Hush – a C++ API for Tcl/Tk
Version 2.0

Anton Eliëns

Vrije Universiteit, Department of Mathematics and Computer Science

De Boelelaan 1081, 1081 HV Amsterdam The Netherlands

email: eliens@cs.vu.nl

Abstract This article describes the C++ programmer interface to *hush*, the *hyper utility shell* based on Tcl/Tk. Tcl is a scripting language that may be embedded in C or C++. Tk is a window and graphics toolkit based on X11, with an associated interpreter called *wish*. The *hush* library contains classes that provide convenient yet flexible access to the functionality offered by the Tcl/Tk toolkit and its extensions. The library is intended to support the needs of both novice and experienced (window) programmers. It offers widget and graphics classes with an easy to use interface, but allows more experienced programmers also to employ the Tcl scripting language to define the behavior and functionality of widget and structured graphics objects. The design of the *hush* library has been inspired by the InterViews library. However, both the use of event callbacks and the functional interface of widget and graphics classes is significantly simpler. An important advantage of basing *hush* on the Tk toolkit is that existing Tk applications written for the Tk interpreter *wish* can easily be (re)used in a C++ context, virtually without any costs. On the other hand, programs employing *hush* may again be used as an enhanced version of the *wish* interpreter, allowing the functionality defined in the program to be used in a (*hush*) script.

1 Introduction

In comparison with ordinary programming (in C++), programming in a window environment (in C++) introduces a number of additional difficulties. First of all, the programmer must become acquainted with the various widgets constituting the (graphical) user interface, such as buttons, menus, messages, canvasses, etcetera. And secondly, perhaps the most difficult aspect of window programming, the programmer must deal with a rather different control structure, involving actions and callbacks in response to events generated by the user or the window environment, such as mouse movements and mouse button manipulations.

A number of toolkits for the X11 environment with an interface to C++ do exist already. Well-known for example is the InterViews library, which offers powerful features for defining the layout of graphical user interfaces. See

This article describes hush-2.0 (and it's 2.x successors). It is a slightly modified version of the version that has been published in The X Resource, Issue 14, April 1995, O'Reilly & Ass., pp 111-155. Part of the material presented here has been adapted from Principles of Object-Oriented Software Development by Anton Eliëns, (c) 1995 Addison-Wesley Publishing Co. Inc.

[LVC89]. However, despite the elegance of its design, InterViews is slightly cumbersome to use and lacks a number of the features and widgets needed for rapidly implementing a graphical user interface.

Commercial packages for GUI (Graphical User Interface) programming in C++ are available. The disadvantage of these packages, apart from their price, is primarily that they do not offer the flexibility needed in a research environment.

A rather different approach to GUI programming has been advocated in [Ousterhout91], which describes the Tcl/Tk toolkit. Tcl is a cshell-like (interpreted) script language that may be embedded in C or C++. Tk is a window and graphics toolkit based on X11, partly implemented in Tcl and partly in C. Tk offers numerous widgets, including a powerful canvas and text widget. Moreover, the Tcl scripting language allows the user to rapidly prototype rather complex graphical user interfaces by writing Tcl scripts. These scripts may be executed by using *wish*, the windowing shell interpreter that comes with Tk. Despite being based on Tcl, the performance of Tk (and *wish*) is comparable with (and in some respects even better than) C or C++ based toolkits.

The Tcl/Tk toolkit has become very popular in a rather short period of time. The popularity of Tcl/Tk is partly due to the extensibility of Tcl. New functionality, implemented in C, may easily be added by creating a new version of the *wish* interpreter, incorporating the additional commands. Numerous extensions to Tcl/Tk and corresponding interpreters have been made available, including extensions offering facilities for distributed programming (*dp*), extensions offering object oriented features ([*incr tcl*])[1], and extensions offering additional widgets such as a barchart and hypertext widget (*blt*).[2]

The possibility of employing interpreted code and the availability of numerous widgets makes the Tcl/Tk toolkit (and its extensions) an ideal vehicle for implementing user interfaces.

However, Tcl/Tk has its drawbacks as well. One problem, obviously, is to manage the large number of extensions. Ideally, there is one *wish*-like shell unifying the various features. Even better, one should have the opportunity to create such a shell in a simple manner.

A second problem is that, when an application grows, script code will not always allow for an optimal solution. Generally, script code is not robust and may be hard to maintain. In particular, when an application contains many components not related to the user interface, efficiently compiled code may be more appropriate.

For the latter problem, the obvious solution is to employ the C API (Application Programmer Interface) offered by Tcl and to create a new interpreter including the functionality needed. In a similar way, the first problem is rather easily solved by linking the appropriate libraries into an extended interpreter.

Nevertheless, this is easier said than done. First of all, the C API offered for Tcl/Tk is rather demanding for the novice programmer and does not support a style of programming that is recommendable from a software engineering perspective. Secondly, although not very difficult, creating a new interpreter with additional C/C++ code is somewhat cumbersome.

The *hush* library has been developed to address the two problems mentioned. *Hush* stands for *hyper utility shell*. The standard interpreter associated with the *hush* library is a shell, called *hush*, including a number of the available extensions of Tcl/Tk and widgets developed by ourselves (such as a *filechooser* and an MPEG video widget). The *hush* library offers a C++ interface to the Tcl/Tk toolkit and its extensions. It allows the programmer to employ the functionality

[1] [*incr tcl*] may be read as a paraphrase of tcl++ in Tcl syntax.

[2] These extensions may be obtained from `harbor.ecn.purdue.edu`.

of Tcl/Tk in a C++ program. Moreover, a program created with *hush* is itself an interpreter extending the *hush* interpreter (and *wish*).

The *hush* library is explicitly intended to support the needs of both novice and experienced window programmers. Its C++ class interface should suffice for most applications, yet it allows for employing Tcl script code when more is demanded.

The contribution of *hush* with respect to the Tcl/Tk toolkit is essentially that it provides a type-secure solution for connecting Tcl and C++ code. As an additional advantage, the *hush* library allows the programmer to employ inheritance for the development of possibly compound widgets. In particular, it provides the means to define composite widgets that behave as the standard Tk widgets.

The class structure of the *hush* library is reminiscent to the class structure of the InterViews library. In comparison with the InterViews library, the widget class interfaces and event callbacks are significantly easier to use. Also, the *hush* library provides many more ready-to-use graphical interface widgets. However, *hush* does not offer resolution-independent graphics and provides no pre-defined classes for complex interactions.

Summarizing, *hush* supports a multi-paradigm approach to window programming, allowing to combine the robustness of compiled C++ code with the flexibility of interpreted Tcl code. As such, it offers the best of both worlds. Or the worst, for that matter.

The material presented here requires at least some knowledge of C++. See for example [Stroustrup91]. Some familiarity with Tcl/Tk is also helpful. See [Ousterhout94].

Availability The *hush* library has been in use for student programming assignments at the Vrije Universiteit for two years. It may be obtained by anonymous ftp from `ftp.cs.vu.nl`, directory `eliens/hush`. You may also retrieve it via `http://www.cs.vu.nl/~eliens/hush/`.

Structure In section 2, some background information concerning Tcl/Tk is given. Section 3 sketches the structure of a typical *hush* program and gives an overview of the *hush* class library, including the *kit* and *session* class. Section 4 describes how handler objects may be defined as event callbacks. Next, section 5 presents a drawing tool application. The application illustrates the use of the various widget classes and demonstrates how to construct compound widgets. In addition, it shows how a widget developed in C++ may be made available as a widget command to be used in scripts. And finally, in section 6, we will look at the facilities offered for structured graphics and hypertext.

2 Background – Tcl/Tk

The language Tcl has first been presented in [Ousterhout90]. Tcl was announced as a flexible cshell-like language, intended to be used for developing an X11-based toolkit. A year later, the Tk toolkit (based on Tcl) was presented in [Ousterhout91]. From the start Tcl/Tk has received a lot of attention, since it provides a flexible and convenient way to develop rather powerful window applications.

The Tcl language offers variables, assignment and a procedure construct. Also it provides a number of control constructs, facilities for manipulating strings and built-in primitives giving access to the underlying operating system. The basic Tcl language may easily be extended by associating a function written in C with a command name. Arguments given to the command are passed as strings to the function defining the command.

The Tk toolkit is an extension of Tcl with commands to create and configure widgets for displaying text and graphics, and providing facilities for window management. The Tk toolkit, and the *wish* interpreter based on Tk, provides a convenient way to program X-window based applications.

Figure 1: Hello world

Wish The *wish* program is an interpreter for executing Tcl/Tk scripts. As an example of a *wish* script, look at the *hello world* program below:

```
button .b -text "hello world"  -command { puts "hello world" }
pack .b
```

This program results in the widget shown in figure 1. It defines a button that displays *hello world*, and prints *hello world* to standard output when it is activated by pressing the left mouse button. The language used to write this script is simply Tcl with the commands defined by Tk, as for example the *button* command (needed to create a button) and the *pack* command (that is used to map the button to the screen).

The *wish* program actually provides an example of a simple application based on Tcl/Tk. It may easily be extended to include for example 3D-graphics by linking the appropriate C libraries and defining the functions making this functionality available as (new) Tcl commands.

The Tcl C API To define Tcl commands in C style, the programmer has to define a command function, with a profile similar to the function *aCommand* shown below, and declare the function to be a command in Tcl by invoking the *Tcl_CreateCommand* function:

```
// Define a command function in C style

int aCommand( ClientData data, Tcl_Interp* interp,
                               int args, char* argv[]) {

some_type* x = (some_type*) data;        // conversion by cast

// some processing

}

// Declare the function aCommand as a Tcl command
// for example in the  main  function

some_type* user = new some_type(); // to create the client data
Tcl_CreateCommand( interp, "aco", aCommand, (ClientData) user );
```

Creating a command is done with reference to an interpreter, which accounts for the first argument of *Tcl_CreateCommand*. The name of the command (*aco* in this case), as may be used in a Tcl script is given as a second argument, and the C style function defining the command as a third argument. Finally, the address of a structure containing client data (*user* in this case) is passed as the fourth parameter.

When the function *aCommand* is invoked as the result of executing the Tcl command *aco*, the client data stored at declaration time is passed as the first argument to the function. Since the type *ClientData* is actually defined to be *void**, the function must first cast the client data argument to an appropriate type as indicated above. Clearly, casting is error-prone.

Another problem with command functions as used in the Tcl C API is that permanent data are possible only in the form of client data, global variables or static local variables. Both client data and global variables are unsafe by being too visible and static local data are simply inelegant.

The *hush* library has been developed to offer a type-secure solution to the problem of connecting C++ code with Tcl, and to allow for a safe way of maintaining a (dynamically changing) state.

