VSH : A MULTIPARADIGM FRAMEWORK FOR GRAPHICAL USER INTERFACES

by

Vipul Jain

School of Computer Science
McGill University
Montréal, Québec
Canada

April 1995

A Thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for
the Degree of Master of Science

Abstract

In this thesis we present a multiparadigm framework *vsh* for graphical user-interface design. It provides a powerful and flexible framework that allows programming in C++ as well as in Tcl. Tcl is a scripting language that may be embedded in C or C++. Tk is a windowing and graphics toolkit based on X11, with an associated interpreter called *wish*. The *vsh* library contains classes that provide convenient yet flexible access to the functionality offered by the Tcl/Tk toolkit and its extensions. Included in the framework is *Wish* + +, an object oriented extension of the windowing shell *wish* based on Tcl/Tk. 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 also allows more experienced programmers to employ the Tcl scripting language to define the behavior and functionality of widgets and structured graphics objects. The design of *wish*++ 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 *vsh* on the Tk toolkit is that existing Tk application written for the Tk interpreter *wish* can easily be (re)used in C++ context, virtually without any cost. On the other hand, programs employing *vsh* may be used as an enhanced version of the *wish* interpreter, allowing the functionality defined in the program to be used in a (*wish* + +) script.

Résumé

Dans cette thèse nous présentons le cadre *vsh* à paradigmes multiples pour la conception graphique d'interfaces utilisateur. *vsh* offre un cadre puissant et flexible permettant la programmation en C++ ainsi qu'en Tcl. Tcl est un langage de scripts qui peut être incorporé à C ou à C++. Tk est une collection d'outils pour fenêtres et graphiques basée sur X11, accompagnée d'un interpréteur appelé *wish*. La bibliothèque *vsh* comprend des classes permettant un accès pratique et flexible aux outils de Tcl/Tk et à ses extensions. Inclue dans ce cadre de développement se trouve *wish* + +, une extension orientée-objet de la coquille de fenêtre *wish*, elle-même basée sur Tcl/Tk. La bibliothèque a pour but de subvenir tant aux besoins des programmeurs (de fenêtres) expérimentés qu' à ceux de programmeurs débutants. Elle offre des classes de "widgets" et de graphiques avec une interface facile à utiliser, mais permet également aux programmeurs plus expérimentés d'employer le langage de script Tcl afin de définir le comportement des "widgets" et des objects graphiques structurés. La conception de *wish* + + est inspirée de la bibliothèque "InterViews". Cependant l'utilisation d'événements "callbacks" et l'interface de "widgets" et de classes graphiques est beaucoup plus facile. Le fait que *vsh* soit basé sur la collection d'outils Tk présente l'avantage suivant: les applications existantes de Tk écrites pour l'interpréteur *wish* peuvent facilement être (ré)utilisées dans un contexte C++, à un coût pratiquement inexistant. De plus, les programmes utilisant *vsh* peuvent être utilisés comme une version améliorée de l'interpréteur *wish*, permittant ainsi aux raffinements définis dans ce programme d'être utilisés dans un script *wish* + +.

Contents

List of Figures

Chapter 1

Introduction

In comparison with ordinary programming (in C++) [Str91, Lip91], programming in a window environment (in C++) [Mye92] 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, canvases, etcetera. And secondly, perhaps the most difficult aspect of window programming, the programmer must deal with a rather different control structure, involving actions and callback in response to events generated by the user or windowing environment, such as mouse button manipulations.

A number of toolkits for X11 environment with an interface to C++ do exist already. A well known example is the InterViews Library, which offers powerful features for defining the layout of graphical user interfaces [LVC89]. However, despite the elegance of its design, InterViews is 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 form their price, is primarily they do not offer the flexibility needed in a research environment.

A rather different approach to GUI programming has been advocated by Ousterhout [Ous91],[Ous94] which describes the Tcl/Tk toolkit. Tcl is a cshell-like (interpreted) script language that may be embedded in C or C++ [Str91]. 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 scripting language allows one to rapidly prototype rather complex graphical user interfaces. 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 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 *dpwish*, extensions offering object oriented features *[incr tcl]*, and extensions offering additional widgets such as barchart and hypertext widget *blt_wish*.

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 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, as it is the case in user interfaces, an 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 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 can be recommended from the software engineering perspective. Secondly, although not very difficult, creating a new interpreter with additional C/C++ code is somewhat cumbersome.

The *vsh* library has been developed to address the two problem mentioned. The standard interpreter associated with the *vsh* library is a shell, called *wish++*, including a number of available extensions of Tcl/Tk and widgets developed by me (such as a *filechooser* and an MPEG video widget). The *vsh* library offers a C++ interface to the Tcl/Tk toolkit and its extensions. It allows the programmer to employ the functionality of Tcl/Tk in a C++ program. Moreover a program created with *vsh* is itself an interpreter extending the *wish* interpreter, the *wish* + +.

The *vsh* 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 allows for employing Tcl script code when more is demanded.

The contribution of *vsh* with respect to Tcl/Tk toolkit is essentially that it provides type secure solutions for connecting Tcl and C++ code. As an additional advantage, the *vsh* 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 a standard Tk widgets.

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

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

However, both the use of event callbacks and the functional interface of widget and graphics classes is significantly simpler. An important advantage of basing *vsh* on the Tk toolkit is that existing Tk application written for the Tk interpreter *wish* can easily be (re)used in C++ context, virtually without any costs. On the other hand programs employing *wish++* may again be used as (an enhanced version of) the *wish* interpreter, allowing the functionality defined in the program to be used in a (*wish++*) script. See Figure 1.1.

Wish++ - *the C++ API*
- session,kit,event,action,handler

Widgets - *for the GUI*
- menu,message,text,filechooser,barchart,xygraph

Graphic Items - *canvas*
- oval,bitmap,text,rectangle,line,polygon

Figure 1.1: The Wish++ API

Structure of Thesis In chapter 2, some background information concerning Tcl/Tk is given, including a brief example. Chapter 3 sketches the structure of typical *wish++* program, including the *kit* and *session* class. Chapter 4 describes how handler objects may be defined as event call backs. Chapter 5 illustrates the various widget classes and their properties. Chapter 6 demonstrates how to construct compound widgets. In chapter 7 we will look at the facilities offered for structured graphics and hypertext. Chapter 8 shows how a widget developed in C++ may be made available as a widget to be used in scripts. Next, chapter 9 presents a drawing tool

application as a medium to depict the functionality offered by the framework to model interaction. Chapter 10 gives a brief overview of the related work. And finally, chapter 11 concludes with a brief discussion on the scope of future enhancements.

Chapter 2

Background-Tcl/Tk

The language Tcl has first been presented in [Ous90]. Tcl was announced as 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 [Ous91]. 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.

Tcl - *an extensible script language*
- variables,procedures,built-ins

Tk - *an X-window toolkit*
- widgets,window management

Figure 2.1: Tcl/Tk

The Tcl language offers variables, assignments and a procedure construct. It also 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 (new) 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.

Wish The wish interpreter is an interpreter for executing Tcl/Tk script, look at the *hello world* program in Figure 2.2.

Scripts - *hello world*

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

Figure 2.2: A Wish example

The *hello world* script defines a button that displays *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, for example the *button* command (needed to create a button) and the *pack* command (that is used to map the button on the screen).

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

Tcl C Application Programmers Interface(API) To define Tcl commands in C, the programmer has to define a command functions, which has the profile function *aCommand* shown in Figure 2.3, and declare the function to be a command by invoking the *Tcl_CreateCommand* function as indicated.

Defining a command - *function profile*
int aCommand(ClientData, Tcl_Interp, int args, char* argv[]);

Declaring a command - *creation of binding*
Tcl_CreateCommand(interp, "aco", aCommand, (ClientData) w);

Figure 2.3: The Tcl C API

Creating a command is done with reference to an interpreter, which accounts for the first argument of *Tcl_CreateCommand*. The name of the command, as may be used in a Tcl script must be given as second argument, and the C/C++ function defining the command as a third argument. Finally, when declaring a command, the address of a structure containing client data may be stored, which for example may be (the address of) the root window.

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. 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 a simply inelegant.

The *vsh* 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 *vsh* 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.

Chapter 3

Program structure

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

As an illustration of the structure of a program using *vsh*, we 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 *vsh* library. A more detailed description will be given of the *kit* class, that encapsulates the embedded Tcl interpreter, and the *session* class, that shields of the details of the window environment.

3.1 Employing Tcl/Tk from within C++

Imagine that a user has 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 Figure 3.1.

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

Figure 3.1: The function *newton*

When such a function is written, one may wish to have a graphical interface to allow him to experiment with possible inputs in a flexible way. For example, he or she 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 one in Figure 3.2.

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 to connect the graphical interface with 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 in Figure 3.3. 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

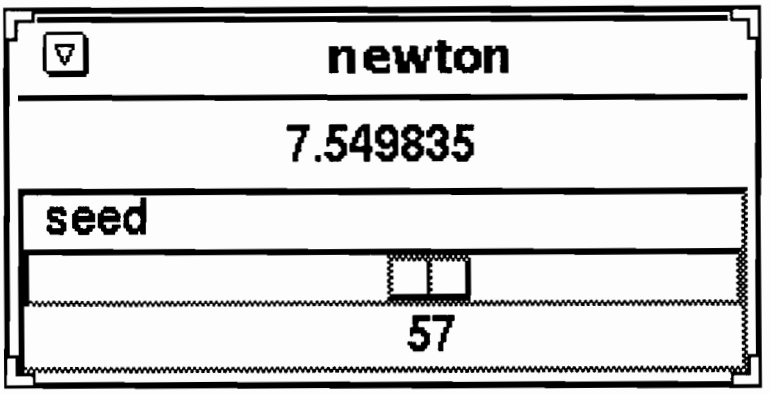

Figure 3.2: The graphical interface for newton

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 echoes the value of the *scale* widget to the *message* widget.

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

```
wish++ -f{
proc generate {} {
    .m configure -text [.s get]
}
scale .s label "seed" -orient horizontal -length 256 -relief sunken
message .m width 256 -aspect 200
pack .m .s -fill x
bind .s <Any-ButtonRelease> {generate}
```

Figure 3.3: The definition of the interface

The C++ code The structure of this program is best explained in four steps. Each

of these steps corresponds with a code fragment. Together, these fragments from the C++ program of our example. We will first look at these steps. Afterwards it will be explained why the individual steps are needed.

```
#include "vsh.h"
double newton(double arg);          // [1]
char* ftoa(double f);               // to convert float to char*
```

Figure 3.4: Step 1: Preliminaries

Step 1: The functional part is represented by the function *newton*. We need to declare its type to satisfy the compiler. The fragment displayed in Figure 3.4, further shows the inclusion of the *vsh.h* header file and the declaration of an auxiliary function *ftoa* that is used to convert floating point values in to a string.

Step 2: The next step, shown in Figure 3.5, involves the definition of the interfacing between the Tcl code and the C++ program.

The class *generater* 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* widgets defined for the interface, C++ pointers to these widgets are stored in instance variables of the object. These pointers are initialized when creating a *generater* object. The widgets are destroyed when deleting the object. Note that the widgets must first be destroyed before deleting the corresponding C++ objects.

All one needs 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 *generater* :: *operator*

function, which is also displayed in Figure 3.5, results in displaying the outcome of the *newton* function applied to the value of the slider, in the message widget.

Step 3: The third step, displayed in Figure 3.6, is to determine an application class, which is needed for the program to initialize the X-windows main event loop. An application class needs to be defined as a subclass of *session*. To initialize the program, the application class redefines the (virtual) function *main* inherited from the *session* class. The function *application* :: *main* takes care of initializing the interface, creates an instance of the *generator* class (see Figure 3.5) to the *generator* object.

Step 4: Finally, as displayed in Figure 3.7, the 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 *vsh* and some of which are due to programming in 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 member functions. When programming a window-based application, at a certain point control is delegated to the window environment. Consequently, there needs to be some kind of main loop and the dispatching of events, in response to which control may be delegated to an appropriate component of the program (Figure 3.7).

