so-called *item-windows*, which may be used for the display and input of single data-items. The effect of sending the message puttext(<*text*>) to a TextItemWindow is that the specified text is displayed in the window; putint(<*int*>) may similarly be sent to an IntItemWindow. If AllowInput is sent to an item-window, the user is allowed to type data in the window. The function gettext may be used to read what the user has typed in a TextItemWindow, getint to read what has been typed in an IntItemWindow.

- The message InputInSubwindow is sent to a window whenever the user types something in one of its subwindows.

- Hide makes a window invisible, SetMaxchars specifies the width of a window, SetLabel places a label in front of a window, SetSizetoSubwindowSize makes a window just large enough to fit around its subwindows. The procedures PlaceBelow, PlaceLeftBelow and PlaceDownright may be used to position a window relative to one of its siblings.

- Settools (which is not described in this paper, code and a short description may be found in [5]) contains tools for handling sets of various kinds. One of the classes in Settools is Sequence, which contains procedures that may be used to handle sequences of objects that are instances of a subclass of the class element

```
begin
  external class Windowtools, Settools;
  ref(Sequence) persons;
  element class Person;    hidden protected Id, pt;
    begin
      text     Id;  text     procedure getIdent;  getIdent  :- Id;
      integer pt;   integer procedure getpoints; getpoints := pt;
      procedure SetData(i, p); text i; integer p; begin Id :- id; pt := p end;
    end;
  SubWindow class PersonWindow;
    begin
      ref(TextItemWindow) ident_wnd;   ref(IntItemWindow) points_wnd;
      ref(Person) person_in_window;     ref(Button) accept_button, cancel_button;
      procedure ShowButtons; begin accept_button . Show; cancel_button . Show end;
      procedure HideButtons;  begin accept_button . Hide;  cancel_button . Hide  end;
      procedure InputInSubwindow(sw); ref(SubWindow) sw;   ShowButtons;
      procedure ClickInButton(b); ref(Button) b;
        if b == accept_button then AcceptData else
        if b == cancel_button  then ShowPerson(person_in_window);
      procedure ShowPerson(p); ref(Person) p;
        begin
          ident_wnd . puttext(p . getIdent); points_wnd . putint(p . getPoints);
          person_in_window :- p;   HideButtons;
        end;
      procedure AcceptData;
        begin
          person_in_window . SetData(ident_wnd . gettext, points_wnd . getint);
          HideButtons;
        end;
      ident_wnd      :- MakeTextItemWindow . AllowInput
                          . SetMaxchars(30) . SetLabel("Ident:");
      points_wnd     :- MakeIntItemWindow . AllowInput . PlaceBelow(ident_wnd)
                          . SetMaxchars(6) . SetLabel("Points:");
      accept_button :- MakeButton("OK") . PlaceDownRight;
      cancel_button :- MakeButton("Cancel") . PlaceLeftof(accept_button);
      SetSizetoSubwindowSize;   SetBackground("gray25");
      PlaceAt(10, 10);   HideButtons;   Show;
    end PersonWindow;
```

Figure 5: First part of a very simple database-program

```
MainWindow class ControlWindow;
  begin
    ref(PersonWindow) person_wnd;
    ref(Button) first_button, next_button, new_button, from_button, to_button;
    procedure ClickInButton(b); ref(Button) b;
      if b == first_button  then ShowPerson(persons . first_element) else
      if b == next_button  then ShowPerson(persons . next_element) else
      if b == new_button  then NewPerson else
      if b == from_button then FromFile else
      if b == to_button    then ToFile;
    procedure NewPerson;
      begin ref(Person) p;
        p :- new Person;  persons . append(p);
        person_wnd . ShowPerson(p);
      end;
    procedure ShowPerson(p); ref(Person) p;
      if p =/= none then person_wnd . ShowPerson(p);
    procedure FromFile; <Read data from file>;
    procedure ToFile;    <Write data to file>;
    person_wnd :- new PersonWindow(this ControlWindow);
    from_button:- MakeButton("From file") . PlaceLeftBelow(person_wnd);
    first_button :- MakeButton("First");
    next_button :- MakeButton("Next");
    new_button :- MakeButton("New");
    to_button   :- MakeButton("To file");
    SetSizetoSubwindowSize;
    SetBackground("gray50");
    Show;
    persons :- new Sequence;
    MakeButton("Quit") . PlaceDownright . Wait;
  end ControlWindow;
 new ControlWindow("Person-data");
end
```

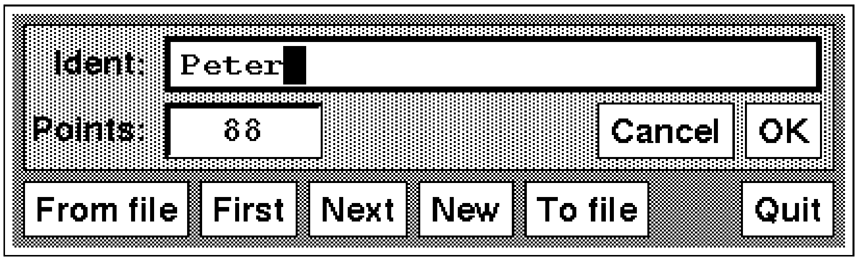

Figure 6: Second part of a very simple database-program

(also one of the Settools-classes). The procedure append may be used to append elements to a sequence; the **ref**(element)-functions first_element and next_element to traverse a sequence.

4 Availability of Windowtools

Windowtools is presently implemented only for use on computers running under Unix and using X, but other versions may possibly be implemented in the future.

Windowtools is copyrighted by the author, but may be used freely for non-commercial purposes at educational institutions.

The *Windowtools*-code is available via anonymous ftp on the Internet from ifi.uio.no in directory /pub/Windowtools.tar.Z. A reference guide to the system [4] is available in the same directory,

References

[1] IBM publication SC26-4582-0. *Common User Access. Advanced Interfac Design Guide*. IBM, 1989.

[2] A. Kay. In discussion at the *History of Programming Languages* conference, Boston 1993.

[3] B. Kirkerud. *Object Oriented Programming with Simula*. Addison-Wesley 1989.

[4] B. Kirkerud. *Windowtools*, 1993. Available via anonymous ftp on ifi.uio.no in directory /pub/Windowtools.tar.Z.

[5] B. Kirkerud. *Settools*, 1993. Available via anonymous ftp on ifi.uio.no in directory /pub/Simulatools.tar.Z.

[6] The X Window System. Volumes 0 – 7. O'Reilly & Associates, Inc., 1990.

[7] D. Mandelkern. Graphical User Interfaces: The Next Generation. Introduction. *Communications of the ACM* (April 1993).