In *hush* the preferred way is to employ *handler* objects. The obvious solution of associating class member functions with Tcl commands does not work since pointers to member functions are different from pointers to ordinary C style functions.

eliens@cs.vu.nl

3 Program structure

The *hush* library is intended to provide a convenient way to program window-based applications in C++. Basically, there are two considerations that may lead you to employ the *hush* library. When you are familiar with Tcl/Tk and you need to combine Tcl scripts with C++ code, you may use *handler* classes to do so in a relatively type-secure way. On the other hand, when you want to program graphical user interfaces in C++, you may employ the *hush* widget classes. In the latter case, you may choose to remain ignorant of the underlying Tcl/Tk implementation or exploit the Tcl script facility to the extent you wish.

As an illustration of the structure of a program using *hush*, we will look at a simple program written in C++ that uses a graphical interface defined by a Tcl/Tk script.

After discussing the example, we will look at a brief overview of the classes that constitute the *hush* library. A more detailed description will be given of the *kit* class, that encapsulates the embedded Tcl interpreter, and the *session* class, that shields off the details of the window environment.

3.1 Employing Tcl/Tk from within C++

Imagine that you have written some numerical function, for example a function employing the Newton method for computing the square root. Such a function may be defined as in the function *newton*:

```
double newton( double arg ) {        // computes square root
double r=arg, x=1, eps=0.0001;
while( fabs(r - x) > eps ) {
   r = x;
   x = r - (r * r - arg) / (2 * r);
```

```
    }
  return r;
  }
```

When you have written such a function, you may wish to have a graphical interface to allow you to experiment with possible inputs in a flexible way. For example, you may wish to have a slider for setting the input value and a message widget displaying the outcome of the function. Such an interface may look like the one in figure 2.

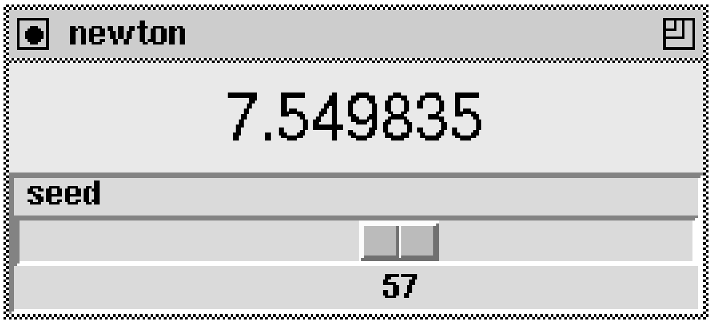

Figure 2: A graphical interface for *newton*

Admittedly, the *newton* function given above is simple enough to be implemented directly in Tcl. Nevertheless, since C++ is to be considered superior for implementing numerical functions, we decide to implement the Newton function in C++ and the graphical interface in Tcl. The problem we need to solve then is how to connect the graphical interface with the C++ code.

The Tcl script Let us start by defining the interface, where we will use a dummy function to generate the output. A Tcl script defining our interface is given below:

```
#!/usr/prac/se/bin/hush -f

proc generate {} {
         .m configure -text [.s get]
}

scale .s -label "seed" -orient horizontal -relief sunken
message .m -width 256 -aspect 200

pack .m .s -fill x

bind .s <Any-ButtonRelease> { generate }
```

The script defines a slider, as a (horizontal) *scale* widget, and a *message* widget, that is used to display the output. The built-in Tcl/Tk *bind* function is used to associate the movement of the slider with the invocation of the Tcl function *generate*. Note that the function *generate* is a dummy function, which merely echos the value of the *scale* widget to the *message* widget.

Now we have developed a graphical interface, which may be tested by using the *hush* shell or *wish*. Next, we need to develop the C++ program embodying the numerical function and connect it to the interface written in Tcl.

The C++ code The structure of this program is best explained in a number steps. Each of these steps corresponds with a code fragment. Together, these fragments form the C++ program shown below. We will first look at the code. Afterwards it will be explained why the individual fragments are needed.

```
// Initial declarations

#include "hush.h"

double newton(double arg);    // declare the function

char* ftoa( double f);        // to convert float to char*

// The generator (handler) class gives access to the widgets

class generator : public handler {
public:

  generator() {                        // access to Tcl widgets
          s = (scale*) new widget(".s");
          m = (message*) new widget(".m");
  }

  ~generator() {
          s->destroy(); m->destroy(); // to destroy widgets
          delete s; delete m;       // to reclaim resources
  }

  int operator()() {             // the generator action
        float f = s->get();
        m->text( ftoa( newton(f) ) );     // display value
        return OK;
}

private:
  scale* s;
  message* m;
};

// The application class takes care of installing the interface

class application : public session {
public:
  application(int argc, char* argv[])
                         : session(argc,argv,"newton") {}

  void main( ) {                    // tk is an instance variable

      tk->source("interface.tcl");  // read interface script
      handler* g = new generator();

      tk->bind("generate",g);     // bind Tcl command
  }
};
```

```
// Finally, the function main is defined

void main (int  argc, char  **argv) {
    session* s = new application(argc,argv);
    s->run();
}
```

The functional part is represented by the function *newton*. We need to declare its type to satisfy the compiler. Further we need to include the `hush.h` header file and declare an auxiliary function *ftoa* that is used to convert floating point values to a string.

The next step involves the definition of the interfacing between the Tcl code and the C++ program. The class *generator* defines a so-called handler object that will be associated with the function *generate* employed in the script, overriding the dummy Tcl function *generate* as defined in the script. In order to access the scale and message widget defined for the interface, C++ pointers to these widgets are stored in instance variables of the object. These pointers are initialized when creating a *generator* object. The widgets are destroyed when deleting the object. Note that the widgets must first be destroyed before deleting the corresponding C++ objects.

All you need to know at this stage is that when the function *generate* is called in response to moving the slider, or more precisely releasing the mouse button, then the *operator()* function of the C++ *generator* object is called. In other words, the *operator()* function is (by convention) the function that is executed when a Tcl command that is bound to a *handler* object is called. The *generator::operator()* function results in displaying the outcome of the *newton* function, applied to the value of the slider, in the message widget.

Then we define an application class, which is needed for the program to initialize the X-windows main event loop. An application class must be a subclass of the *session* class. To initialize the program, the application class redefines the (virtual) function *main* inherited from *session*. The function *application::main* takes care of initializing the interface, creates an instance of the *generator* class, and binds the Tcl command *generate* to the *generator* object.

Finally, the function *main* is defined. A function *main* is required for each C or C++ program. It consists merely of creating an instance of the *application* class and the invocation of *run*, which starts the actual program.

Comments The example C++ program illustrates a number of features, some of which are typical for *hush* and some of which are due to programming in a window environment.

In an ordinary C++ program the function *main* is used to start the computation. Control is effected by creating objects and calling the appropriate functions. When programming a window-based application, at a certain moment control is delegated to the window environment. Consequently, there needs to be some kind of main loop which waits for incoming events, in response to which the control may be delegated to an appropriate component of the program.

To hide the details of activating the main loop and the dispatching of events, the *hush* library provides a class *session* that allows you to define an application class to initialize your program.

In order to respond to events, the *hush* library provides a *handler* class, that allows you to associate a C++ object with a Tcl function. Each time the corresponding Tcl function is invoked, the *operator()* function of the object is called. The actual object is an instance of a derived class, redefining the virtual *operator()* function of the *handler* class.

Handler classes are typical for *hush*. Another feature typical for *hush* is the use of a *kit* object, that may be accessed by using the *tk* instance variable of the *handler* object. The *kit* object provides access to the Tcl interpreter embedded in the C++ program. In the example it is used to initialize the graphical interface by reading a script file and to define the association between the Tcl function *generate* and the C++ instance of *generator*.

The widgets defined in the Tcl script are accessed in the C++ program by means of a *scale* and *message* pointer. The *hush* library provides for each Tk widget a class of the same name. Note that not the widgets themselves are created in the constructor of the *generator* class, but only abstract widget objects that are casted to the appropriate widget types. Casts are needed to access these objects as respectively a *scale* and *message* widget. Widgets can be created, however, directly in C++ as well, by employing the appropriate widget class constructors. See section 5.

As a final comment, the example illustrates a classical stratagem of software engineering, namely the *separation of concerns*. On the one hand we have a script defining the interface that may be independently tested, and on the other hand we have C++ code embodying the real functionality of our program.

3.2 An overview of the hush class library

The example given in the previous section showed what kind of components are typically used when developing a program with the *hush* library. However, instead of employing a Tcl script, the window interface may also be developed entirely by employing *hush* C++ widgets. In this section, a brief overview will be given of the classes offered by the *hush* library. Further, it will be shown how to construct the *hush* interpreter referred to in the introduction. In addition, we will take a closer look at the classes *kit* and *session*, which are needed to communicate with the embedded Tcl interpreter and to initialize the main event loop, respectively.

The library The *hush* C++ library consists of three kinds of classes, namely the widget classes which mimic the functionality of Tk, the handler classes, which are involved in the handling of events and the binding of C++ code to Tcl commands, and the classes *kit* and *session*, which encapsulate the embedded interpreter and the window management system,

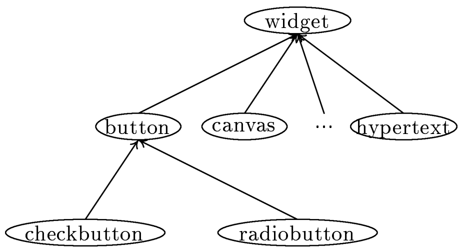

Figure 3: Hush widget classes

In the widget class hierarchy depicted in figure 3, the class *widget* represents an abstract widget, defining the commands that are valid for each of the descendant concrete widget classes. The *widget* class, however, is not an abstract class in C++ terms. As shown in the example in the previous section, the *widget* class allows for creating pointers to widgets defined in Tcl. In contrast, employing the constructor of one of the concrete widget classes results in actually creating a widget. A more detailed example showing the functionality offered by the widget classes will be given in section 5. A description of the individual widget classes is included in the appendix.

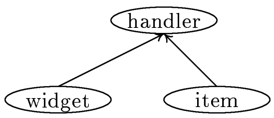

Figure 4: Hush handler classes

The *handler* class may also be considered an abstract class, in the sense that it is intended to be used as the ancestor of a user-defined handler class. Recall that in the example we defined the *generator* class as a descendant of *handler*. The *handler* class has two pre-defined descendant classes, namely the *widget* class and the class *item*. This implies, indeed, that both the *widget* and the *item* class (that is treated in section 6.1) may be used as ancestor handler classes as well. The reason for this is that any descendant of a *widget* or *item* class may declare itself to be its own handler and define the actions that are invoked in response to particular events. This will be illustrated and discussed in sections 4 and 5.