To hide the details of activating the main loop and the dispatching of events, the *vsh* library provides a class *session* that allows one to define an application class to initialize the program (Figure 3.6).

In order to respond to events, the *vsh* library provides a *handler* class, that allows one 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 (Figure 3.5).

```
class generator : public handler {          // [2]
public:
generator() {
     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()();
private:
scale* s;
message* m;
}

int generator::operator()() {               // operator()
     float f = s →get();
     m →text( ftoa( newton( f) ) );
     return OK;
}
```

Figure 3.5: Step 2: The *generator* class

Handler classes are typical for *vsh*. Another feature typical for *vsh* 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 *vsh* 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

```
class application : public session {          // [3]
public:
appliaction(int argc, char* argv[]) : session(argc,argv,"newton") {}
void main( kit* tk, int, char* ){
    tk →source("interface.tcl");              // read interface script
    handler* g = new generator();
    tk →action("generate",g);                 // declare action
}
}
```

Figure 3.6: Step 3: The *application* class

```
void main( int argc, char **argv ){          // [4]
    session* s = new application(argc,argv);
    s →run();
}
```

Figure 3.7: Step 4: The function *main*

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 9.5.

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 *vsh* class library

The example given in the previous section showed what kind of components are typically used when developing a program with *vsh* library. However, instead of employing a Tcl script, the window interface may also be developed entirely by employing *vsh* C++ widgets. In this section, a brief overview will be given of the classes offered by the *vsh* library. Further it will be shown how to construct *wish* + + interpreter reffered 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.

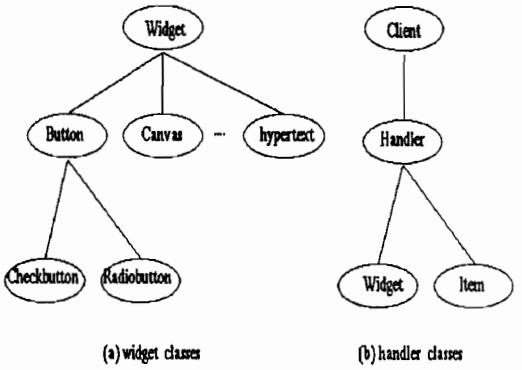

Figure 3.8: An overview of the *vsh* library

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

In the widget class hierarchy depicted in Figure 3.8(a), 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 chapter 5.

The class hierarchy depicted in Figure 3.8(b) depicts the *handler* class as a subclass of client. The reason for this will become clear in chapter 4. 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 chapter 7) 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 chapter 4 and chapter 5.

The wish++ interpreter In the introduction, *vsh* and *wish++* were respectively announced as a C++ library and as an interpreter extending the *wish* interpreter. The program shown in Figure 3.9 presents their relation in a simple way.

The structure of the program is similar to the C++ example of section 3.1. Part[1] consists merely of including the *vsh.h* header file. Part[2] is empty. Part[3] consists of an application class, derived from *session*, defining how the *wish++* interpreter deals with command-line arguments (a and d) and the initialization that takes place when the main event loop is started (c). To understand (a) and (d) it suffices to know that the *vsh* library provides a hypertext widget and that the -x option treats the next argument as the name of a hypertext file. In chapter 9, an example will be given that involves the hypertext widget. In (b) the "vsh.tcl" is declared to be initialization file. It contains the Tcl code for installing the extensions loaded in (c). In (e), a predefined button .quit is packed to the root widget. Part[4] is identical to its counterpart in the previously given example.

The *wish++* interpreter defined by the program extends the *wish* interpreter by

```
#include "vsh.h"                                          // [1]
class application : public session {                      // [3]
public:
application( int argc, char* argv[]) : session(argc,argv){
    hyper = 0;
    if ( (argc == 3) && !strcmp( argv[1],"-x") ) {        // a
        hyper = 0;
        strcpy(hyperfile,argv[2]);
    }
    init(vsh.tcl);                                        // b
}
void main( kit* tk, int argc, char* argv[] ){
    init_expect(tk);init_itcl(tk);init_dp(tk);            // c
    if (hyper) {
        hypertext* h = new hypertext(".help");            // d
        h →file(hyperfile);
        h →geometry(330,250);
        h →pack();
        tk →pack(".quit");                                // e
    }
}
private:
char hyperfile[BUFSIZ];
int hyper;
};
int main (int argc, char* argv[]) {                       // [4]
    session* s = new application(argc,argv);
    s →run();
}
```

Figure 3.9: The *wish* + + interpreter

loading the *Itcl* and *Dp* extensions discussed in the introduction 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 Figure 3.3, for example, may be executed using the *wish++* interpreter.

3.3 The *kit* class

Vsh is meant to provide a parsimonious C++ interface to Tcl/Tk. Nevertheless, as with many a toolkit, some kind of API shock seems to be unavoidable. This is specially true for the *widget* class (treated in chapter 5) and the class *kit* defining the C++ interface with the embedded Tcl interpreter. The functionality of the *kit* can only be understood after reading this article. However, since an instance of *kit* is used in almost any other object (class), it is presented here first. See Figure 3.10. The reader will undoubtingly gradually learn the functionality of *kit* by studying the examples.

To understand why a *kit* class is needed, recall that each *vsh* 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 confirming 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 Tcl command, when this command is defined in C++ (as may be done with *kit::action*). The function *evaluate* provides a shorthand for combining *eval*() and *result*(). The function *source* may be used to read in file containing a Tcl script.

Also, we have the *kit* :: *action* function that may be used to associate a Tcl command with a handler object. In section4.3, alternative ways of defining an action are discussed.

The interpreter *kit*

```
item interface kit:vcl{
    int eval(char* cmd);
    void result(char* s);
    char* evaluate(char* cmd);
    int source(char* f);
    void after(int msecs,char* cmd);
    char* send(char* it, char* cmd);
    char* selection(char* options ="");
    class event event();
    widget* root();
    widget* pack(widget* w,char* options = "{top fillx filly}");
    widget* pack(char* wp,char* options = "{top fillx filly}");
    action& action(char* name);
    action& action(char* name, handler* h);
    action& action(char* name, command f,client* data =0);
    action& action(char* name, tclcommand f,clientdata data = 0);
    void trace(int level =1);
    void notrace();
    void quit();
}
```

Figure 3.10: The *kit* class

The next group of functions is related to widgets and events that may occur to widgets. The function *event* delivers the latest event. It may only be used in command that is bound to some particular. When other event occur before accessing the event object, the information it contains may be obsolete.

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.

Next, we have a group of functions related to X environment. The function *selection* delivers the current X selection. The function *after* may be used set a timer callback for a handler. Setting a time 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 process any pending event. 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* object.

The function *action* may be used to associate a Tcl command to a command function to handler written in C++. See section 4.3 for details.

The function *trace*() and *notrace*() 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 session.

3.4 The *session* class

Each program written with *vsh* may contain only one embedded *wish++* interpreter. To initialize an instance of the *kit* and to start the main (event dispatch) loop an instance of *session* must be created. See Figure 3.11.

Consequently, the class *session* offers functions to initialize and install the functionality needed apart from function to start the main loop. The preffered way of doing this 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 Figure 3.9. When creating a *session* object, the *name* of the application may be

given as the last parameter. Under this name, the application is known to other applications, that may communicate with each other by means of *send* command.

The function *init* may be used to specify a different initialization script. This script must include the default *wish* + + initialization script, which is an adapted version of the original *wish* initialization script.

The function *install* takes a function as a parameter. The function will be called after the initialization script has been evaluated, but before evaluating a script provided by the user or executing the program function specified when calling *run*

The *session* class

```
interface session{
    session(int argc, char** argv, char* name = 0);
    void init(char* fname);
    void install(void proc(kit*, int, char**));
    int run(void proc(kit*, int, char** ) = 0);
}
```

Figure 3.11: The *session* class

Finally, the function *run* is called to start the main loop. The parameter of *run* specifies the program to be executed. It may be zero in case the program will only be used to execute scripts. To execute both a script and the program function specified for *run*, the script must contain the command *goback* as its last command. In either case, the main loop must be started by calling *run*.

Chapter 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, etcectra. Instead of explicitly dealing with all incoming events, the application delegates control to the environment by associating a callback function with each event that is relevant to 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 a member functions of a single class. 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.

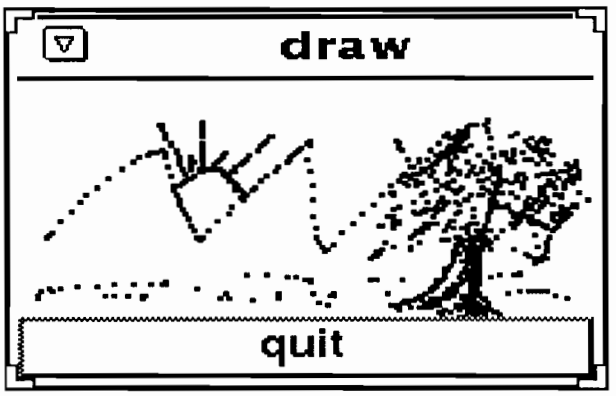

Figure 4.1: Painting a canvas

A simple drawing editor Before looking at the program, think of what one would like a drawing editor to offer him. And, if one has any experience in programming graphics applications, how would he 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. See Figure 4.1.

The program realizing our first attempt is depicted in Figure 4.2. Again, the program may be broken up in four components.

Component [1] consists of simply including the *vsh.h* header file.

Component [2] defines 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 7.2 for further details on the *canvas* class.

Now 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

```
#include "vsh.h"                                          // [1]
class drawing_canvas : public canvas {                    // [2]
public:
drawing_canvas( char* path) : canvas( path) {             // (a)
    geometry(200,100);
    handler(this);
    dragging = 0;
}
void press( event&) { dragging = 1 }
void motion( event& e) {                                  // (b)
    if (dragging) circle(e.x(),e.y(),1,"-fill black");
}
void release( event&) { dragging = 0 }
protected:
int dragging;
};
class application : public session {                      // [3]
public:
application( int argc, char* argv[] ) : session( argc,argv,"draw"){}
void main( kit* tk, int, char*){
    canvas* c = new drawing_canvas(".draw");
    c →pack();
    tk →pack(".quit");
}
};
int main( int argc, char* argv[]){                        // [4]
    session* s = new application(argc,argv);
    return s →run();
}
```

Figure 4.2: A simple drawing tool

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 *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 (a), 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 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* and results in making the widget sensitive to a member of predefined events, that may be different for each concrete widget class.