The hush interpreter In the introduction, *hush* was announced as both a C++ library and as an interpreter extending the *wish* interpreter. The program shown below substantiates this claim, albeit in a perhaps disappointingly simple way.

```
#include "hush.h"

// Include definitions of external package(s)

#include "extern/ht.h"

// Define the application class

class application : public session {
public:

  application(int argc, char* argv[]) : session(argc,argv) {
      hyper = 0;
      if ((argc==3) && !strcmp(argv[1],"-x")) {  // check for -x
                  hyper = 1;
                  strcpy(hyperfile,argv[2]);
                  }
  }
```

```
    void main() {                              // tk represents the kit
       init_ht(tk);                            // initialize package(s)

       if (hyper) {                           // initialize hypertext
            hypertext* h = new hypertext(".help");
            h->file(hyperfile);
            h->geometry(330,250);
            h->pack();
            tk->pack(".quit");           // predefined button to quit
            }
    }
  private:
    char hyperfile[BUFSIZ];
    int hyper;
  };

  // Define the main function

  int main (int argc, char* argv[]) {
    session* s = new application(argc,argv);
    s->run();                                 // start X event loop
  }
```

The structure of the program is similar to the C++ example of section 3.1. In addition to including the `hush.h` header file, however, we must include the declarations needed for employing the external hypertext (*ht*) and distributed processing (*dp*) packages. Next, we need to define an application class, derived from *session*, specifying how the *hush* interpreter deals with command-line arguments and what initialization must take place before starting the main event loop.

At this stage it suffices to know that the *hush* library provides a hypertext widget and that the `-x` option treats the next argument as the name of a hypertext file. In section 6.3, an example will be given that involves the hypertext widget.

Further, a predefined button `.quit` is packed to the root widget.

The *hush* interpreter defined by the program extends the *wish* interpreter by loading the *distributed processing* extension and by allowing for the display of a hypertext file. The interpreter accepts any command-line argument accepted by the *wish* interpreter, in addition to the `-x` hypertext option. The Tcl interface script given in section 3.1, for example, may be executed using the *hush* interpreter.

3.3 The *kit* class

Hush is meant to provide a simple C++ interface to Tcl/Tk. Nevertheless, as with many a toolkit, some kind of API shock seems to be unavoidable. This is especially true for the *widget* class (treated in section 5.1) and the class *kit* defining the C++ interface with the embedded Tcl interpreter. The functionality of *kit* can only be completely understood after reading this article. However, since an instance of *kit* is used in almost any other object (class), it is presented here first. The reader will undoubtly gradually learn the functionality of *kit* by studying the examples. The class interface of *kit* is given below:[3]

```
interface kit {

  int eval(char* cmd);        // to evaluate script commands
  char* result();             // to fetch the result of eval
  void result(char* s);       // to set the result of eval
  char* evaluate(char* cmd)   // combines eval and result
  int source(char* f);        // to load a script from file

  void bind(char* name, handler* h); // to bind Tcl command

  widget* root();             // returns toplevel (root) widget
  widget* pack(widget* w, char* options = "-side top -fill x");
  widget* pack(char* wp, char* options = "-side top -fill x";

  char* selection(char* options="");          // X environment

  void after(int msecs, char* cmd);
  void after(int n, handler* h);

  void update(char* options="");

  char* send(char* it, char* cmd);

  void trace(int level = 1);
  void notrace();
  void quit()                         // to terminate the session
};
```

To understand why a *kit* class is needed, recall that each *hush* program contains an embedded Tcl interpreter. The *kit* class encapsulates this interpreter and provides a collection of member functions to interact with the embedded interpreter.

The first group of functions (*eval*, *result*, *evaluate* and *source*) may be used to execute commands in Tcl scripting language directly. A Tcl command is simply a string conforming to certain syntactic requirements. The function *eval* evaluates a Tcl command. The function *result()* may be used to fetch the result of the last Tcl command. In contrast, the function *result(char*)* may be used to set the result of a Tcl command, when this command is defined in C++ (as may be done with *kit::bind*). The function *evaluate* provides a shorthand for combining *eval* and *result()*. The function *source* may be used to read in a file containing a Tcl script.

Also, we have the *kit::bind* function that may be used to associate a Tcl command with a handler object.

The next group of functions is related to widgets. The function *root* gives access to the toplevel root widget associated with that particular instance of the *kit*. The function *pack* may be used to append widgets to the root widget, in order to map them to the screen. Widgets may be identified either by a pointer to a *widget* object or by their *path name*, which is a string. See section 5.1.

[3]Each function listed in the class interface is *public* unless it is explicitly indicated as *protected*. The interface descriptions start with the pseudo-keyword *interface*. This is merely done to avoid the explicit indication of *public* for both the ancestor and the member functions of the class.

Next, we have a group of functions related to the X environment. The function *selection* delivers the current X selection. The function *after* may be used to set a timer callback for a handler. Setting a timer callback means that the handler object will be invoked after the number of milliseconds given as the first argument to *after*.

The function *update* may be used to enforce that all pending events are processed. For example, when moving items on a canvas, an update may be needed for making the changes visible. Also, we have a function *send* that may be used to communicate with other Tcl/Tk applications. The first argument of *send* must be the name of an application, which may be set when creating a session.

Further, we have the functions *trace* and *notrace*, which may be used to turn on, respectively off, tracing. The level indicates in what detail information will be given. Trace level zero is equivalent to notrace(). Finally, the function *quit* may be used to terminate the main event loop.

3.4 The *session* class

Each program written with *hush* may contain only one embedded *hush* interpreter. To initialize the *kit* object wrapping the interpreter and to start the main event loop, an instance of the class *session* must be created.

The preferred way of doing this is by defining a descendant class of the *session* class, redefining the virtual function *session::main* to specify what needs to be done before starting the main loop. In addition, the constructor of the newly defined class may be used to check command line arguments and to initialize application specific data, as illustrated in the code for the interpreter in section 3.2.

```
interface session  {
  session(int argc, char** argv, char* name = 0);

  virtual void prelude();
  virtual void main();
  int run( );
protected:
  kit* tk;
};
```

When creating a *session* object, the name of the application may be given as the last parameter. By this name the application is known to other Tk applications, that may communicate with each other by means of the *send* command.

Apart from the function *main*, also a function *prelude* may be defined. When the program is used as an interpreter (by giving `-f` *file* as command line arguments) only the *prelude* function will be executed, otherwise *prelude* will be executed before *main*. In interpreter mode, the function *main* will also be executed when the script contains the command *go-back*. Finally, the function *run* must be called to actually initialize the program and start the main loop.

4 Binding actions to events

In the example in section 3.1 we have seen that *handler* objects may be bound to Tcl commands. Handler objects may also be bound to events.

Events are generated by the (X) window environment in response to actions of the user. These actions include pressing a mouse button, releasing a mouse button, moving the mouse, etcetera. Instead of explicitly dealing with all incoming events, applications delegate control to the environment by associating a callback function with each event that is relevant for a particular widget. This mechanism frees the programmer from the responsibility to decide to which widget the event belongs and what action to take.

Nevertheless, from the perspective of program design, the proper organization of the callback functions is not a trivial matter. Common practice is to write only a limited number of callback functions and perform explicit dispatching according to the type of event.

An object oriented approach may be advantageous as a means to organize a collection of callback functions as member functions of a single class [Eliens95]. One way of doing this is to define an abstract event handler class which provides a virtual member function for each of the most commonly occurring events. In effect, such a handler class hides the dispatching according to the type of the event. A concrete handler class may then be defined simply by overriding the member functions corresponding to the events of interest.

In the following, we will look at how we may define a simple drawing editor by declaring a handler defining the response to pressing, moving and releasing a mouse button. After that we will look more closely at the notion of events and the definition of handlers and actions.

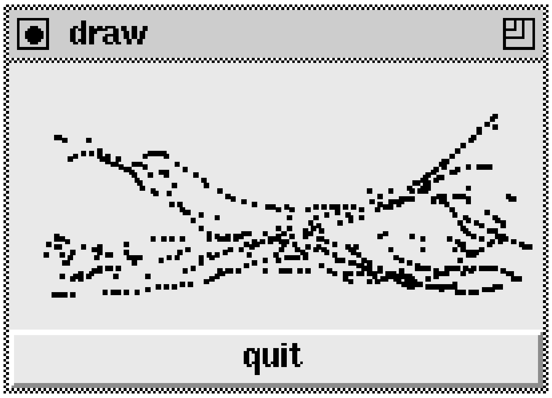

Figure 5: A simple drawing editor

A simple drawing editor Before looking at the program, think of what you would like a drawing editor to offer you. And, if you have any experience in programming graphics applications, how would you approach the implementation of a drawing editor?

A drawing editor is a typical example of an interactive program. As a first approximation, we will define a drawing editor that allows the user to paint a series of black dots by pressing and moving the mouse button, as pictured in figure 5. The program realizing our first attempt is given below:

```
#include "hush.h"

// The drawing_canvas responds to press, motion and release events
```

```
class drawing_canvas : public canvas {
public:

  drawing_canvas( char* path ) : canvas(path) {
          geometry(200,100);
          bind(this);                 // become sensitive
          dragging = 0;
  }

  void press( event& ) { dragging = 1; }

  void motion( event& e) {
          if (dragging) circle(e.x(),e.y(),1,"-fill black");
  }

  void release( event&  ) { dragging = 0; }

protected:
  int dragging;
};

// To initialize the application

class application : public session {
public:
  application( int argc, char* argv[] )
                             : session(argc,argv,"draw") {}
  void main() {

      canvas* c = new drawing_canvas(".draw");

      c->pack();
      tk->pack(".quit");
  }
};

int main (int  argc, char* argv[]) {
    session* s = new application(argc,argv);
    return s->run();
}
```

Again, the program may be broken up in a number components. First we must include the hush.h header file. Next we define the class *drawing_canvas*. The class *drawing_canvas* inherits from the *canvas* widget class and consequently allows for drawing figures such as a *circle*. See section 6.2 and the appendix for further details on the *canvas* class.

Before looking at the constructor of the *drawing_canvas*, note that the member functions *press*, *motion* and *release* expect a reference to an *event*. These are precisely the member functions corresponding to the event types for which the canvas is sensitive. The meaning of these member functions becomes clear when looking at the role of the instance variable *dragging*. When *dragging* is non-zero and a *motion* event occurs, a black dot is painted on the canvas. Drawing starts when pressing a mouse button and ends when releasing the button.

Turning back to the constructor, we see that it expects a *path* string, which is passed to the *canvas* ancestor class to create an actual canvas widget. Further, the body of the constructor sets the size of the widget to 200 by 100 and initializes the variable *dragging* to zero. Finally, the *drawing_canvas* widget is declared to be its own handler. The member function *handler* is defined by the class *widget*. Invoking the *handler* function results in making the widget sensitive to a number of predefined events.

Discussion A note on terminology is in place here. The reader may be confused by the fact that handlers can be bound to Tcl actions as well as to events. The situation may become even more confusing when realizing that the *widget* class itself is a descendant of the *handler* class. Schematically, we have

```
class widget : public handler {
public:
  ...
  void bind(class handler* h) { ... }
  ...
};
```

The *bind* function declares a *handler* object to be responsible for dealing with the events that are of interest to the widget.

In other words, a *drawing_canvas* fulfills the dual role of being a widget and its handler. This must, however, be explicitly indicated by the programmer, which explains the occurrence of the otherwise mysterious expression *bind(this)*. The reason not to identify a widget with a handler is simply that some widgets may need separate handlers.

Before studying the abstract *handler* class in more detail, we will briefly look at the definition of the *event* class.

4.1 Events

Events always belong to a particular widget. To which widget events are actually directed depends on whether the programmer has defined a binding for the event type. When such a binding exists for a widget and the (toolkit) environment decides that the event belongs to that widget, then the callback associated with that event is executed. Information concerning the event may be retrieved by asking the kit for the latest event.

```
interface event  {
  int type();                    // X event type
  char* name();                  // type as string

  int x();
  int y();

  int button(int i = 0);          // ButtonPress
  int buttonup(int i = 0);        // ButtonRelease
  int motion();                   // MotionNotify

  int keyevent();                 // KeyPress or KeyRelease
  int buttonevent(int i = 0);     // ButtonPress or ButtonRelease

  int keycode();
```

```
  void trace();                   // prints event information

  void* rawevent();               // delivers raw X event
};
```

Event objects represent the events generated by the X-window system. Each event has a type. The type of the event can be inspected with *type()* which returns an integer value or *name()* which returns a string representation of the type. For some of the common event types, such as *ButtonPress*, *ButtonRelease*, and *MotionNotify*, member functions are provided to facilitate testing. If an integer argument (1,2 or 3) is given to *button*, *buttonup* or *buttonevent*, the routines check whether the event has occurred for the corresponding button.

The functions x and y deliver the widget coordinates of the event, if appropriate.

Calling *trace* for the event results in printing the type and coordinate information for the event. When setting the *kit::trace* level to 2 this information is automatically printed.

Programmers not satisfied with this interface can check the type and access the underlying XEvent at their own risk.

4.2 Handlers

Handler objects provide a type secure way to deal with local data and global resources needed when responding to an event. An actual handler class must be derived from the (abstract) handler class defined below:

```
interface handler {

  virtual event* dispatch(event* e);
  virtual int operator()();

  virtual void press( event& ) { }
  virtual void release( event& ) { }
  virtual void keypress( event& ) { }
  virtual void keyrelease( event& ) { }
  virtual void motion( event& ) { }
  virtual void enter( event& ) { }
  virtual void leave( event& ) { }
  virtual void other( event& ) { }

protected:
  event* _event;
  kit* tk;
};
```

An instance of an actual handler class may store the information it needs in its instance variables when it is created. A handler object can be activated in response to an event or a Tcl command by calling the *dispatch* function of the handler. The system takes care of this, provided that the user has bound the handler object to a Tcl command or event. The definition of the *dispatch* function is given below:

```
event* handler::dispatch( event* e ) {
        _event = e;
        int res = this->operator()();
```

```
      return (res != OK) ? _event : 0;
}

int handler::operator()() {

        event& e = * _event;                /// fetch event

        if ( e.type() == ButtonPress ) press(e);
        else if ( e.type() == ButtonRelease ) release(e);
        else if ( e.type() == KeyPress ) keypress(e);
        else if ( e.type() == KeyRelease ) keyrelease(e);
        else if ( e.type() == MotionNotify ) motion(e);
        else if ( e.type() == EnterNotify ) enter(e);
        else if ( e.type() == LeaveNotify ) leave(e);
        else other(e);
        return OK;
}
```

The *dispatch* function, in its turn, calls the *operator()* function, after storing the incoming *event* in the corresponding instance variable. The kit variable is initialized by the constructor of the handler.

The *handler::operator()* function selects one of the predefined member functions (*press, motion, release, etcetera*) according to the type of the event.

The original handler class knows only virtual functions. Each of these function, including the *dispatch* and *operator()* function may be redefined.

The two-step indirection, via the *dispatch* and *operator()* function, is introduced to facilitate derivation by inheritance, directly from the *handler* or from classes that are itself derived from the *handler* class, such as the widget classes.

Handler objects are activated only when a binding has been defined, by using *kit::bind* or implicitly by employing *widget::bind*. Such bindings may also be defined by *kit::after* or *item::bind*. Implicit binding results in the creation of an anonymous binding.

Technically, the implementation of event handling by means of handler objects employs a callback function with the handler object as client data. The *dispatch* function of the client object is called when the callback function is invoked in response to an event or a Tcl function.

Bindings encode the binding of C++ handlers to Tcl/Tk commands. They will not be further discussed here.

5 User interface widgets

The Tk toolkit offers numerous built-in widgets. The Tk widgets conform to the look-and-feel of the OSF/Motif standard. The *hush* C++ interface for Tk provides for each Tk widget a class of the same name, which supports the creation of a widget and allows the user to access and modify it. In addition to the standard Tk widgets, the *hush* library includes a number of other widgets, such as a *barchart*, *hypertext*, and *photo* widget (created by other Tk adepts). Also widgets of our own making are offered, such as a *filechooser*, an MPEG video widget, and recently a World Wide Web browser widget.

In this section we will look at an extension of the simple drawing tool presented in section 5.

The example illustrates how to use the *hush* library widgets. The example serves to illustrate in particular how handlers may be attached to widgets, either by declaration or by inheritance, and how to construct compound widgets.

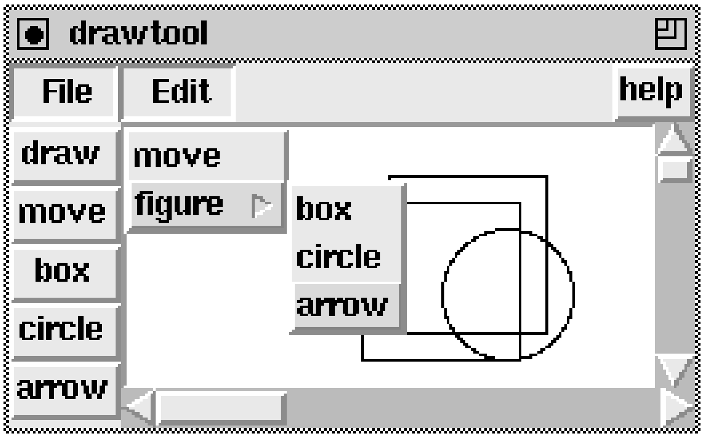

Figure 6: The *drawtool* interface

Our approach may be considered object oriented, in the sense that each component of the user interface is defined by a class derived from a widget class.

It must be pointed out beforehand, that the major difficulty in defining compound or mega widgets is not the construction of the components themselves, but to delegate the configuration and binding instructions to the appropriate components.

In section 5.5 it will be shown how a compound widget defined in C++ may be made to correspond to a widget command that may be used in a Tcl script. Ideally, defining a new widget includes both the definition of a C++ class interface and the definition of the corresponding Tcl command.

Example – drawtool Our drawing tool consists of a *tablet*, which is a canvas with scrollbars to allow for a large size canvas of which only a part is displayed, a *menu_bar*, having a *File* and an *Edit* menu, and a *toolbox*, which is a collection of buttons for selecting among the drawing facilities. See figure 6. In addition, a help facility is offered. The application class for *drawtool* is defined as follows:

```
class application : public session {
public:
  application(int argc, char* argv[])
                            : session(argc,argv,"drawtool") {}

  void main() {
    widget* root = tk->root();
    frame* f = new frame(root,".frame");

    tablet* c = new tablet(f);                // create tablet

    toolbox* b = new toolbox(f,c);
    menubar* m = new menu_bar(root,c,b);

    b->pack("-side left");                      // pack tablet
    c->pack("-side right");
```

```
    tk->pack(m)->pack(f);                 // pack menu and frame
  }
};
```

Before the main event loop is started, the components of the drawing tool are created and packed to the root widget.

In addition to the *tablet*, *menu_bar* and *toolbox*, a *frame* widget is created to pack the *toolbox* and *tablet* together. This is needed to ensure that the geometrical layout of the widget comes out right.

Each of the component widgets is given a pointer to the root widget. In addition, a pointer to the *tablet* is given to the *toolbox* and a pointer to the *toolbox* is given to the *menu_bar*. The reason for this will become clear when discussing the *toolbox* and *menu_bar* in sections 5.2 and 5.3, respectively. In the example, no attention will be paid to memory management.

5.1 Configuring widgets

Widgets are the elements a GUI is made of. They appear as windows on the screen to display text or graphics and may respond to events such as motioning the mouse or pressing a key by calling an action associated with that event.

Most often, the various widgets constituting the user interface are (hierarchically) related to each other, as for instance in the *drawtool* application which contains a canvas to display graphic elements, a button toolbox for selecting the graphic items and a menubar offering various options such as saving the drawing in a file.

Pathnames Widgets in Tk are identified by a *pathname*. The pathname of a widget reflects its possible subordination to another widget. See figure 7.

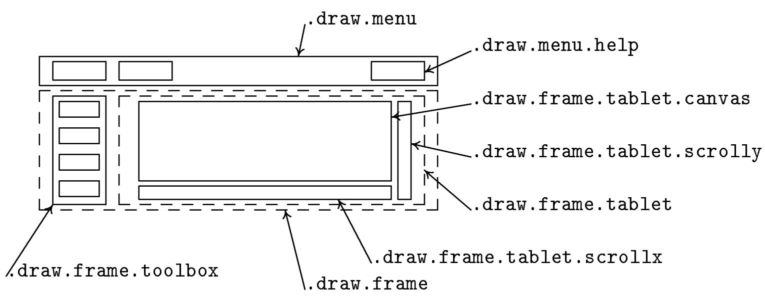

Figure 7: The *drawtool* widget hierarchy

Pathnames consist of strings separated by dots. The first character of a path must be a dot. The first letter of a path must be lower case. The format of a path name may be expressed in BNF form as

```
path ::= . | .string | path.string
```

where *string* and *path* are nonterminals. For example "." is the pathname of the root widget, whereas ".quit" is the pathname of a child of the root widget. A widget that is a child of another widget must have the pathname of its parent as part of its own path name. For example, the widget ".f.m" may have a widget ".f.m.h" as a child widget. Note that the widget hierarchy induced by the pathnames is completely orthogonal to the widget class inheritance hierarchy depicted in figures 3 and 11. Pathnames are treated somewhat more liberally in *hush*. For example, widget pathnames may simply be defined or extended by a string. The missing dot is then automatically inserted.