Discussion A note on terminology is in place here. The reader may be a bit astounded by the fact that we have a both *handler* class and a *handler* function, which is more properly written as *widget* :: *handler*. 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 handler(class handler* h) { ... }

...
};
```

Note that there is no ambiguity here. A *handler* object is an object that may be invoked in response to a Tcl command or an event. The *handler* 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 occurance of the otherwise mysterious expression *handler*(*this*). The reason not to identify a widget with a handler is simply that some widgets need separate handlers. Another reason is to avoid the proliferation of the class name space, which would inevitably occur when forcing the programmer to define the response to events by defining a descendant class of the particular widget class.

Before studying the abstract *handler* class in more detail, we will briefly look at the definition of the *event* class. Note that in (b), the event reference is only used to inform after the position of the mouse pointer.

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 event type. When such a binding exists for a widget and the (toolkit) environment decides that the event belongs to the widget, then the callback associated with the event is executed. Information concerning the event may be retrieved by asking the kit for the latest event.

Event objects represent the events generated by the X-window system. Each event has a type, which may be one of the types listed below. The type of the event can be inspected with *type*() which return an integer value or *name*() which returns a string representation of the type. For some of the common events 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*(), it is checked 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.

Program not satisfied with the interface can check the type and access the underlying Xevent at their own risk.

Event

```
interface event {
    int type();
    char* name();
    int x();
    int y();
    int x_root();
    int y_root();
    int button(int i = 0);
    int buttonup(int i = 0);
    int motion();
    int keyevent();
    int buttonevent(int i = 0);
    int keycode();
    void trace();
    class widget* widget();
    XEvent* xevent();
}
```

Figure 4.3: The *event* class

In addition to ordinary event information, an event also contains a reference to widget for which the event occurred. This information is valid only when the event is the last event. It may be lost when other event occur.

4.2 Handlers

Handler objects provide a type-secure way to deal with client data. Client data are often needed to share some common resource or to update a global structure. Explicit coercion, however, are usually error-prone. Handler objects may only be created as instances of classes derived from the class *handler*. The (client) information that needs to be passed around when using plain *command* function, can conveniently be stored in the instance variables of the (derived) handler class. A *handler* object offers a special member function *dispatch* that is called directly to execute the action associated with an event or Tcl command. In this way, explicit coercion are avoided.

Handler

```
interface handler:client {
    virtual int dispatch(kit* _tk, int _argc, char** _argv);
    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:
    int argc;
    char** argv;
    kit* tk;
};
```

Figure 4.4: The *handler* class

The class *handler* defines a number of other member functions, corresponding to events type related common user actions. A class derived from *handler* may redefine

these functions and rely on the original *dispatch* function to call the proper member in response to an event.

Dispatching the most important (member) function of a handler object is the *dispatch* function. The *dispatch* function is called when an action is invoked either to execute a Tcl script command or as a callback in response to an event.

The original *handler* :: *dispatch*, shown in Figure 4.4, stores the *kit*, *argc* and *argv* parameters in the corresponding instance variables of the handler object and calls the member function dependent on the type of the event. (See section 9.1 for an example.)

Dispatching

```
inline
int handler::dispatch(kit * _tk, int _argc, char** _argv) {
    tk = _tk; argc = _argc; argv = _argv;
    return this →operator()();
}
int handler::operator()() {
    event e = tk →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() == EntryNotify) enter(e);
    else if (e.type() == LeaveNotify) leave(e);
    else other(e);
    return OK;
}
```

Figure 4.5: The *dispatch* and *operator*() function

The handler class knows only virtual functions. Each function, including the *dispatch* function may be redefined, according to the programmers need.

Example Handler classes may also be conveniently used for actions that do not involve (window) events by redefining the *dispatch* function. The event related member functions are then simply ignored.

Example

```
class application:public handler{
    int n;
    public:
    application() { n = 0; }
    int dispatch(kit* tk, int argc, char** argv);
    void red() {cout << "red" << n++;}
};

int application::dispatch(kit* tk, int argc, char** argv){
    argv++;
    while(argc- > 1) {
        if (strcmp("red",*argv) == 0) a →red();
        else cout ¡¡ "no such option" ¡¡ endl;
        argv++;
    }
    return OK;
}

void program(kit* tk, int, char**){
    application app;
    tk →action("do-something", f, &app);
    tk →eval("do-something red green blue");
    tk →quit();
};
```

Figure 4.6: An Example

In this example a Tcl command is defined, not involving widgets or bindings. The example is very similar to the example given for action. The action is declared by means of the *kit* :: *action* definition.

4.3 Actions

The procedural interface with Tcl is handled by so-called *actions*, defined by a Tcl command, a C/C++ *command* function or a *handler* object. The most common use of actions is to execute a *command* in response to an *event*. But for the more experienced programmer, *actions* provide a powerful means to define Tcl script commands as well.

Client

```
class client { };
```

Command

```
typedef command(client*, kit*, int, char**);
typedef tclcommand(clientdata, tclinterp, int, char**);
```

Figure 4.7: The *client* class

Data passed to *command* function must be of type *client*, which is defined by an empty class introduced only to please the compiler. Below the type definition of *command* is given. Apart from the *client** parameter, a command function must also declare a kit* parameter and an *argc* and *argv* parameter, similar as for *main*. The *client** data of a command (and similar for the *clientdata* parameter of a *tclcommand*) can be any kind of class. However, it is to be preffered that such classes are made subclass of *client*. The *client* data pointer is declared when creating an action and passed to the command function when the actual call is made. The parameters (*argc* and *argv*) depend on the actual call. The use of *argc* and *argv* comes from the original C interface of Tcl. It proves to be a very flexible way of communicating data, especially in string-oriented applications.

The class *action* offers no less than seven constructors. The first constructor, which takes a (*char**) string as a parameter, is merely for convenience. It may be

Action

```
interface action {
    action(char* name);
    action(char* name, handler* h);
    action(char* name, command f, client* data = 0);
    action(char* name, tclcommand f, clientdata data = 0);
    action(handler* h);
    action(command f, client* data = 0);
    action(tclcommand f, clientdata data = 0);
    char* name();
}
```

Figure 4.8: The *action* class

used to convert the name of a Tcl script command into an action. The following three constructors differ from the last three constructors only by their first string parameter which serves to define the name under which the action will be known by the Tcl interpreter. The last three constructors, in contrast, creates anonymous actions, of which the user, however, can ask its name by invoking the function *name*.

The preffered form of creating an action is giving it (apart form a name) a handler as parameter. Handler objects are discussed in the next session. They offer a type-secure way of dealing with client information. In contrast the second constructor of this group, which takes a command function and possibly a pointer to *client* data as parameters, may make (type-insecure) conversions of the client data necessary.

The constructor taking a *tclcommand* and *clientdata* as parameter is incorporated for compatibility reasons only and will further be discussed.

When using any of the last six constructors, as a side-effect an association is created between the *name* of action and Tcl command. If such a Tcl command already exists, the previous association will be overwritten. This is also the case if it has been defined as a Tcl script command.

Example In Figure 4.9, the declaration and use of an *action* is shown as a simple example. The *command* function f communicates with the (*client*) *application* object by checking *argc* and *argv* parameters of the call to f. Dependent on the value of the (argv) string argument(s), an appropriate member function of the *application* object is invoked.

Example

```
class application:public client{
    int n;
    public:
    application() { n = 0; }
    void red() {cout << "red" << n++;}
};

int f(client* c, kit* tk, int argc, char** argv){
    application* a = (application* )c;
    argv++;
    while(argc- > 1) {
        if (strcmp("red",*argv) == 0) a →red();
        else cout << "no such option" << endl;
        argv++;
    }
    return OK;
}

void program(kit* tk, int, char**){
    application app;
    tk →action("do-something", f, &app);
    tk →eval("do-something red green blue");
    tk →quit();
};
```

Figure 4.9: An example

The function f is invoked by calling the Tcl scripts command $do-something$ via

kit :: *eval*. However, since *do – something* is not a built-in Tcl or Tk command, it must first be declared as a command by means of the *kit* :: *action* function. When defining the action, the function *f* is declared to be the command function, with (the address of) the application object *app* as the client parameter. To use the (*Client*) data in *f* as an application object, data must be coerced to (application*). A better, that is type-secure, way to deal with this is to define a handler class.

Chapter 5

User Interface widgets

The Tk toolkit offers numerous built-in widgets. The Tk widget confirm to the look-and-feel of the OSF/Motif standard. *vsh* the 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 widget, the *vsh* library integrates a number of other widgets, such as *barchart*, *hypertext*, and *photo* widget (created by other Tk adapts). Also some others widget are offered, such as *filechooser* and MPEG video widget.

The widget classes are organized as a tree, with the class *widget* at the root. See Figure 5.1. Each concrete widget class offers the functionality supported by the (abstract) *widget* class and many in addition define functions specific to the particular widget class. The member functions for a widget class have usually a straightforward correspondence with the command interface defined by the tcl/tk toolkit.

Each function listed in the class interface is *public* unless it is explicitly indicated as *protected*. The interface descriptions start with 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.

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

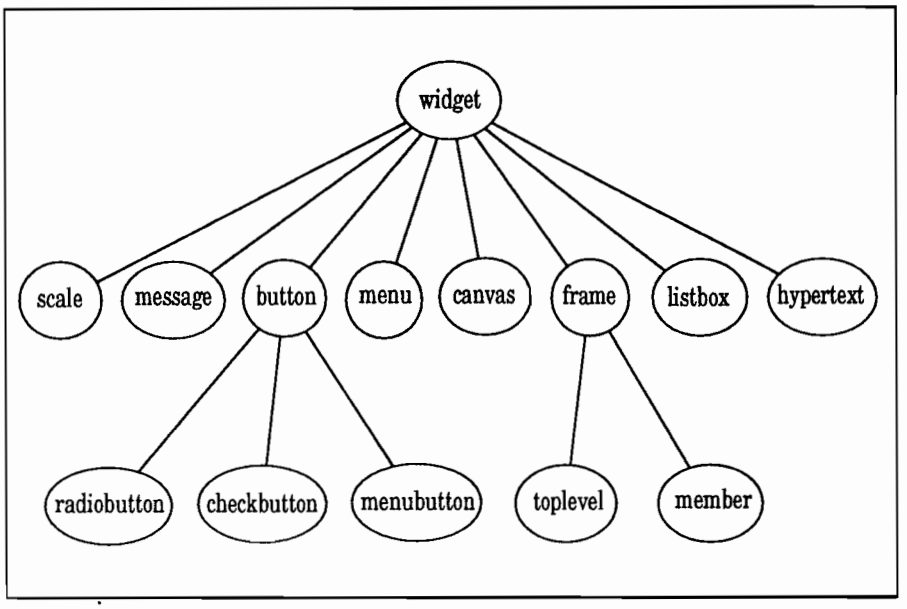

Figure 5.1: *vsh* Widget Class Hierarchy

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.

In this section all of the widgets available with *vsh* library will be described. In section 9.5 it is explained how to create new widgets in C++ and make them available for use in a Tcl script.

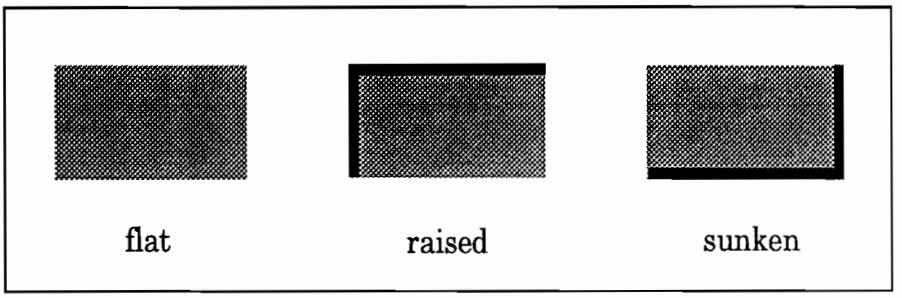

Figure 5.2: The Frames and Toplevel widgets

5.1 Frames and toplevels

Frames and toplevels are the simplest widgets. They have almost no interesting properties. A frame appears as a rectangular region with a color and possibly a border that gives the frame a raised or sunken appearance as shown in Figure 5.2. Frame serves two purposes. First, they can be used to generate decorations such as block of color or a raised or sunken border around a group of widgets. Second they serve as containers for grouping other widgets; most of the non-leaf widgets in the widget hierarchy are frames.

Toplevel widgets are identical to frames except that, as the name implies, they are top-level widgets whereas frames (and almost all other widgets) are internal widgets. This means that a toplevel widget can be positioned anywhere on its screen, independent of its parent in the widget hierarchy, and need not even appear on the same screen as its parent. Toplevels are typically used as the outermost containers for panels and dialog boxes. When you create a toplevel you can specify a screen for it to be displayed on. The class structure for frame is shown in Figure 5.3.

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

Figure 5.3: The *frame* class

5.2 Buttons

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 subclass of the class *button*.

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 defines, albeit protected.)

When a mouse cursor moves over a button, the button lights up. This indicates that pressing a mouse button will cause something to happen. It is a general property of Tk widgets that they light up if the mouse cursor passes over them when they are prepared to respond to button presses. A button or other widget lit up in this way

```
interface button : widget {
    button(char* p, char* options = "");
    button(widget* w, char* p, char* options = "");
    void text(char* s);
    void bitmap(char* s);
    void state(char *s);
    void flash();
    char* invoke();
    protected :
    void install(action& ac, char* args = "");
}
```

Figure 5.4: The *button* class

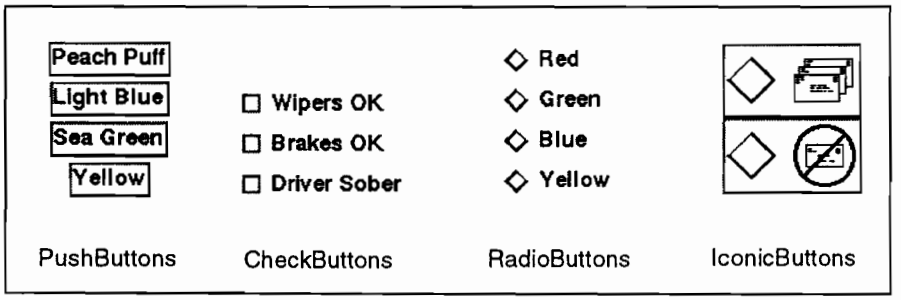

Figure 5.5: Members of the button family of widgets

it is said to be *active*. Buttons become inactive again when the mouse cursor leaves them.

5.2.1 Checkbuttons

Checkbutton allows users to make binary choices such as enabling or disabling underlining or grid-alignment. They are similar to regular button except for two things. First, whenever mouse button 1 is clicked over a checkbutton a Tcl variable toggles between two values representing an "on" state and other representing "off" state. The name of the variables and the values corresponding to "on" and "off" states are configuration options for the widget. Second, the checkbutton displays a small rectangular *selector* to the left of its text or bitmap. If the variable has the "on" value then the selector is displayed in a bright color and the button is said to be *selected.* If the variable has the "off" value then selector box appears empty. Each checkbutton monitors the value of of its associated variable and if the variable's value changes (e.g because of set command) the checkbutton updates the selector display.

5.2.2 Radiobuttons

The last member of the button family is the radiobutton class. Radiobuttons are typically arranged in groups and used to select one from among several mutually-exclusive choices, such as one of several colors or one of several styles of dashed lines. Radiobuttons are named after the radio selector buttons on older cars, where pressing the button for one station caused all the other buttons to be released. When mouse button 1 is clicked over a radiobutton, the widget sets the variable to the "on" value associated with the radiobutton. All of the radiobuttons in a group will share the same variable but each will have a different "on" value.

5.2.3 Labels

Labels are the simplest member of the family. One can use labels to display a text string or a bitmap (see Figure 5.5). Like normal buttons, labels do not normally respond to the mouse or keyboard; they are simply to provide decoration in the form of a text string or bitmap.

```
interface menubutton : button {
menubutton(char* p, char* options = "");
menubutton(widget* w,char* p, char* options = "");
void menu(char* s);
void menu(class menu* m);
}
```

Figure 5.6: The *menubutton* class

5.2.4 Menubuttons

The menubutton is a specialization of the *button* widgets. It allows for attaching a menu that will be displayed when pressing the button. The menubutton Figure 5.6 must be used to pack menus in a *menubar*.

5.3 Menus

Another, frequently occurring, widget is the *menu* widget. 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.

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 an *action* to be associated with the entry. Alternatively, if no action is specified, the default handler action installed by invoking *widget* :: *handler* will be used. The string used as a label for the entries (the first parameter of *entry*) will be

```
interface menu : public widget {
    menu(char* p, char* options = "");
    menu(widget* w, char* p, char* options = "");
    menu* add(char* s, char* options = "");
    menu* entry(char* s, action& ac, char* args ="", char* options = "");
    menu* entry(char* s, char* args ="", char* options = "")A);
    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();
    void del(int i);
    void del(char* s);
    char* invoke(int i);
    char* invoke(char* s );
    void post(int x = 500, int y = 500);
    void unpost();
    protected :
    char* name();
    void install(action& ac, char* args);
}
```

Figure 5.7: The *menu* class

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 descendant 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 enquire which entry has been selected. See the example below.

Entries may be deleted using the function *del* and invoked by using *invoke*. For

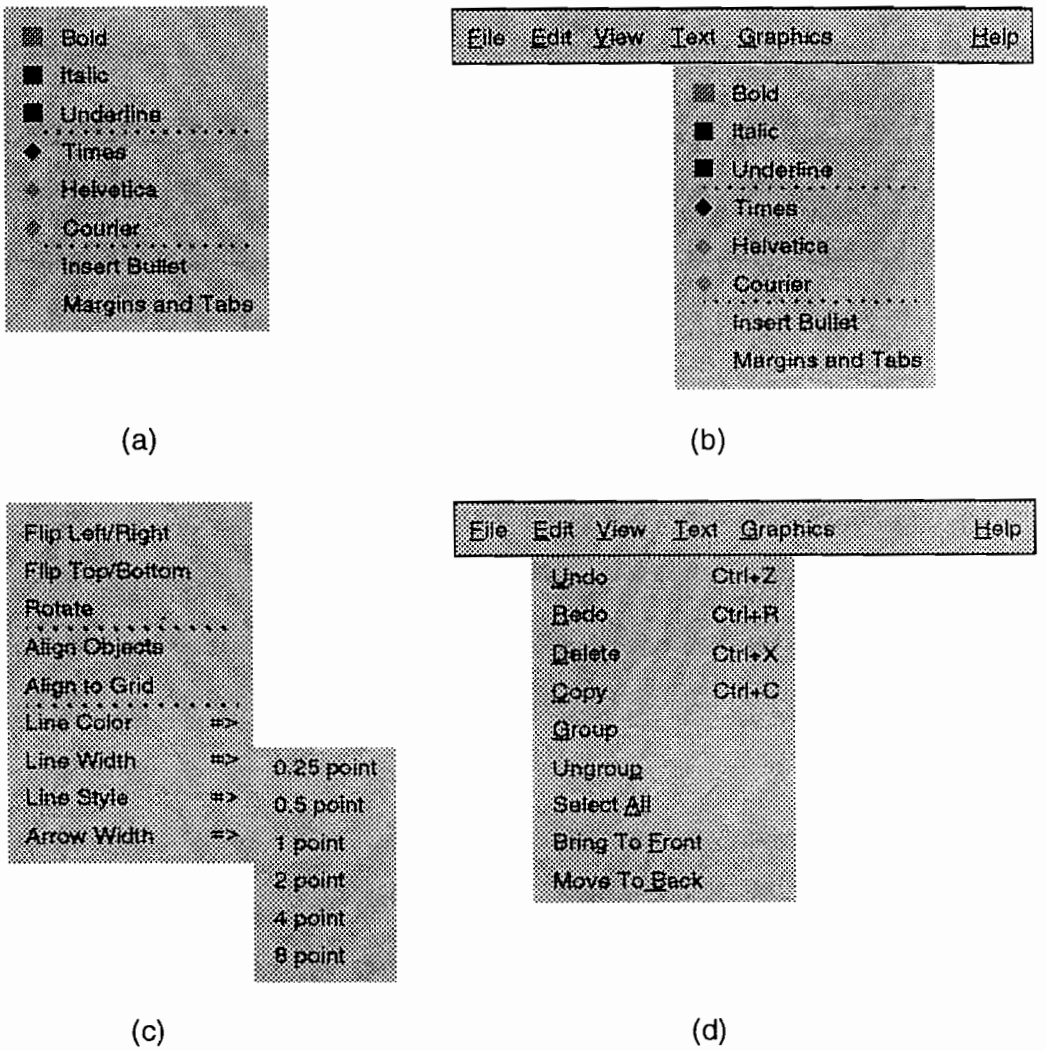

Figure 5.8: Examples of menus

both functions, the entry must 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 embedding the menu in a *menubutton*.

Unlike most other widgets, menus do not normally appear on the screen. They

spend their time in an invisible state called *unposted.* When a user wants to invoke a menu entry, he or she *posts* the menu, which makes it appear on the screen. Then the user moves the mouse over the desired entry and releases button 1 to invoke that entry. Once the menu has been invoked it is usually unposted until it is needed again. Menus are posted or unposted by invoking their widget commands, which gives the interface designer a lot of flexibility in deciding when to post and unpost them. The subsection below describe four of the most common approaches.

5.3.1 Pull-down menus

Menus are most commonly used in a $pull - down$ style. In this style the application displays a *menubar* near the top of its main window. A menu bar is a frame widget that contains several menubuttons widgets as shown in Figure 5.8(b). Menubuttons are similar to button widgets except that instead of executing handler associated with it they post menu widgets. When a user presses presses mouse button 1 over a menubutton it posts its associated menu underneath the menubutton widget. Then the user can slide the mouse down over the menu with the button still down and release the mouse button over the desired entry. When the mouse button is released the menu entry is invoked and the menu is unposted.

5.3.2 Pop-up menus

The second common style of menu usage is called $pop - up$ menus. In this approach, pressing one of the mouse buttons in a particular widget causes a menu to post next to the mouse cursor and the user can slide the mouse over the desired entry and release it there to invoke the entry and unpost the menu. As with pull-down menus, releasing the mouse button outside the menu causes it to unpost without invoking any of its entries.

5.3.3 Cascaded menus

The third commonly used approach to posting menus is called *cascaded* menus. Cascaded menus are implemented using cascade menu entries in other menus, such as pull-down and pop-up menus. Each cascade menu entry is similar to a menubutton in that it is associated with a menu widget. When the mouse cursor passes over the cascaded entry, its associated menu is posted just to the right of the cascaded entry, as shown in Figure 5.8. The user can then slide the mouse to the right onto the cascaded menu and select an entry in the cascaded menu. Menus can be cascaded to any depth.

5.4 The *scale* class

The scale widget may be used to obtain numerical input from the user. When a handler is attached to *scale* it is called when the user releases the slider. The value of the *scale* is passed as an additional parameter when invoking the handler. The default binding for the scale is the *ButtonRelease* event. The class structure for scale is shown in Figure 5.9.

5.5 The *message* class

The message widget may be used to display a message on the screen. The message class shown in Figure 5.10 does not define default bindings, but the user is free to associate events to a message widget by employing *widget* :: *bind*.

5.6 Listboxes

A listbox is a widget that allows the user to select one or more possibilities from a range of alternatives, such as a file name from those in the current directory or a

```
interface scale : public 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:
    void install(action& ac, char* args = "");
};
```

Figure 5.9: The *scale* class

```
interface message : public widget {
public:
char* type();
message(char* p, char* options = "");
message(widget* w, char* p, char* options = "") : widget(w,p);
void text(char* s);
};
```

Figure 5.10: The *message* class

color from a database of defined colors. A listbox contains one or more entries, each of which displays a one-line string. The widget commands for listboxes allow entries to be created,destroyed and queried. If there are more entries than there are lines in the listbox's window then only a few of them are displayed at a time;the user can control which portion is displayed by using a separate scrollbar widget associated with the listbox. The view in a listbox can also be controlled by pressing mouse button 2 in the widget and dragging up or down. This is called scanning:it has the effect of dragging the listbox contents past the window at high speed. Most Tk widgets that support scrollbars also support scanning. If the string in the listbox are too long to fit in the window then the listbox can also be scrolled and scanned in the horizontal direction. Typically listboxes are configured so that the user can select an entry by clicking on it with mouse button 1. In some cases the user can also select a range of entries by pressing and dragging with button 1. Selected Entries appear in a different color and usually have a raised 3-D effect. Once the desired entries have been selected, the user will typically use those entries by invoking another widget, such as a button widget or menu entry. For example, the user might select one or more file names from a listbox and then click on a button widget to delete the selected files;the TCL command associated with the button widget can read out the strings from the selected listbox entries. It's also common for listboxes to support double-clicking,which both selects an entry and invokes some operations on it. For example, in a file-open dialog box, double-clicking on a file name might cause that file to be opened by the application.

5.7 Entry

An entry is a widget that allows the user to type in and edit a one-line string. For example, if a document is being saved to disk for the first time then the user will have to provide a file name to use. The user might type the file name in an entry widget, then click on a bottom widget whose Tcl command retrieves the file name from the entry and saves the document in that file.