The *widget* class When creating a widget, a pathname must be given to the widget constructor. Pathnames may be defined relative to a parent widget. The class interface of *widget* is given below:

```
interface widget : handler {

  widget(char* p);
  widget(widget& w, char* p);

  char* type();                // returns type of the widget
  char* path();                // returns path of the widget

  int eval(char* cmd);         // invokes "thepath() cmd"
  char* result();              // returns the result of eval
  char* evaluate(char* cmd);   // combines eval and result()

  virtual void configure(char* cmd);  // invokes Tk configure
  virtual void geometry(int w, int h); // determines w x h

  widget* pack(char* options = "" ); // maps it to the screen

  bind(char *b, handler* h, char* args = "" );  // binding

  bind(handler* h, char* args = "" );           // implicit

  void xscroll(scrollbar* s);        // to attach scrollbars
  void yscroll(scrollbar* s);

  void focus(char* options="");
  void grab(char* options="");

  void destroy();             // to remove it from the screen

  void* tkwin();  // gives access to Tk_Window implementation

  widget* self();            // for constructing mega widgets
  void redirect(widget* w);

protected:
  char* thepath();                // delivers the virtual path
  void alias( widget* );          // to create widget command
  virtual install(binding*,char* args="");  // default bindings
  virtual direct(char* bnd, binding*, char* args=""); // effect
};
```

The *widget* class is an abstract class. Calling the constructor *widget* as in

```
widget* w = new widget(".awry");
```

does not result in creating an actual widget but only defines a pointer to the widget with that particular name. If a widget with that name exists, it may be treated as an ordinary widget object, otherwise an error will occur. The constructor *widget(widget* w,char* path)* creates a widget by appending the pathname *path* to the pathname of the argument widget *w*.

The function *path* delivers the pathname of a widget object. Each widget created by Tk actually defines a Tcl command associated with the pathname of the widget. In other words, an actual widget may be regarded as an object which can be asked to evaluate commands. For example a widget ".b" may be asked to change its background color by a Tcl command like

```
.b configure -background blue
```

The functions *eval*, *result* and *evaluate* enable the programmer to apply Tcl commands to the widget directly, as does the *configure* command. The function *geometry* sets the width and height of the widget.

Packing Naming widgets in a hierarchical fashion does not imply that the widgets behave accordingly. In particular, to position widgets properly, they must be packed in relation to one another. Packing results in displaying the widgets on the screen. The widget class interface offers two *pack* functions. The function *widget::pack(char*)* applies to individual widgets. As options one may specify for example `-side` *X*, where *X* is either `top`, `bottom`, `left` or `right`, to pack the widget to the appropriate side of the cavity specified by the ancestor widget. Other options are `-fill x` or `-fill y`, to fill up the space in the appropriate dimensions or `-padx` *N* or `-pady` *N*, for some integer *N*, to surround the widget with some extra space.

As a remark, the *kit::pack* function may only be used to pack widgets to the root window.

Binding events Widgets may respond to events. To associate an event with an action, an explicit binding must be specified for that particular widget. Some widgets provide default bindings. These may, however, be overruled.

The function *bind* is used to associate handlers or bindings with events. The first string parameter of *bind* may be used to specify the event type. Common event types are, for example, *ButtonPress*, *ButtonRelease* and *Motion*, which are the default events for canvas widgets. Also keystrokes may be defined as events, as for example *Return*, which is the default event for the *entry* widget.

The function *widget::bind(handler*, char*)* may be used to associate a handler object or action with the default bindings for the widget. Concrete widgets may not override the *handler* function itself, but must define the protected virtual function *install*. Typically, the install function consists of calls to *bind* for each of the event types that is relevant to the widget. Bindings are effected by the virtual function *direct* that may be redefined to effect the binding for multiple widgets, for example.

For both the *bind* functions, the optional *args* parameter may be used to specify the arguments that will be passed to the handler or action when it is invoked. For the *button* widget for example, the default *install* function supplies the text of the button as an additional argument for its handler.

Compound widgets In addition, the widget class offers four functions that may be used when defining compound or mega widgets. The function call *redirect(w)* must by used to delegate the invocation of the *eval*, *configure*, *bind* and *handler* functions to the widget *w*. The function *self()* gives access to the widget to which the commands are redirected. After invoking *redirect*, the function *thepath* will deliver the path that is determined by `self()->path()`. In contrast, the function *path* will still deliver the pathname of the outer widget.

Calling *redirect* when creating the compound widget class suffices for most situations. However, when the default events must be changed or the declaration of a handler must take effect for several component widgets, the virtual function *install* must be redefined to handle the delegation explicitly. The *alias* function is needed when creating widgets that are also used in Tcl scripts. It creates the command corresponding to the widget's path name. How *redirect* and *alias* actually work will hopefully become clear in the examples.

5.2 Buttons

As the first component of the drawing tool, we will look at the *toolbox*. The *toolbox* is a collection of buttons packed in a frame:

```
class toolbutton : public button {        // the toolbutton
public:
  toolbutton(widget* w, char* name) : button(w,name) {
      text(name);
      bind(w,name);    // the parent becomes the handler
      pack();
  }
};

class toolbox : public frame {               // the toolbox
public:
  toolbox(widget* w, tablet* t) : c(t), frame(w,"toolbox") {
      button* b0 = new toolbutton(this,"draw");
      button* b1 = new toolbutton(this,"move");
      button* b2 = new toolbutton(this,"box");
      button* b3 = new toolbutton(this,"circle");
      button* b4 = new toolbutton(this,"arrow");
  }

  int operator()() {
              c->mode( _event->arg(1) );     // transfer to tablet
              return OK;
              }
private:
  tablet* c;
};
```

Each individual button is an instance of the class *toolbutton*. When a toolbutton is created, the actual button is given the name of the button as its path. Next, the button is given the name as its text, the parent widget w is declared to be the handler for the button and the button is packed. The function *text* is a member function of the class *button*, whereas both *bind* and *pack* are common widget functions. Note that the parameter *name* is used as a pathname, as the text to display, and as an argument for the handler, that will passed as a parameter when invoking the handler object.

The *toolbox* class inherits from the *frame* widget class, and creates a frame widget with a path relative to the widget parameter provided by the constructor. The constructor further creates the toolbuttons.

The *toolbox* is both the parent widget and handler for each individual *toolbutton*. When the *operator()* function of the *toolbox* is invoked in response to pressing a button, the call is delegated to the *mode* function of the *tablet*. The argument given to *mode* corresponds to the name of the button pressed.

Comments The definition of the *toolbutton* and *toolbox* illustrates that a widget need not necessarily be its own handler. The decision whether to define a subclass which is made its own handler or to install an external handler depends on what is considered the most convenient way to access the resources needed. As a guideline, exploit the regularity of the application!

5.3 Menus

The second component of our drawing tool is the *menu_bar*:

```
class menu_bar : public menubar {          // row of menubuttons
public:
  menu_bar(widget* w, tablet* t, toolbox* b) : menubar(w,"bar") {
    configure("-relief sunken");

    menubutton* b1 = new file_menu(this,t);
    menubutton* b2 = new edit_menu(this,b);
    button* b3 = new help_button(this);
  }
};
```

The class *menu_bar* is derived from the *hush* widget *menubar*, which is described in the appendix. Its constructor requires an ancestor widget, a *tablet* and a *toolbox*. The *tablet* is passed as a parameter to the *file_menu*, so that the application can write to a file and read from a file. The *toolbox* is passed as a parameter to the *edit_menu* because the *toolbox* is employed as a handler for the *edit_menu*. In addition, a *help_button* is created, which provides on-line help in a hypertext format when pressed. The help facility will be discussed in section 6.3.

The *menu_bar* consists of menubuttons to which actual menus are attached. The *menubutton* and *menu* widgets are built-in widgets. They are described in the appendix. Each menu consists of a number of entries, which may possibly lead to cascaded menus.

The *file_menu* class defines a menu, but is derived from *menubutton* in order to attach the menu to its *menu_bar* parent:

```
class file_menu : public menubutton {

public:
  file_menu(widget* w, tablet* t) : c(t), menubutton(w,"file") {

    configure("-relief sunken"); text("File"); pack("-side left");

    f = new file_handler(c);   // create a file_handler

    class menu* m = new class menu(this,"menu");
    this->menu(m);             // declares it for the menubutton
    m->bind(this);             // installs this as the handler
```

```
    m->entry("Open");
    m->entry("Save");
    m->entry("Quit");
  }

  int operator()() {

        if (!strcmp( _event->arg(1),"Quit")) tk->quit();
        else f->dispatch( _event ); // transfer to file_handler
        return OK;
  }

protected:
  tablet* c;
  file_handler* f;
};
```

The *file_menu* constructor defines the appearance of the button and creates a *file_handler* (which will be discussed in section 5.6). It then defines the actual menu. The menu must explicitly be attached to the *menubutton* by invoking the function *menubutton::menu*. For creating the menu, the keyword *class* is needed to disambiguate between the creation of an instance of the class menu and the call of the *menubutton::menu* function.

Before defining the various entries of the menu, the *file_menu* instance is declared as the handler for the menu entries. However, except for the entry *Quit*, which is handled by calling the *kit::quit* function, the calls are delegated to the previously created *file_handler*.

The second button of the *menu_bar* is defined by the *edit_menu*. The *edit_menu* requires a *toolbox* and creates a menubutton. It configures the button and defines a menu containing two entries, one of which is a cascaded menu. Both the main menu and the cascaded menu are given the *toolbox* as a handler. This makes sense only because for our simple application, the functionality offered by the *toolbox* and *edit_menu* coincide.

5.4 Defining actions – delegation versus inheritance

The most important component of our *drawtool* application is defined by the *tablet* class:

```
class drawmode {                                // drawing modes
public: enum { draw, move, box, circle, arrow, lastmode };
};

class tablet : public canvas {                       // the tablet
public:

  tablet(widget* w, char* options="");

  int operator()() {                       // according to _mode
          return handlers[_mode]->dispatch( _event );
  }

  void mode(char* m);                 // to set the drawing mode
```

```
protected:
  void init(char* options);          // initializes the tablet
  int _mode;
  class handler* handlers[drawmode::lastmode];   // keeps modes
  canvas* c;                                    // the actual canvas
};
```

The various modes supported by the drawing tool are enumerated in a separate class *drawmode*. The *tablet* class itself inherits from the *canvas* widget class. This has the advantage that it offers the full functionality of a canvas. In addition to the constructor and *operator()* function, which delegates the incoming event to the appropriate handler according to the *_mode* variable, it offers a function *mode*, which sets the mode of the canvas as indicated by its string argument, and a function *init* that determines the creation and geometrical layout of the component widgets. As instance variables, it contains the integer *_mode* variable and an array of handlers that contains the handlers corresponding to the modes supported. See section 6.2 for an example of a typical canvas handler.