```
interface entry:widget {
    entry(char* p, char* options = "");
    entry(widget* w,char* p, char* options = "");
    void insert(char* s);
    char* get();
}
```

Figure 5.11: The *entry* class

To enter text into an entry the user clicks mouse button 1 in the entry. This makes a blinking vertical bar appear, called the insertion cursor. The user can then type characters and they will be inserted into the entry at the point of the insertion cursor. The insertion cursor can be moved by clicking anywhere in the entry's text. Text in an entry can be selected by pressing and dragging with the mouse button 1, and it can be selected by pressing and dragging with mouse button 1, and it can be edited with a variety of keyboard actions; see the reference documentation for details.

If the test for an entry is too long to fit in its window then only a portion of it is displayed and the view can be adjusted using an associated scrollbar widget or by scanning with the mouse button 2. Entries can be disabled so that no insertion cursor will appear and the text in the entry cannot be modified. The text in an entry can be associated with a Tcl variable so that changes to the variable are reflected in the entry and changes made in the entry are reflected in the variable.

The *entry* class shown in Figure 5.11 offers, in addition to the two (standard) constructors, the functions *insert* and *get*, that are used respectively to set and get the text that will appear in the entry. The text appearing in the entry may be edited by the user.

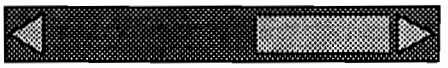

Figure 5.12: Scrollbar

5.8 Scrollbar

Scrollbar widgets are used to control what is displayed in other widgets. Each scrollbar is associated with some other widget such as a listbox or entry. The scrollbar is typically displayed next to the other widget and when the user clicks and drags on the scrollbar the view in the associated widget will change. A scrollbar appears as shown in Figure 5.12 with an arrow at each end and a slider in the middle. The size and position of the slider correspond to the portion of the associated widget's document that is currently visible in its Window. Foe example, if the slider covers the rightmost 20% of the region between the two arrows as in Figure 5.12 it means that the rightmost 20% of the document is visible in the window. Scrollbars can be oriented either vertically or horizontally.

Users can adjust the view by clicking mouse button 1 on the arrows, which moves the view a small amount in the direction of the arrow, or by clicking in the empty space on either side of the slider, which moves the view by one screenful in that direction. The view can also be changed by pressing on the slider and dragging it.

A scrollbar interacts with its associated widget using Tcl scripts. One of a scrollbar's configuration options is a Tcl script to invoke to change the view;typically this script invokes the widget command fir the associated widget. When the user manipulates the scrollbar, the scrollbar invokes the script, including additional information about the new view that the user requested. The associated widget changes its view and then invokes another Tcl script(one of its configuration options) that tells the scrollbar exactly what information is now displayed in the window, so the scrollbar can display the slider correctly. The scrollbar doesn't update its slider until told to do so by the associated widget; this makes it possible for the associated widget to

reject or modify the user's request(e.g. to prevent the user from scrolling past the ends of the information in the widget).

Chapter 6

Configure widgets

Widgets are the element 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 a drawing application containing 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 *path name*. The path name of a widget reflects its possible subordination to another widget. A pathnames consists of strings separated by dots. The first character of a path must be a dot.The format of a path name may be expressed in BNF form as

$$< path >::=' .'|'.' < string > \mid < path >' .' < string >$$

For example "." is the pathname of the root widget, whereas ".quit" is the pathname of a widget subordinate to the root widget. A widget subordinate to another widget must have the pathname of that widget as part of its own pathname. For example, the widget ".f.m" may have a widget ".f.m.h" as a subordinate widget. Note that the widget hierarchy depicted in Figure 3.8(a) and Figure 6.1. With respect to

the pathname hierarchy, when speaking of ancestors we simply mean superordinates widgets.

Pathnames are treated somewhat more liberally in *vsh*. For example, widget pathnames may simply be defined or extended by a string. The missing dot is then automatically inserted.

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 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 Tcl command like

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

The function *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. The widget class interface offers two *pack* functions. The function *widget::pack(char*)* applies to individual widgets. As options may specify for example `-side X`, where X is either top, bottom, left or right, to pack the widget to appropriate side of the cavity specified by the ancestor widget. Other options are `-fill x` of `-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.

Alternatively, the function *widget::pack(widget*, char*)* may be used, which allows for the same options but applies packing to the widget parameter. This function is convenient when packing widgets in a *frame* or *toplevel* widget.

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

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

The function *bind* is used to associate actions 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 the *canvas* widget. Also keystrokes may be defined as events, as for example Return, which is the default event for the *entry* widget.

The function *widget* :: *handler* may be used to associate a handler object or action with the default binding 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.

For both the *bind* and *handler* 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 its handler.

6.1 Compound widgets

In addition, the widget class offers four functions that may be used when defining compound or mega widgets. The function *redirect*(w) must be used to delegate the invocation to the widget to which the commands are redirected. After invoking *redirect*, the function *thepath* will deliver the path that is determined by *self()*

Widget

```
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();                      // combines eval and result()
    virtual void configure(char* cmd);     // invokes 'path() configure cmd'
    virtual void geometry(int xs, int ys); // determines width x height y
    widget* pack(char* options = "top");
    widget* pack(widget* w, char* options = "top");
    virtual bind(char* b, handler* h, char* args = "");
    virtual bind(char* b, action& ac, char* args = "" );
    void handler(class handler* h, char* args = "" );
    void handler(action& ac, char* args = "" );
    void xscroll(scrollbar* s);            // to attach scrollbars
    void yscroll(scrollbar* s);
    void focus(char* options = "");
    void grab(char* options = "");
    void destroy();                        // removes widget from the screen
    Tk_Window tkwin();                     // gives access to Tk_Window
    widget* self();                        // for constructing mega widgets
    void redirect(widget* w);

protected;
    char* thepath();                       // delivers the virtual path
    virtual install(action&, char* args=""); // default binding
}
```

Figure 6.1: The *widget* class

Chapter 7

Graphics and Hypertext

The Tk toolkit offers powerful facilities and (hyper)text [Con87, Ous93]. In this section we will discuss only the *canvas* widget offered by Tk. And instead of looking at *text* widget provided by Tk, we will (briefly) look at the *hypertext* widget, which presents an alternative approach to defining hyperstructure.

7.1 The *item* class

The canvas widget allows the programmer to create a number of built-in graphics 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 *vsh* library contains a class *item* of which the functionality is shown below.

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 any 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, item may be associated with events, either explicitly by using *item* :: *bind*, or implicitly by using *item* :: *handler*. The default

```
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);
    virtual bind(char* b, handler* h, char* args = "");
    virtual bind(char* b, action& ac, char* args = "" );
    void handler(class handler* h, char* args = "" );
    void handler(action& ac, char* args = "" );
protected;
    virtual install(action&, char* args=""); // default binding
};
```

Figure 7.1: The *item* class

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 defining their own handler.

7.2 The *canvas* widget

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

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

Currently, the graphic items *bitmap*, *line*, *oval*, *polygon* and *rectangle* may be created and, in addition to,*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 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*.

```
interface canvas : widget {
    canvas(char* p, char* options="");
    canvas(widget* w, char* p, char* options="");
    void tag(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 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 current();
    item overlapping(int x, int y);
    itemconfigure(int it, char* options);
    itemconfigure(char* tag, char* options);
    itembind(int it, char* s, action& a, char* args = "");
    itembind(char* tag, char*s, action& a, char* args = "");
    void postscript(char* file, char* options="");
};
```

Figure 7.2: The *canvas* class

7.3 The *Hypertext* widget

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

A different, in a way more static, approach is offered by the hypertext widget developed by george.howlett@att.com. The *vsh* class interface to the hypertext widget is given below.

```
interface hypertext : widget {
    hypertext(char* p, char* options = "");
    hypertext(widget* w, char* p, char* options = "");
    void file(char* f); to read in hypertext file
}
```

Figure 7.3: The *hypertext* class

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

Widgets created when reading in hypertext file may be given a pathname relative to 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 filename.

The *hypertext* widgets may be used to display textfiles 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.

Any of the widgets and commands offered by Tcl/Tk or supported by *vsh* may be included in a hypertext file. However, this is for more advanced programmers only since it requires intimate knowledge of the Tcl/Tk intrinsic.

Chapter 8

Employing the scripting language

When developing complex user interfaces, it will often be advantageous to do most of the work in the scripting languages and to restrict the interaction with the program written in C++ to a bare minimum. However, occasionally, the (C++) program will need to access or modify variables defined in the Tcl script. To this end, the *vsh* library offers the class *var* Figure 8.1.

An instance of *var* may be created as a reference to an already existing variable or by giving the name of the variable. Creating such a variable in C++ has no side-effects. It does not create or modify Tcl variable, it merely modifies access to it(if it exists). A variable may be assigned a value of type *char*, *int*, *float* or the value of another instance of *var*. In the future, other value types may be supported as well.

Variables may be associated with an *action* that is triggered whenever one of the operations *read*, *write* or *unset* as defined by the options is performed on the variable. For more details see the Tcl *trace* command.

The name and (string) value of the variable may be retrieved by respectively the function *name* and the application operator. In addition for each value type supported a type conversion operator is supplied. (No type checking is supported yet.)

```
interface var:client{
    var(var& v);
    var(char* s);
    char* operator = (char* s);
    char* operator = (int i);
    char* operator = (float f);
    void trigger(action& ac, char* options = "rwu", char* args = "");
    char* name();
    char* operator();
    operator char* ();
    operator int ();
    operator float ();
}
```

Figure 8.1: The *var* class

```
int inc(client* data, kit* tk, int argc, char** argv){
    var x = * (var*) data;
    cout << x.name() << " = " << x() << endl;
    x = (int) x + 1;
    return OK;
}
```

Figure 8.2: The *script* example

Example The function *inc*, shown in Figure 8.2, assumes to have a variable (with value type *int*) as a client. It prints the name and value of the variable and then performs an increment.

The function *install* shown in Figure 8.3 declares a (Tcl) variable X and another (Tcl) variable (Y). Further, it associates the function *inc* (with as client the variable

```
void install(kit* tk, int, char** ){
    var x("X");
    var* v = new var("Y");
    action ac(browse,v);
    x.trigger(ac,"w");
    x = 0; (* v) = 0;
}
```

Figure 8.3: Script example: The *install* function

```
void program(kit* tk, int, char** ){
    var* v = new var("X");
    button* b = new button(".b");
    b →text("X++");
    b →handler(browse,v);
    tk →pack(".quit");
```

Figure 8.4: Script example: The *program* function

Y) with variable X. When X is assigned a value, due to the association, the function *inc* will be invoked (for Y). Note that the variable for Y is created dynamically, otherwise the (client) pointer to Y would be dangling when invoking *inc*. In contrast, the variable object created for X may be discarded, since it is merely used to provide temporary access to the (Tcl) variable X.

The function *program* (shown in Figure 8.4 also declares a (separate) *var* object to give access to (the same) variable X and uses it as the client for an action with X, incrementing X will result in subsequently incrementing Y.

No doubt, the reader will need to experiment to get a feel for the interaction

```
int main (int argc, char** argv){
    session* s = new session(argc,argv,"root");
    s →install(install);
    return s →run(program);
}
```

Figure 8.5: Script example: The *main* function

between C++ functions and Tcl scripts.

To complete the example, the function *main* using *install* and *program* is shown in Figure 8.5.

Chapter 9

Model Interaction

In this section we will look at an extension of the simple drawing tool presented in chapter 4.

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

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 component themselves, but to delegate the configuration and binding instructions to the appropriate components. In section 9.5 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 Tcl script. Ideally, defining a new widget includes both the definition of a C++ class interface and the definition of the corresponding Tcl command.

9.1 The *drawtool*

Our drawing tool consists of a *tablet*, which is a 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. In addition, a help facility is offered. See Figure 9.1.

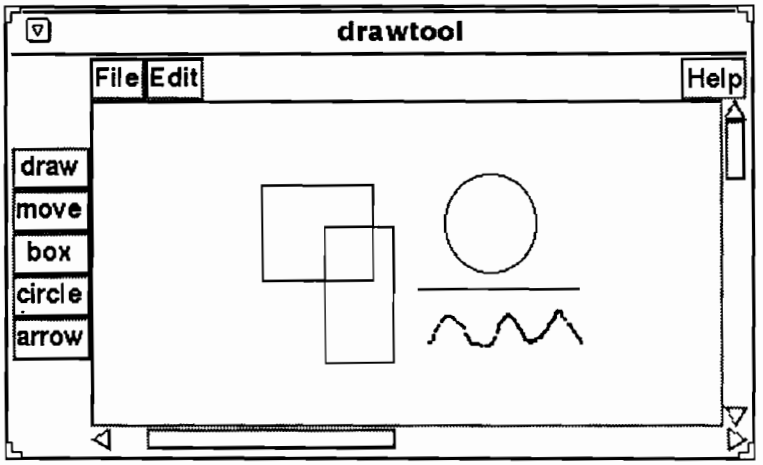

Figure 9.1: The *drawtool* interface

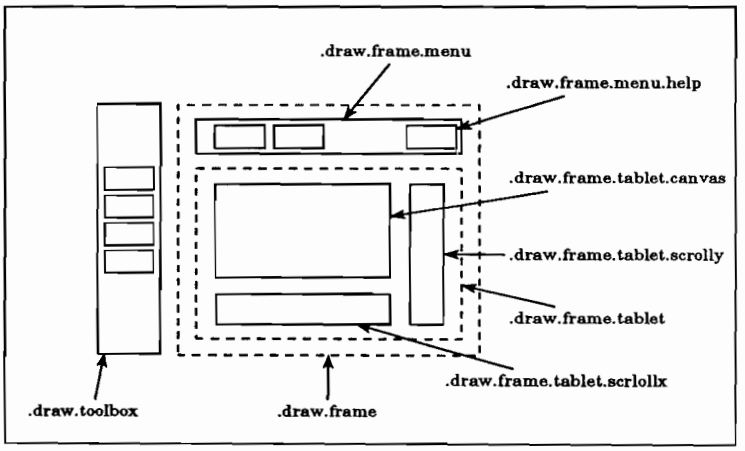

Figure 9.2: The *drawtool* widget hierarchy

In Figure 9.3 the application class for the *drawtool* is depicted. Before the main event loop is started, the components of the drawing tool are created and packed to

the root widget. The widget hierarchy is shown in Figure 9.2.

```
class application : public session {              // drawtool
public:
application(int argc, char* argv[]) : session(argc,argv,"drawtool") {}
void main( kit* tk, int argc, char* argv[] ) {
    frame* root = new frame( path() );        // (a)
    frame* f = new frame(root,".frame");
    tablet* c = new tablet(f,options);
    redirect(c);                              /// (b)
    toolbox* b = new toolbox(root,c);
    menubar* m = new menu_bar(f,c,b);
    tk →pack(m) →pack(c) →pack(b,"-side left") →pack(f,"-side right");
}
}
```

Figure 9.3: The drawing tool

In addition to the *tablet*, *menu_bar* and *toolbox*, a *frame* widget is created to pack the menubar 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 *toolbox* is given to the *menu_bar* in sections 9.2 and 9.3, respectively. In the example, no attention will be paid to memory management.

9.2 toolbox

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. See Figure 9.4.

```
class toolbutton : public button {                          // toolbutton
public:
toolbutton(widget* w, char* name) : button(w,name){         // (a)
    text(name); bind(w,name); pack();                       // (b)
}
};
class toolbox : public frame {                              // toolbox
public:
toolbox(widget* w, tablet* t) : c(t), frame(w,"toolbox"){   // (c)
    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(argv[1]); return OK; }           // (d)
private:
tablet* c;
};
```

Figure 9.4: The *toolbox*

Each individual button is an instance of the class *toolbutton*. When a toolbutton is created (a), the actual button is given the name of the button as its path. Next, (b) the button is given the name as its text, the ancestor 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 *handler* 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 be 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 (c). The

constructor further creates the four toolbuttons.

The *toolbox* is both the superordinate 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*(d)). 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.

9.3 Menus

The second component of our drawing tool is the *menubar*.

The class *menu_bar*, depicted in Figure 9.5 is derived from the *vsh* widget *menubar*. Its constructor requires an ancestor widget, a *tablet* to *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 9.8.

A menubar consists of menubuttons to which actual menus are attached. Each menu consists of a number of entries, which may possibly lead to cascaded menus.

The *file_menu* class, depicted in Figure 9.6, defines a menu, but is derived from *menubutton* in order to attach the menu to its *menubar* ancestor(a). Its constructor defines the appearance of the button and creates a *file_handler* (which will be discussed in section 9.6(b)). It then defines the actual menu(c). The menu must explicitly be attached to the menubutton by invoking the *menubar* member function *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 *menubar* :: *menu* function.

```
class menu_bar : public menubar {              // menu_bar
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);
}
};
```

Figure 9.5: The *menu_bar*

Before defining the various entries of the menu, the *file_menu* instance is declared as the handler for the menu entries(d). However, except for the entry *Quit*, which is handled by calling the *kit* :: *quit* function(e), 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.

9.4 Defining actions - delegation verses inheritance

The most important component of our *drawtool* application is defined by the *tablet* widget class depicted in Figure 9.7. The various modes supported by the drawing tool are enumerated in separate class *drawmode*. The *tablet* class itself inherits from

```
class file_menu : public menubutton {                          // file_menu
public:
file_menu(widget* w,tablet* t) : c(t), menubutton(w,"file") {   // (a)
    configure("-relief sunken -text File"); pack("-side left");
    f = new file_handler(c);                                   // (b)
    class menu* m = new class menu(this,"menu");               // (c)
    this →menu(m);
    m →bind(this);                                             // (d)
    m →entry("Open");
    m →entry("Save");
    m →entry("Quit");
}
int operator()() {                                             // (e)
    if (!strcmp(argv[1],"Quit")) tk →quit();
    else f →dispatch(tk,argc,argv);
    return OK;
}
protected:
tablet* c;
file_handler* f;
};
```

Figure 9.6: The *file_menu*

the *canvas* widget class. This has the advantage that 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 an integer *_mode* variable and an array of handlers that contains the handlers corresponding to the modes supported. See section 9.7 for an example of a typical canvas handler.

Dispatching Although the *tablet* must act as a canvas, the actual widget is nothing but a *frame* that contains a canvas widget as one of its components. See Figure 9.8.

This is reflected in the invocation of the canvas constructor (a). By convention,

```
class drawmode {                                    // drawmode
public: enum { draw, move, box, circle, arrow, lastmode };
};
class tablet : public canvas {                      // tablet
public:
tablet(widget* w, char* options="");
int operator()() {                                  // operator()
    return handlers[_mode] →dispatch(tk,argc,argv);
}
void mode(char* m);
protected:
void init(char* options);
int _mode;
class handler* handlers[drawmode::lastmode];
canvas* c;
};
```

Figure 9.7: The *tablet*

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 widget and redirects the *eval*,*configure*,*bind* and *handler* invocations to the subordinate *canvas* widget(b). It then declares itself to be its own handler, which results in declaring itself to be handler for the canvas component (c). Note, that reversing the order of calling *redirect* and *handler* would be disastrous. After that it creates the handlers for the various modes and sets the initial mode to *move*.

The *operator*() function takes care of dispatching calls to the appropriate handler. The *dispatch* function must be called to pass the *tk*, *argc* and *argv* parameters.

```
tablet::tablet(widget* w, char* options) :      // tablet :: tablet
        canvas(w,"tablet",0) {                  // (a)
    widget* top = new frame(path());
    init(options);
    redirect(c);                                // (b)
    bind(this);                                 // (c)
    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;
}
```

Figure 9.8: Installing the handlers

9.5 Creating new widgets

Having taken care of the basic components of the drawing tool, that is the *toolbox*, *menu_bar*, and the *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 9.6 and 9.8 respectively.

However, before that it will be shown how we may grant the *drawtool* the status of a veritable Tk widget, by defining a *drawtool* handler class and a corresponding *drawtool* widget command. See Figure 9.9.

Defining a widget command involves three steps: (I) the declaration of the binding between a command and a handler, (II) the definition of the *operator*() action function, which actually defines a mini-interpreter, and (III) the definition of the actual creation of the widget and its declaration as a Tcl/Tk command.

Step(I) is straightforward. We need to define an empty handler, which will be associated with the *drawtool* command when starting the application. See Figure 9.10(a).

```
class drawtool : public canvas {                    // drawtool
public:
drawtool() : canvas() { }                           // (I)
drawtool(char* p, char* opts="") : canvas(p,0) {
    init(opts);
    redirect(c);
}
int operator()(){                                   // (II)
    if (!strcmp("self",argv[1]) ) tk →result(self() →path());
    else if ( !strcmp( "drawtool" ,*argv) ) create(-argc,++argv);
    else self() →eval( flatten(-argc,++argv) );
    return OK;
}
protected:
tablet* c;
void init(char* options);
void create(int argc, char* argv[]) {               // (III)
    char* name = *argv;
    tk →action(name, new drawtool(name, flatten(-argc,++argv)));
}
};
```

Figure 9.9: The drawtool widget command

The functionality offered by the interpreter defined by the *operator*() function in (II) is kept rather simple, but may easily be extended. When the first argument of the call is *drawtool*, a new *drawtool* widget is created as specified in (III), 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* canvas. It is the responsibility of the writer of the script that the *self* command is not addressed to the empty handler. If neither of these cases apply, the function *widget* :: *eval* is invoked for *self*, with the remaining arguments flattened to a string. This allows for using the drawtool almost as an ordinary canvas. See the example hypertext script shown in section 9.8.

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

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 the the handler of the command corresponding to its pathname.

```
class application : public session {          // drawtool
public:
application(int argc, char* argv[]) : session(argc,argv,"drawtool") {}
void prelude( kit* tk, int, char** ) {
    init_ht(tk);
    tk →trace();
    tk →action("drawtool", new drawtool());// (a)
}
void main( kit* tk, int, char** ) {
    tk →trace();
    drawtool* d = new drawtool(".draw");
    tk →action("drawtool",d);                 // (b)
    d →rectangle(30,30,80,80,"-fill red");
    d →pack();
}
};
```

Figure 9.10: The drawtool application

The *application* class depicted in Figure 9.10 will by now look familiar, except for the function *prelude*. In the body of the *prelude* function, the tcl command *drawtool* is declared, with an instance of *drawtool* as its handler(a).

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, in order to allow for a script to address the *drawtool* by calling *drawtool*

self.

Discussion The reader may by now have lost track of how delegation within a compound widget takes place. Perhaps 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 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.

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.

9.6 Dialogs

Interactive applications may require the user to type some input after reading a message or to select an item for a list of alternatives. One of the widgets that may be used in a dialog with the user is the *filechooser* widget. The *filechooser* 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 filename directly. In addition, the *filechooser* has an OK button, to

confirm the choice and *cancel* button, to break off the dialog. Below, the construction of simplified version of the *filechooser* will be discussed briefly.

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 in Figure 9.11, that was invoked by *file_menu* in section 9.3

9.6.1 The *file_handler* widget

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

```
class file_handler : public handler {
public:
file_handler( canvas* x ) : c(x) {}
int operator()() {
    if (!strcmp("Open", argv[1])) launch("OPEN");
    else if (!strcmp("Save", argv[1])) launch("SAVE");
    else if (!strcmp("OPEN", argv[1])) open();
    else if (!strcmp("SAVE", argv[1])) save();
    return OK;
}
protected:
canvas* c;
file_chooser* f;
void launch(char* args) { f = new file_chooser(); f →bind(this,args); }
void open() { tk →source( f →get() ); f →destroy(); }
void save() { c →postscript( f →get() ); f →destroy(); }
};
```

Figure 9.11: The *file_handler* class

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 the similar way, the menu entry *Save* results in writing the canvas to a postscript file.

9.6.2 The *file_chooser* widget

Despite its simple appearance, which is left to the imagination of the reader, the *file_chooser* widget has some subtle complexities.

A rudimentary *file_chooser* class is depicted in Figure 9.12. 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, 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, in a function list to fill the *listbox* and the function *install*, which is used to install an external handler for the two buttons widgets. The *install* function is defined as in Figure 9.13.

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 shown in Figure 9.14.