Dispatching Although the *tablet* must act as a canvas, the actual *tablet* widget is nothing but a *frame* that contains a canvas widget as one of its components. This is reflected in the definition of the *tablet* constructor and the way it invokes the *canvas* constructor.

```
tablet::tablet(widget* w, char* options) : canvas(w,"tablet",0) {
  widget* top = new frame(path());

  init(options);                          // inialization, layout
  redirect(c);                            // redirect to canvas
  bind(this);                             // this is the handler

  handlers[drawmode::draw] = new draw_handler(this);
  handlers[drawmode::move] = new move_handler(this);
  handlers[drawmode::box] = new box_handler(this);
  handlers[drawmode::circle] = new circle_handler(this);
  handlers[drawmode::arrow] = new arrow_handler(this);
  _mode = drawmode::draw;
}
```

By convention, when the options parameter is *0* instead of the empty string, no actual widget is created but only an abstract widget, as happens when calling the *widget* class constructor. Instead of creating a *canvas* rightaway, the *tablet* constructor creates a top frame, initializes the actual component widgets, and redirects the *eval*, *configure*, *bind* and *handler* invocations to the *canvas* child widget. It then declares itself to be its own handler, which results in declaring itself to be the handler for the canvas component. Note that reversing the order of calling *redirect* and *handler* would be disastrous, since the bindings resulting from calling *handler* would then be defined not for the *canvas* but for the *frame* containing the *canvas*. After that it creates the handlers for the various modes and sets the initial mode to *draw*.

The *operator()* function takes care of dispatching calls to the appropriate handler. The *dispatch* function must be called to pass the *event* information.

5.5 Creating new widgets

Having taken care of the basic components of the drawing tool, that is the *toolbox*, *menu_bar* and *tablet* widgets, all that remains to be done is to define a suitable *file_handler*, appropriate handlers for the various drawing modes and a *help_handler*. This will be done in sections 5.6, 6.2 and 6.3, respectively.

However, before that we will look at how to define the *drawtool* widget class such that we may also declare a corresponding *drawtool* script command. The actual declaration of the *drawtool* command is done in the application class defined below, which will by now look familiar, except for the function *prelude*:

```
class application : public session {
public:
  application(int argc, char* argv[])
                               : session(argc,argv,"drawtool") {}

  void prelude( ) {
          tk->bind("drawtool", new drawtool());    // declare
  }

  void main( kit* tk, int, char* argv[] ) {
          drawtool* d = new drawtool(".draw");
          tk->bind("drawtool",d);                 // override
          d->pack();
  }
};
```

In the body of the *prelude* function, the Tcl command *drawtool* is declared, with an instance of *drawtool* as its handler.

In this way, the *drawtool* widget is made available as a command when the program is used as an interpreter. However, in the function *main* this declaration is overridden. Instead, the actual *drawtool* widget is made the handler of the command, in order to allow for a script to address the *drawtool* by calling *drawtool self*, as will be explained later.

Since an instance of *drawtool* may also be used as simply a handler for the *drawtool* command, the *drawtool* class must offer a constructor that creates no widget, in addition to a constructor that does create a *drawtool* widget:

```
class drawtool : public canvas {
public:
  drawtool() : canvas() {  }                        // no widget
  drawtool(char* p, char* opts="") : canvas(p,0) {
      top = new frame(path(),"-class Drawtool");  // outer frame
          init(opts);
          redirect(c);                      // redirect to tablet
      alias( top );              // to declare widget command
  }

  // Define the semantics of the drawtool command

  int operator()(){
          if (!strcmp("self",argv[1]) )                   // self
                         tk->result(self()->path());
          else if ( !strcmp( "drawtool" ,*argv) )           // create
```

```
                        create(--argc,++argv);
        else                                                // eval
                        self()->eval( flatten(--argc,++argv) );
        return OK;
  }

protected:
  wiget* top;                                          // outer frame
  tablet* c;                                       // inner component

  void init(char* options);

  // To create a new drawtool widget and corresponding command

  void create(int argc, char* argv[]) {
    char* name = *argv;
    new drawtool(name, flatten(--argc,++argv));
  }
};
```

The *drawtool* widget constructor redirects itself to the *tablet* widget, which is initialized by calling *init*.

The *drawtool::operator()* function defines the semantics of the *drawtool* script command. When the first argument of the call is *drawtool*, a new *drawtool* widget is created, except when the second argument is *self*. In that case, the virtual path of the widget is returned, which is actually the path of the *tablet*'s canvas. As an example of a script employing *self* consider

```
set x [drawtool self]
$x create rectangle 100 20 160 80
$x create rectangle 90 30 150 90
$x create oval 120 40 170 90
```

Evaluating the script results in the drawing displayed in figure 6. Such a script may be read in by using the *Open* option in the *File* menu (see section 5.6).

If neither of these cases apply, the function *widget::eval* is invoked for *self()*, with the remaining arguments flattened to a string. This makes it possible to use the *drawtool* almost as an ordinary canvas as illustrated above and in the example hypertext script shown in section 6.3.

The creation of the actual widget and declaration of the corresponding Tcl command, according to the Tk convention, is somewhat more involved.

Recall that each Tk widget is identified by its path, which simultaneously defines a command that may be used to configure the widget or, as for a canvas, to draw figures on the screen. Hence, the function *create* must create a new widget and declare the widget to be the handler of the command corresponding to its pathname.

Discussion By now you may have lost track of how delegation within a compound widget takes place. Hopefully, a brief look at the implementation will clarify this.

Each *eval*, *configure* or *bind* function call for a widget results in a command addressed at the path of the widget. By redirecting the command to a different path, the instructions may be delegated to the appropriate (component) widget. Delegation occurs, in other words, by directing the commands to the widget's virtual path, which is obtained by the protected function *thepath()*. In contrast, the function *path()* delivers the path of the widget's outer component. Indirection takes place by invoking the function *self()*, which relies on an instance variable *_self* that may be set by the *redirect* function.

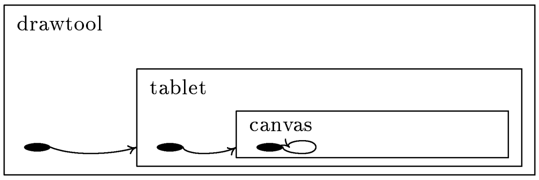

Figure 8: Dereferencing *self()*

Figure 8 represents the evaluation of *self()* for *drawtool* in a pictorial way. Dereferencing ultimately results in addressing commands to the *canvas* widget, because of the redirections declared for *drawtool* and *tablet*. The implementation of *thepath()* and *self()* is simply:

```
char* thepath() { return self()->path(); }
widget* self() { return _self?_self->self():this; }
```

Hence, resolving a compound widget's primary inner component relies on simple pointer chasing, which may be applied recursively to an arbitrary depth at acceptable costs.

5.6 Dialogs

Interactive applications may require the user to type some input after reading a message or to select an item from a list of alternatives. One of the widgets that may be used in a dialog with the user is a *file_chooser* widget as depicted in figure 9.

Despite its simple appearance, the *file_chooser* widget has some subtle complexities. The *file_chooser* class is given below:

```
class file_chooser : public toplevel {              // toplevel

  file_chooser() : toplevel( gensym("filechooser") ) { init(); }

  int operator()();
  char* get() { return e->get(); }

protected:
  button* b; button *c;                            // OK and CANCEL
  entry* e; listbox* l;
  int install(char* s, binding* a, char* opts);
  void init();
  void list();
};
```

The *file_chooser* widget consists of a *listbox* filled with filenames and an *entry* widget that contains the filename selected by the user (by double clicking on the name) or which may, alternatively, be used to type in a filename directly. In addition, the *file_chooser* has an *OK* button, to confirm the choice and a *CANCEL* button, to break off the dialog.

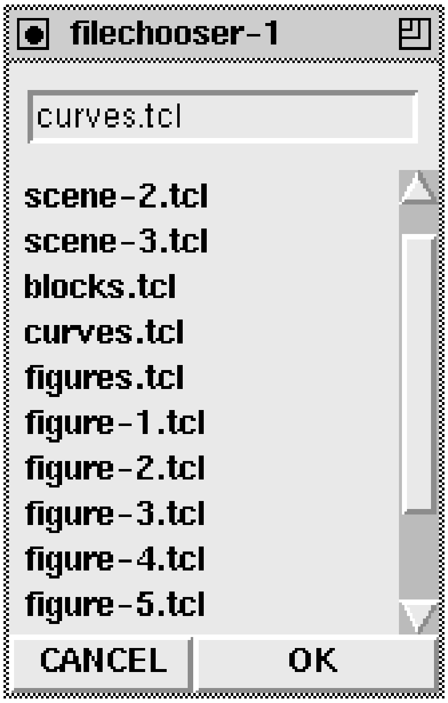

Figure 9: A filechooser

Typically, a *file_chooser* is a toplevel widget, that is a widget that is independently mapped to the screen. To avoid name clashes the function *gensym*, which delivers a system-wide unique name (with *filechooser* as a prefix), is used to determine its path. Apart from the *operator()* function, the *file_chooser* has only one public function *get*, which delivers the name selected or typed in by the user.

The widget components of the *file_chooser*, two buttons and the *entry* and *listbox* widgets, are stored in its instance variables. Further, we have a function *init* to construct the actual *file_chooser* widget, a function *list* to fill the *listbox* and the function *install*, which is used to install an external handler for the two button widgets. The *install* function is defined as

```
void file_chooser::install(binding* a, char* args) {
         b->handler(a,args);
          c->handler(a,args);
}
```

Recall, that when declaring a handler for a button, the name of the button is given as an additional argument when invoking the handler. This enables the *file_handler* to distinguish between a call due to pressing the *OK* button and a call due to pressing the *CANCEL* button.

The interplay between the C++ definition and the underlying Tcl/Tk toolkit is nicely illustrated by the definition of the *list* function.

```
void file_chooser::list() {
  sprintf(buf,"foreach i [glob *.tcl] { %s insert end $i }",
                                                  l->path());
  tk->eval( buf );
```

```
}
```

Calling *list* results in filling the *listbox* with the filenames in the current directory. Its corresponding definition in C++ would, no doubt, be much more involved.

The *file_handler* class Window-based interactive applications differ from ordinary interactive applications by relying on an event-driven flow of control. The indirection that is typical for event-driven control is exemplified in the definition of the *file_handler* depicted below (recall that the *file_handler* was employed by the *file_menu* described in section 5.3):

```
class file_handler : public handler {
public:
  file_handler( canvas* x ) : c(x) {}

  int operator()() {
      char* key = _event->arg(1);
          if (!strcmp("Open", key)) launch("OPEN");
          else if (!strcmp("Save", key)) launch("SAVE");
          else if (!strcmp("OPEN", key)) open();
          else if (!strcmp("SAVE", key)) save();
          return OK;
  }

protected:
  canvas* c;
  file_chooser* f;

  void launch(char* args) {       // launch new filechooser
          f = new file_chooser();
          f->handler(this, args);
  }

  void open() { tk->source( f->get() ); f->destroy(); }
  void save() { c->postscript( f->get() ); f->destroy(); }
};
```

Since the *file_handler* does not correspond to an actual widget when created, its constructor merely stores the canvas pointer, which is actually a pointer to the *tablet*.