```
class file_chooser : public toplevel {
public:
char* type() { return "file_chooser"; }
file_chooser() : toplevel(gensym()) {init(); }
char* get() { return e →get(); }
int operator()();
protected:
void list();
button* b; button* c;
entry* e; listbox* l;
void init();
void list();
};
```

Figure 9.12: The *file_chooser* class

```
void file_chooser::install(action& a, char* args) {
    b →handler(a,args);
    c →handler(a,args);
}
```

Figure 9.13: *file_chooser*:install

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

The *init* function constructs the various component widgets and an auxiliary frame widget to obtain the desired layout. It further defines the appropriate event binding for the *listbox*, letting the double clicks results in setting the *entry*.

```
void file_chooser::list() {
    sprintf(buf,"foreach i [exec ls -a]
        { %s insert end $i }", l →path());
    tk →eval( buf );
}
```

Figure 9.14: *file_chooser*:list

9.7 Graphics

The Tk canvas widgets offers powerful means for doing structured graphics. The *vsh* 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* class in Figure 9.15. 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 a *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. 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 by calling

```
c->rectangle(x,y,e.x(),e.y());
```

```
class move_handler : public handler {
public:

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

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

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

void release( event& ) { dragging = 0; }
protected:
canvas* c; int dragging; item id; int x,y;
};
```

Figure 9.15: The *move_handler* class

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.

9.8 Hypertext

As described in section 7.3 the *hypertext* widget may be used to display text files containing embedded Tcl code. The Tcl code must be place between escapes, that

take the form %% 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 9.16. Notice that the on-line help provides a replica of the *drawtool* application, surrounded by text. When looking at (again as fragment of) the hypertext file specifying the contents of the on-line help, given in Figure 9.17, you see that the *drawtool* command defined in section 9.5 is employed to create the embedded widget.

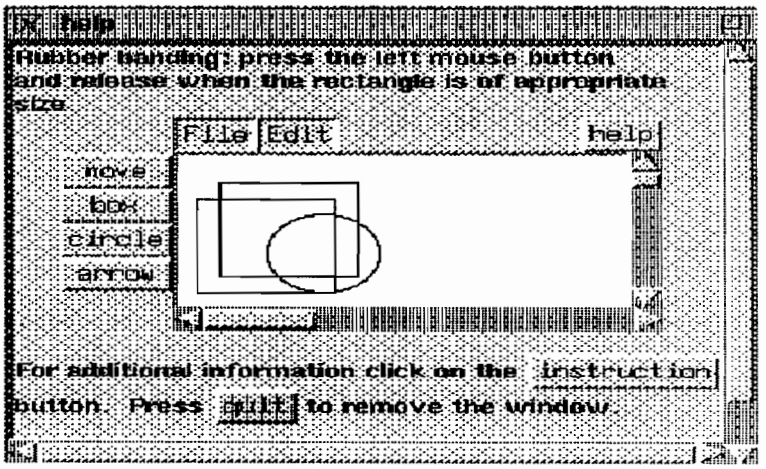

Figure 9.16: Hypertext help

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* to identify the current line number and the current file name.