In response to the *Open* or *Save* menu entries, the *file_handler* launches a *file_chooser* and declares itself to be the handler (with the appropriate arguments). For example, when selecting the *Open* entry, the *file_chooser* is launched which eventually calls the *file_handler* :: *dispatch* function with *OPEN* as its argument. The *file_handler* then invokes the *open* function, which results in reading in the file and destroys the *file_chooser*. In a similar way, the menu entry *Save* results in writing the canvas to a postscript file. (The code for checking whether the *OK* or *CANCEL* button is pressed is left as an exercise.)

6 Graphics and hypertext

The Tk toolkit offers powerful facilities for graphics and (hyper)text. See [Ousterhout93]. In this section we discuss the *canvas* widget offered by Tk.

Instead of looking at the *text* widget provided by Tk, we will (briefly) look at the *hypertext* widget, which presents an alternative approach to defining hyperstructures.

6.1 The *item* class

The canvas widget allows the programmer to create a number of built-in graphic items. Items are given a numerical index when created and, in addition, they may be given a (string) tag. Tags allow items to be manipulated in a group-wise fashion. To deal with items in a C++ context, the *hush* library contains a class *item* of which the functionality is shown below.

```
interface item {
  operator int();                        // returns item index

  void configure(char* cmd);    // calls canvas::itemconfigure

  void tag(char* s);                      // sets tag for item
  char* tags();              // delivers tags set for the item
  void move(int x, int y);

  bind(char *b, handler* h, char* args = "" );
  bind(char *b, binding* ac, char* args = "" );

  handler(class handler* h, char* args = "" );
  handler(binding* ac, char* args = "" );

protected:
  virtual install(action&,char* args="");    // default bindings
};
```

Instances of *item* may not be created directly by the user, but instead are created by the canvas widget. For an item, its index may be obtained by casting the item to *int*. If the index does not identify an existing item, it will be zero. Existing items may be moved, in a relative way, by the function *move*.

In a similar way as for widgets, items may be associated with events, either explicitly by using *item::bind*, or implicitly by using *item::handler*. The default bindings for *items* are identical to the default bindings for the canvas widget, but these may be overridden by descendant classes.

Similar as the *widget* class, the *item* class is derived from the *handler* class. This allows the user to define possibly compound shapes that have their own handlers.

6.2 The *canvas* widget

The Tk canvas widget offers powerful means for doing structured graphics. The *hush* class *canvas* provides merely a simplified interface to the corresponding Tk widget.

As an example of the use of a canvas, consider the definition of the *move_handler* below. The *move_handler* defines one of the modes of the *tablet* and allows for moving graphical items on the canvas.

```
class move_handler : public handler {
public:
```

```
  move_handler( canvas* cv ) : c(cv) { dragging = 0; }

  void press( event& e ) {     // if overlapping start dragging
            x = e.x(); y = e.y();
            id = c->overlapping(x, y);
            if (id) dragging = 1;
  }

  void motion( event& e ) {                  // if dragging move
    if (dragging) {
          id.move( e.x() - x, e.y() - y );
          x = e.x(); y = e.y();
          }
  }

  void release( event&  ) { dragging = 0; }     // stop dragging

protected:
  canvas* c; int dragging; item id; int x,y;
};
```

The *move_handler* class is derived from the class *handler*. It makes use of the *dispatch* and *operator()* function defined for *handler*, but redefines the (virtual) functions *press*, *motion* and *release*.

When creating an instance of *move_handler*, a pointer to the canvas must be given to the constructor. In addition, the class has data members to record position coordinates and whether a particular item is being moved. Actually moving an item occurs by pressing the (left) mouse button on an item and dragging the item along. When the mouse button is released, moving stops. To identify the item, the function *overlapping* is used. The movement is determined by the distance between the last recorded position and the current position of the cursor.

In an analogous manner, a *box_handler* may be defined, which is used for drawing rectangles and allows for rubberbanding. The *box_handler* sets dragging to true when the button is pressed and creates a rectangle of zero width and height. Each time the function *motion* is called, the item created in the previous round is deleted and a new rectangle is created:

```
void box_handler::motion( event& e ) {   // if dragging stretch
    if (dragging) {
      id.del();
      id = c->rectangle(x,y,e.x(),e.y());  // x and y are fixed
          }
  }
```

where c is a pointer to the canvas and x and y the button pointer coordinates stored when dragging began. For circles and lines, it suffices to replace the call to rectangle with a call to the appropriate figure creation function.

6.3 The *hypertext* widget

Both the Tk canvas and text widget allow to bind actions to particular items and hence to define dynamically what we may call *hyperstructures*.

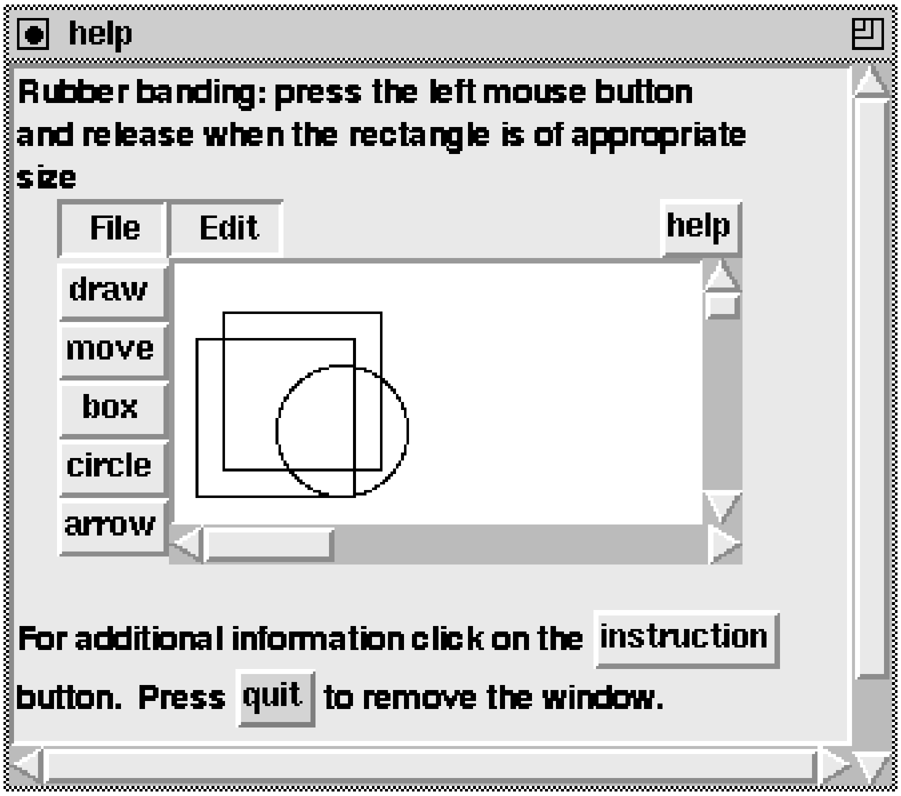

Figure 10: Hypertext help

A different, in a way more static, approach is offered by the *hypertext* widget originally developed by George Howlett.

The *hypertext* widget may be used to display text files containing embedded Tcl code. The Tcl code must be placed between escapes, that take the form of `%%` for both the begin and end of the code. A screen shot of a fragment of the on-line help for *drawtool* is given in figure 10. Notice, that the on-line help provides a replica of the *drawtool* application, surrounded by text. When looking at (again a fragment of) the hypertext file specifying the contents of the on-line help, given below, you see that the *drawtool* command defined in section 5.5 is employed to create the embedded widget:

```
Rubber banding: press the left mouse button
and release when the rectangle is of appropriate
size
    %%
drawtool $this.draw
$this append $this.draw
$this.draw create rectangle 20 20 80 80
$this.draw create rectangle 10 30 70 90
$this.draw create oval 40 40 90 90
$this append $this.draw
%%
For additional information click on the %%
button $this.goto -text instruction -command end-of-text
$this append $this.goto
```

```
%%
button.  Press %%
button $this.quit -command { destroy . } -text quit -bg pink
$this append quit
%% to remove the window.
```

When specifying the hypertext file, widgets may be given a pathname relative to the pathname of the hypertext widget by using the variable *this*. In addition the hypertext widget offers the variables *thisline* and *thisfile* to identify the current line number and current file name.

Any of the widgets and commands offered by Tcl/Tk or supported by *hush* may be included in a hypertext file, including the ones defined by the program itself.

7 Conclusions

The Tcl/Tk toolkit offers very versatile means to create graphical user interfaces and couple these with programs written in C. However, from the point of view of object oriented programming and the use of Tcl/Tk in a C++ context, the standard interface does not suffice.

The *hush* library is meant to provide a flexible, yet easy to use, and above all simple, interface for Tcl/Tk in C++. To some extent, it may be regarded as syntactic sugar of an object oriented flavor, merely simplifying the interface already provided by Tcl/Tk. However, *hush* improves on the standard Tcl C interface by providing the opportunity to employ *handler* objects, allowing the programmer to deal in a type-secure way with client information associated with events.

In addition, the *hush* library allows for the definition of composite widgets with the behavior of one of the standard widgets. To support such composite widgets, each widget has a virtual path that coincides with the widget's own path, unless it is redirected to an inner component widget. Composite widgets may be nested to arbitrary depth. This solution has the advantage that the composite widget may be given an already familiar interface, both in C++ and Tcl, with a minimum of coding.

The approach embodied by *hush* is intended to allow the novice programmer to develop graphical user interfaces easily, however, without restricting experienced and more demanding programmers, who may gradually exploit the full functionality offered by Tk and extend this by using C++.

Acknowledgement I would like to thank Paula Ferguson from The X Resource for her detailed advice on improving the readability of the original article. Acknowledgements are also due to Jacco van Ossenbruggen, Martijn van Welie and Bastiaan Schönhage for their suggestions, their contributions to the extensions of hush, and their unrelenting insistence on conceptual clarity.

References

[Eliens95] Eliëns A. (1995) Principles of Object-Oriented Software Development Addison-Wesley

[LVC89] Linton M., Vlissides J. and Calder P. (1989) Composing user interfaces with Interviews. IEEE Computer 22(2), pp. 8-22

[Ousterhout90] Ousterhout J.K. (1990) Tcl: an embeddable command language. In Proc. USENIX Winter Conference, pp. 133-146

[Ousterhout91] Ousterhout J.K. (1991) An X11 Toolkit based on the Tcl language. In Proc. USENIX Winter Conference, pp. 105-115

[Ousterhout93] Ousterhout J.K. (1993) Hypergraphics and hypertext in Tk. The X Resource 5, pp. 113-127

[Ousterhout94] Ousterhout J.K. (1994) Tcl and the Tk toolkit. Addison-Wesley

[Stroustrup91] Stroustrup B. (1991) The C++ Programming Language. Addison-Wesley, 2nd edn

A Appendix: The *hush* widget classes

The *hush* widget class library encapsulates the standard Tk widgets. In addition, a hypertext widget is offered. The widget classes are organized as a tree, with the class *widget* at the root.

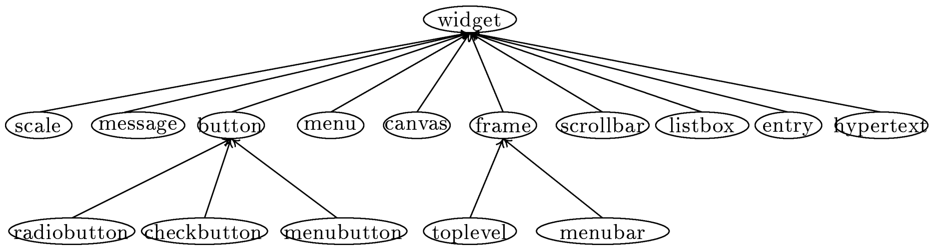

Figure 11: Widget classes

Each concrete widget class offers the functionality supported by the (abstract) *widget* class and may in addition define functions specific to the particular widget class. The member function for a widget class have usually a straightforward correspondence with the command interface defined by the Tcl/Tk toolkit. See [Ousterhout94] for the most recent description.

Each widget class specifies two constructors, one with only a path and one which allows both for a widget and a path. In the latter case, the actual path consists of the concatenation of the path of the widget and the path specified by the string parameter. For the concrete widget classes, no widget will be created when the *options* parameter is zero. This convention is adopted to allow composite widgets to inherit from the standard widgets, yet define their own components.

In addition, each widget class has a destructor, which is omitted for brevity. The destructor may be used to reclaim the storage for a widget object. To remove a widget from the screen, the function *widget::destroy* must be used.

The ***scale*** **class** The *scale* widget may be used to obtain numerical input from the user.

```
interface scale : widget {

  scale(char* p, char* options = "");
  scale(widget* w, char* p, char* options = "");
```

```
    void text(char* s);                              // text to display
    void from(int n);                                // begin value
    void to(int n);                                  // end value
    int get();                                       // gets the value
    void set(int v);                                 // sets the value

  protected:
    install(binding*, char* args);
  };
```

When a handler is attached to a *scale* it is called when the user releases the slider. The value of the *scale* is passed as an additional parameter when the handler is invoked. The default binding for the scale is the *ButtonRelease* event.

The *message* class The *message* widget may be used to display a message on the screen.

```
  interface message : widget {

    message(char* p, char* options = "" );
    message(widget* w, char* p, char* options = "" );

    void text(char* s);                                   // the text
  };
```

The *message* class does not define default bindings, but the user is free to associate events to a message widget by employing *widget::bind*.

The *button* class Buttons come in a number of varieties, such as ordinary (push) buttons, that simply invoke an action, checkbuttons, that toggle between an on and off state, and radiobuttons, that may be used to constrain buttons to allow the selection of only a single alternative. Checkbuttons and radiobuttons are implemented as subclasses of the class *button*, and will not be further discussed here.

```
  interface button : widget {

    button(char* p, char* options = "");
    button(widget* w, char* p, char* options = "");

    void text(char* s);                          // text to display
    void bitmap(char* s);                        // to display a bitmap

    void state(char *s);             // to change the buttons state

    void flash();
    char* invoke();

  protected:
    install(binding*,char* args);             // default binding
  };
```

In addition to the constructors, which have the same format for each widget class, the *button* class offers the function *text* to define the text displayed by the button and the function *bitmap*, which takes as argument the name of a file containing a bitmap, to have a bitmap displayed instead. The function *state* may be used to change the state of the button. Legal arguments are either *normal*, *active* or *disabled*. Further, the *button* class defines the function *flash* and *invoke* that result respectively in flashing the button and in invoking the action associated with the button by means of the *widget::handler* function. (Note that *button::install* is defined, albeit protected.)

The *menubutton* class The *menubutton* is a specialization of the *button* widget. It allows for attaching a menu that will be displayed when pressing the button.

```
interface menubutton : button {

  menubutton(char* p, char* options = "");
  menubutton(widget* w, char* p, char* options = "");

  void menu(char* s);                    // to attach a menu
  void menu(class menu* m);
};
```

The *menubutton* must be used to pack menus in a *menubar*.

The *menu* class A menu consists of a number of button-like entries, each associated with an action. A menu entry may also consist of another menu, that pops up whenever the entry is selected.

```
interface menu : widget {

  menu(char* p, char* options = "");
  menu(widget* w, char* p, char* options = "");

  menu* add(char* s, char* options = "");

  menu* entry(char* s, char* args ="", char* options="");
  menu* entry(char* s, binding* ac, char* args="", char* opts="");

  menu* cascade(char* s, char* m, char* options = "");
  menu* cascade(char* s, menu* m, char* options = "");

  char* entryconfigure(int i, char* options);

  int index(char *s);
  int active();                     // returns active index

  void del(int i);                  // delete entry with index i
  void del(char* s);                // delete entry with tag s

  char* invoke(int i);              // invoke entry with index i
  char* invoke(char *s );           // invoke entry with tag s

  void post(int x = 500, int y = 500);
  void unpost();
```

```
protected:
  install(binding*, char* args);
};
```

The *add* function is included to allow arbitrary entries (as defined by Tk) to be added. We restrict ourselves to simple command and cascade entries.

The *entry* function (that is used for adding simple command entries) may explicitly be given a *binding* to be associated with the entry. Alternatively, if no binding is specified, the default handler binding installed by invoking *widget::bind* will be used. The string used as a label for the entries (the first parameter of *entry*) will be given as a parameter to the action invoked when selecting the entry. The string given in the *args* parameter will be added to the actual parameters for the action invoked.

The *cascade* function may either be given a *menu* or a string, containing the pathname of the menu. In any case the cascaded menu must be a child of the original menu.

The function *index* returns the integer index associated with the string describing the entry. The function *active* may be used to inquire which entry has been selected.

Entries may be deleted using the function *del* and invoked by using *invoke*. For both functions, the entry may be indicated by its numerical index or a string. Menus are toplevel widgets, they are mapped to the screen either by invoking the function *post*, or by pressing the *menubutton* to which the menu is attached.

The *canvas* class Apart from the two standard constructors, it offers the functions *tag*, *tags* and *move* that merely repeat the functions offered by the *item* class, except that *move* may also be given a tag to identify the items to be moved.

```
interface canvas : widget {

  canvas(char *p, char* options="");
  canvas(widget* w, char *p, char* options="");

  void tag(int id, char* tag);
  char* tags(int id);

  void move(int id, int x, int y);
  void move(char* id, int x, int y);

  item bitmap(int x1, int y1, char* bitmap, char* options="");
  item line(int x1, int y1, int x2, int y2, char* options="");
  item line(char* linespec, char* options="");
  item circle(int x1, int y1, int rad, char* options="");
  item oval(int x1, int y1, int x2, int y2, char* options="");
  item polygon(char* linespec, char* options="");
  item rectangle(int x1, int y1, int x2, int y2, char* options="");
  item text(int x1, int y1, char* txt, char* options="");
  item window(int x1, int y1, char* win, char* options="");
  item window(int x1, int y1, widget* win, char* options="");

  item current();
  item overlapping(int x, int y);
```

```
    itemconfigure(int it, char* options);
    itemconfigure(char* tag, char* options);
    itembind(int it, char* s, binding* a, char* args = "" );
    itembind(char* tag, char* s, binding* a, char* args = "" );

    void postscript(char* file, char* options="");

  protected:
    install(binding*, char* args);
  };
```

Currently, the graphic items *bitmap*, *line*, *oval*, *polygon* and *rectangle* may created and, in addition, *text* items and *window* items consisting of a widget. The function *overlapping* may be used to retrieve the item overlapping a particular position.

In addition, the *canvas* class offers auxiliary functions needed to support the functionality provided by the *item* class. The canvas may be written as Postscript to a file with the function *canvas::postscript*.

The *frame* class Frame widgets may used to combine widgets. A frame has no functionality or bindings of its own.

```
  interface frame : widget {

    frame(char* p, char * options = "");
    frame(widget* w, char* p, char * options = "");
  };
```

The frame widget class has the *toplevel* and *menubar* as subclasses.

The *toplevel* widget is used when the widget must be independently mapped to the screen.

The *menubar* widget is used as a special frame to collect *button* widgets including *menubutton* widgets.

The *scrollbar* class The *scrollbar* allows the user to scroll through widgets that are only partly displayed.

```
  interface scrollbar : widget {

    scrollbar(char* p, char* options = "");
    scrollbar(widget* w, char* p, char* options = "");

    void orient(char* opts="vertical");   // orientation
    xview(widget* w);                     // widget to scroll
    yview(widget* w);                     // widget to scroll
  };
```

The default orientation of a *scrollbar* is vertical. A scrollbar must be explicitly attached to a widget *w* by calling the *scrollbar::yview* functions for vertical scrollbars and *scrollbar::xview* for horizontal scrollbars. To obtain the proper geometrical layout, the scrollbar and the widget it controls must usually be packed in a frame.

The *listbox* class The *listbox* widget is used to allow the user to select an item from a list of alternatives.

```
interface listbox : widget {

  listbox(char* p, char* options = "");
  listbox(widget* w, char* p, char* options = "");

  void insert(char* s);
  char* get(int d);             // entry with index d

  void singleselect();

protected:
  install(binding*, char* args);
};
```

The *listbox* may be filled by using *insert*. When a handler is attached to the widget, it is activated when the user double clicks on an item. The selected entry is passed as an additional parameter to the handler. The entry may also be obtained by either *kit::selection* or *listbox::get*.

The *entry* class An *entry* widget may be used to display text or allow the user to type a short text.

```
interface entry : widget {

  entry(char* p, char* options = "");
  entry(widget* w,char* p, char* options = "");

  void insert(char* s);                    // insert text
  char* get();                           // to get the text

protected:
  install(binding*, char* args);
};
```

When a handler is attached to the entry widget it is activated whenever the user double clicks on the widget or presses the return key. The contents of the entry are added as an argument when calling the handler.

The *hypertext* class The *hypertext* widget may be used to display text with embedded Tcl code.

```
interface hypertext : widget {

  hypertext(char* p, char* options = "");
  hypertext(widget* w, char* p, char* options = "");

  void file(char* f);         // to read in hypertext file
};
```

Apart from the standard constructors, it offers the function *file* to read in a hypertext file. Such a hypertext file allows to embed widgets in the text by inserting them in escape sequences.