```
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
%%

For additional information click on the %%
button $this.goto -text instruction
    -command "global EOT; $this gotoline $EOT"
$this append $this.goto
%%
button. Press %%
button $this.quit -command "destroy .help" -text quit -bg pink
$this append quit
%% to remove the window.
%%
global EOT
set EOT [expr $thisline-1 ]
%%
Additonal information ...
```

Figure 9.17: A Hypertext help file

Chapter 10

Related Work

The two most popular window environment nowadays are X-windows (which operates under Unix) and MS-windows (which provides a window environment on top of DOS). Although both environments provide a C interface (as a collection of low-level library functions), most application developers prefer a library that provides a higher-level functionality and pre-defined widgets.

The credo of the consortium supporting X-windows may be expressed as the wish to offer *mechanism instead of policy.* Two standards have been developed which do support *policy* (that is, conventions with respect to the graphical layout of the user interfaces) as well, namely the Motif standard (that is adhered to by many X-window library vendors) and Openlook standard. The following sections of this chapter present an overview of the toolkits (See Figure 10.1) based on these standards and their comparison with *Vsh.*

10.1 ET++

One of the earliest graphic user interface development libraries for C++ is ET++ [WGM88]. Originally developed for Sunview, ET++ is now also available for X-windows. It is part of a programming environment PE++, which includes program

User Interface development

- ET++ - *general*
- InterViews - *interaction*
- Suit - *portable*
- Andrew,Thesesus++ - *hypermedia*
- Vsh - *multiparadigm*

Commercial

- Xv++, StarView, Zinc Interface Library

Figure 10.1: Libraries for GUI development

support tools such as browsers and editors. ET++ is a class library which aims to provide most of the facilities found in the standard Smalltalk class library. It is structured as a single-rooted inheritance hierarchy, with many virtual functions available to provide flexibility and opportunities for extension.

ET stands for "Editor Toolkit" and the original aim of ET++ was to make it easy to build highly interactive tools such as drawing programs. CASE diagrams editors, source code browsers, etc. Many of the ideas from Apple's MacApp system were also borrowed and used. This gives ET++ applications a strong "Macintosh feel".

All classes in ET++ inherit from class Object, which defines the protocol for actions common to all classes. These include comparison of objects, object input/output, status flags and object dependency. Most of the member functions in Object do nothing and exist purely to be over-ridden in subclasses. Each Object can have a collection of dependents, so that whenever it calls the member functions

Changed(), all dependents receive a call to Update(), because it is embedded at the root of the hierarchy. This mechanism was borrowed from Smalltalk-80, and simplifies many problems in interactive systems, such as displaying of multiple views of model objects, or re-sizing containers when their contents alter in shape.

Another feature that ET++ has borrowed from Smalltalk is the MetaClass concept. It can be very useful in object-oriented systems to be able to ask any object what class it is, or to know what instance variable it possesses. Unfortunately, C++ gives no access, at run-time, to this information, so ET++ implements a class Class, analogous to Smalltalk's MetaClass. Besides this ET++ implements a rich set of container classes akin to those found in Smalltalk: Set, Bag, OrdCollection, ObjList, Dictionary and so on. Each one is derived from the abstract class Collection. Collection deals with instances of any class derived from Object, so no compile-time type checking is possible, and users of Collections must type-cast objects extracted from them. This is contrary to the spirit of C++. The IsKindOf() member function provided by ET++'s metaclass system makes it possible to check the types of contained objects at run-time, and to perform "guarded casts", which flags an error if the object is not of the intended type.

To go through each of the elements of a collection, ET++ provides a class Iterator which overrides the () operator to return the next object. This is similar to the system described by Stroustrup, but is considerably more useful because more than one Iterator at a time can be active over a Collection. Each Collection subclass has a corresponding iterator subclass which knows the internal structure of the Collection class, and is a friend of it.

All drawing in ET++ is done through a Port, which defines member functions to draw lines, ovals, text, etc. This Port may be an instance of a SunWindowPort, an XWindowPort or a NeWSWindowPort., without the application caring at all, and the same mechanism is used to implement device-independent printing with a PostscriptPort and PicPort provided. Another class, WindowSystem, is sub-classed by each actual window system implementation to provide management facilities, create windows etc.

Porting ET++ to a new window system is a relatively simple job, with around 35 member functions from Port and WindowSystem to be overridden, plus Font, Bitmap and ClipBoard support. The system is somewhat slanted to SunWindows, the basis for the first ET++ implementation. This doesn't pose any real structural problem but the look and feel of ET++ is hard-wired making it difficult to accommodate a standard user-interface definition such as Motif or OpenLook.

Another concern related to ET++ applications is the large size of binaries. This is because most classes use several other classes to get their jobs done. For example a Cluster uses a Collection to hold its VObjects, and Collection relies on the inherited behaviour of Object for most of its functions, and so on. This is a good evidence for ET++'s reusability, but it means that it is impossible to use one part of ET++ without needing to link in the whole system, and so the smallest ET++ programs are over one megabytes in size. The largest programs are not a lot larger, however, as they reuse more of the core system.

10.2 InterViews

Another important library for developing user interfaces (under X-windows) in C++ is the InterViews library [LVC89]. Intended as a research vehicle, the InterViews library provides an important example of designing an object-oriented library. The name InterViews comes from *Inter*active *Views*, as it is a library of C++ classes that can be used to construct a graphical user interface from interactive components. It supports object interaction mechanism related to the Model-View-Controller (MVC) paradigm [KP88]. Like MacApp [Sch86] and Smalltalk [Gol84], the approach used in InterViews separates interactive behaviour from abstract behaviour. An interactive object, called a *view*, defines the user interface to an abstract object, called the *subject*. The separation of subject and view supports different views of the same subject to suit the particular application or to customize interactive style. A view can be customized dynamically using a *metaview*, a view of another view's internal

state. For example a metaview might allow the user to interactively modify the mapping from keystrokes commands in a text view.

Building graphical interfaces from reusable components requires the ability to define an interactive object that can be used in variety of contexts. To fulfill this requirement, one must consider the way in which the characteristics of a component and its context affect each other.

In InterViews, each interactive component, called an *interactor*, has a preferred shape and size. The preferred shape and size of composition of components, called a *scene*, is calculated from those of the components. However, the actual display space allocated to an interactor is responsible for making best use of the space it has been allocated. Different scenes allocate display space to component interactors using different algorithms. For example, a *box* tiles its components, but a *tray* allows them to overlap.

In extension to existing components there is a set of "flyweight" components, called Glyphs, that are simple and efficient. The glyph base class defines a protocol for drawing; subclasses define specific appearances such as graphic primitives (lines and circles), textual primitives (characters and spaces), and composite objects (tiling and overlays). Applications define their appearance by building hierarchies of glyphs.

InterViews primitive class operations make direct X library calls to implement their semantics. The key issues in interfacing to X were managing X windows and translating X input events into InterViews events. Each canvas is represented as an X-window. The world's canvas is the root window for a display. The scene class contains operations to handle the creation, mapping and configuration of windows. It also includes a library, Unidraw [VL90] for structured graphics and even provides a user interface builder, Ibuild [VT91].

Experience however shows that, although it is written in C++ and claims to be object-oriented, the primary benefit of the system comes from its support for composition (mechanism for assembling independent widgets into interesting arrangements) along with a variety of predefined object to use, instead of providing inheritance.

10.3 Andrew

The Andrew toolkit [Pal88] is an object-oriented system based on a minimal protocol that allows components to communicate with each other about user interface policies, while allowing the developer maximum freedom to determine the actual interactions between components.

The Andrew Toolkit was built using an object-oriented system called the Andrew Class System. This system provides the ability to dynamically load and link code, which in turn provides a powerful extension capability for applications. Furthermore, the dynamic loading facility can be used to add additional components to the basic Toolkit without having to rebuild applications. This feature has been used to build a generic, multi-media editor, EZ, that can edit a wide variety of components by loading the appropriate code when needed.

The Andrew Toolkit is based on the development of components that can be used as building blocks for applications as well as more complex components. The Data objects and Views are the Toolkit's basic object types; a Toolkit component is usually made of a view/data object pair. While the data object contains information that is to be displayed, the view contains information about how data is to be displayed and how the user is to manipulate the data object. For example, the text data object contains the actual characters, style information, and pointers to embedded data objects, as well as ways to alter the data, such as inserting and deleting characters. The text view is made of information such as the location of the text, what portion of text is currently visible, and what piece of text is currently selected. The text view provides ways to draw the text, handle various input events, and manipulate the visual representation of the text.

The contents of a data object can be saved in a file, but the contents of a view cannot. The information associated with a view is transient and is valid only while the application is running. When the application terminates, that information has no further meaning. However, views do provide the facility for printing within the Andrew Toolkit.

While it is often the case that a view has an underlying data object, there are many cases when a view is used solely to provide a user interface function. In such a case, there is no underlying data object. The scrollbar is one such example; it only adjusts the information contained in another view.

The view/data object distinction has been made to provide a system where multiple views can simultaneously display information contained in a single data object. The design is similar to the Model-View-Controller design used in Smalltalk systems.

Despite its advantages, there is a cost to separating information into data objects and views. Two particular areas of difficulties are coordinating data objects and views, and maintaining a stable view state. The system does not encourage a close connection between the changing of the information contained in a data object and the update of the visual appearance provided by the view. Since only one view causes the data object to change, and multiple views may have to reflect the change, a delayed update mechanism must be used. When the user issues a command to a view to alter the underlying data object, the view first requests that the data object modify itself, then requests that the data object inform all its views of the change. When a view is informed that the underlying data object has changed, it must determine what the change is and appropriately update its visual image.

The delayed update mechanism is the trickiest challenge in building a data object/view pair. The developer must create a mechanism with which the view can determine what portion of the data object has changed. This method is normally provided by a set of methods exported by the data object. However, it is not considered proper for the data object to have detailed knowledge of a specific type of view. While this is one way to handle the delayed update, it precludes the development of other kinds of views on the same type of data object.

The Toolkit provides the usual set of simple components, such as menus and scrollbars, and a number of higher-level, editable components, including multi-font text, tables and spreadsheets, drawings, equations, rasters, and simple animations. The text and table components are multi-media; they allow the embedding of other components within their description.

In addition to the editor, a number of basic applications have been developed, including a mail system, a help system, a typescript facility that provides an enhanced interface to the C-shell, a ditroff previewer, and a system monitor, Console, that displays status information such as the time, the date, the CPU load, and file system information. Since both the mail and help applications use the text component for the display of information, they automatically inherit the multi-media functionality of the text component.

The Andrew Toolkit has been designed to be window-system independent. It currently runs on two window systems, including X.11, and can be ported easily to others.

10.4 SUIT

SUIT, the Simple User Interface Toolkit, is subroutine library which helps C programmers create graphical user interfaces that may be modified interactively. SUIT acts as a window manager for screen objects such as buttons, scroll bars, and menus. As a SUIT-based program execute, user may change attributes of the screen objects including an object's location and appearance. The changes to these attributes are then saved with the program.

The model of computation, in SUIT, is based on external control, that is commonly used in event driven scenarios. The thread of control lies in the hands of the user; the main loop is the server that handles the mouse and keyboard events and dispatches them to the appropriate widgets.

The SUIT library was developed at the University of Virginia to help C programmers create sophisticated mouse based interfaces without the lengthy learning period associated with traditional user interface toolkits. Ease of learning and fast ramp up time is central to SUIT's design. The SUIT tutorial is designed to make the user productive in under two hours.

Also central to SUIT design is portability. SUIT programs currently run without changes to the source code on the following platforms:

- IBM PC
- Macintosh
- Sun3
- Sun4 (SparcStation)
- SGI (Silicon graphics IRIS workstations)
- DECstation
- HP

The simplicity of the toolkit has its cost associated too. Each time when the user quits a SUIT application, the properties associated with each of the widgets are saved in a file called an ".sui" file. Not only does this file contain all the information necessary to display the user's application's interface (all the widget location and sizes) but it also keeps a record of the state the program was in when he or she last left it (the values of all the bounded values, the choices that were selected for each radio button in the interface etc.). If a SUIT program starts up without an accompanying ".sui" file, the widgets appear on the screen in random locations with all properties given default values. This hard wiring of interface affects considerably on the performance of the application.

10.5 Theseus++

Theseus++ [Din90] is an object oriented high level user interface toolkit designed to ease development of application-specific interaction with 21/2D graphics.

The basic building blocks of Theseus++ for creating user interfaces are *objects*. Objects can be used for modeling interactions, presentations and graphical constraints. Each object is an instance of its corresponding Theseus++ *class*.

Interactions, which are the instances of the interaction classes, describe basic interaction techniques and mechanism to structure interaction techniques. The root class of the Theseus++ class tree, called *UIInteractionObject*, defines a general frame for any kind of interaction. This includes *components*, *attributes*, and *working scheme* for the user input.

Components connect application-specific and dialogue-specific functionality (realized as methods) to interconnections, attribute control and behaviour of interactions. Each dialogue can be seen as a hierarchy of subdialogues. From the leaves to the root of that hierarchy, the degree of abstraction increases. Theseus++ provides the concept of the *complex interactions* to model the existing abstraction levels in the dialogue in a direct way. Complex interactions control other interactions (subdialogues). These subdialogues are called elements of the complex interaction. An arbitrary amount of dialogues and subdialogues can be active at any time. Interactions that can be triggered with a single, atomic action are *basic interactions*. Basic interactions are buttons, dragging of graphical objects and input via keyboard.

The presentation manager provides *presentations* and *constraints* to the application as well as to the output related to interaction components (prompt and feedback). Presentations can be used by application programmers and dialogue designers to compose both domain-specific pictures and visual feedback of interactions. Constraints model dynamic layout relations without invoking the application.

There are two main classes for presentations: *Basic presentations* correspond to the *Primitives* of graphical standards like GKS or PHIGS. Basic presentations are Lines, Fill Areas, Text and Marker objects. The structural relations in pictures are modeled by using *complex presentations*. They allow to express *Part-Of-Hierarchies*. The incremental creation of new levels in the picture hierarchy by using complex presentations results in a tree presentations. There can be many such trees at any time, while the root of every tree has to be a screen window.

The main feature of Theseus++ is to enforce the separation between the application and user interface in a flexible way. Parts of a dialogue, those don't require application-specific knowledge, can be described on a higher level of abstraction, while semantic feedback on a low level is possible for those dialogue cycles needing application knowledge. This is accomplished by the following features:

- A description model for the separate specification of interaction components like prompt, feedback, and semantics on each level of the interaction hierarchy.
- A dynamic composition mechanism both for definition of complex interactions (dialogue hierarchy) and for complex graphical objects (picture hierarchy).
- The support for constraint based direct manipulation-defined graphics including continuous interactions like dragging and stretching.
- A mechanism to support extensibility by new classes of domain-specific interaction techniques and graphical objects.

Theseus++ is implemented in C++ on top of OSF/Motif and the X-Windows system. The Theseus++ library also provides a collection of classes for developing hypermedia systems.

10.6 Comparison

Lastly I mention the *vsh* library, which provides a multiparadigm framework for user interfaces operating under X-windows. There are similarities between *Vsh* and InterViews and Andrew toolkits in that all support some sort of widget-like notation to decompose applications and support for underlying application object structures.

The most significant difference between *vsh* and the other toolkits is the presence of Tcl in *vsh*. Run time languages are starting to appear in other systems, such as Ness, which is used to embed executable programs into documents in the Andrew

toolkit [Han90], and UIL, which is used to specify interfaces in Motif [OSF90]. However, these languages have three disadvantages relative to Tcl. First, they are less dynamic. For example, UIL programs must be compiled before being processed by running applications, and Ness appears to require many decisions to be made statically. In contrast, Tcl is interpretive, so any available operation can be invoked any time. Second, the other languages are less complete. For example, UIL does not include control constructs such as **if** and **while**, and Ness functions are not first-class objects. In contrast, Tcl is a complete programming language that even provides access to its own internals (e.g. it is possible to retrieve the body of a Tcl procedure or a list of all defined variable names (Chapter 8)). Third, the other languages are special-purpose: they only control a portion of an application's functions. In contrast, Tcl is used for virtually all aspects of an application, which makes it possible to compose all of those aspects to work together.

Another difference between *vsh* and other toolkits is Tcl's **send** command for inter-application communication. I know of no equivalent construct in other X toolkits. The closest existing facility is Microsoft Windows's Dynamic Data exchange protocol (DDE), which allows applications to communicate in several ways including passing commands for remote executions [Mic90]. However, for remote execution to be most useful it must access to all the internals of the remote application. For this to happen, the language used by the remote execution facility should be the same as the language used to control the user interface and internals of the target application, as it is with Tcl and in turn *vsh*. Unfortunately, the Windows environment does not include a universal command language. Although a standard syntax is suggested for remote commands, there is no built-in connection between these remote commands and the internals of the remote application. Each application must provide special code to parse and execute all the remote commands it wishes to support. This will probably limit the use of of remote execution in DDE to small set of functions. In contrast, **send** command provides access to all aspects of other Tk-based or *vsh* based applications without any extra effort on the part of applications' developers.

One final difference between *vsh* and other toolkits is the level of object-orientation.

Et++ and Andrew are strongly object-oriented with support for hierarchical classes and inheritance. In contrast, *vsh* is not strongly object-oriented, although it has a class mechanism, and provides inheritance in widgets implementation. It focuses on *composition*. In my opinion composition is more important for a toolkit than inheritance. There isn't enough commonality between widgets for inheritance to provide much benefit. As experience with some of the toolkit shows deep hierarchy and inheritance adds complexity (to understand one widget you must understand all the widgets it inherits from), whereas composition mechanism allow any user to create new interface elements out of existing widgets. Further support for this comes from the InterViews system: although it is written in C++ and claims to be object-oriented, the primary benefit claimed for the system is its support for composition [LVC89].

For the commercial libraries a distinction may be made between the libraries that claim to be platform-independent and the libraries associated (such as StarView and the Zinc Interface Library) provide an interface that is portable across X-windows and MS-windows environment. For example, StarView even supports OS/2, MS-Windows, OSF/Motif, OPENLOOK and the MacIntosh window system.

Bibliography

[Con87] J. Conklin. Hypertext: An introduction and survey. *IEEE Computer*, 20(9):17–41, 1987.

[Din90] Dennis Dingeldein. Theseus++: A high level user interface toolkit for graphical applications. *TH Darmstadt, FB Informatik, FG Graphisch-Interaktive Systeme*, pages 6–21, December 1990.

[Gol84] A. Goldberg. *Smalltalk-80: The Interactive Programming Environment.* Addison-Wesley, Reading, Massachusetts, 1984.

[Han90] W. Hansen. Enhancing documents with embedded programs: How ness extends insets in the andrew toolkit. In *Proc. 1990 International conference on Computer languages*, pages 1–17, March 1990.

[KP88] G.E. Krasner and S.T. Pope. A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. *JOOP*, pages 26–49, 1988.

[Lip91] S. Lippman. *A C++ Primer.* Addison-Wesley, 2nd edn, 1991.

[LVC89] M. Linton, J. Vlissides, and P. Calder. Composing user interfaces with interviews. *IEEE Computer*, 22(2):8–22, 1989.

[Mic90] *Microsoft Windows Software Development Kit, Guide to Programming, Version 3.0.* Microsoft Corporation, 1990.

[Mye92] Brad A. Myers. *Languages for Developing User Interfaces.* editor, Jones and Bartlett, 1992.

[OSF90] *OSF/Motif Programmer's Guide, Revision 1.0.* Prentice Hall, Englewood Cliffs, NJ, 1990.

[Ous90] John Ousterhout. Tcl : An embeddable command language. In *Inproceedings of the winter 1990 USENIX Conference*, pages 133–146, Berkeley (CA), USA, January 1990. Usenix Association.

[Ous91] John Ousterhout. An x11 toolkit based on tcl language. In *Proceedings of Winter 1991 USENIX Conference*, pages 105–115, Berkely (CA),USA, January 1991. Usenix Association.

[Ous93] John Ousterhout. Hypertext and hypermedia. *Usenix Conference*, October 1993.

[Ous94] John Ousterhout. *Tcl and the Tk toolkit.* Addison-Wesley, 1994.

[Pal88] Andrew Paley. The andrew tollkit- an overview. In *Proc. USENIX Winter Conference*, pages 133–146, January 1988.

[Sch86] K. J. Schmucker. *Object-Oriented Programming for the Macintosh.* Hayden, Hasbrouck Heights, New Jersey, 1986.

[Str91] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, 2nd edn, 1991.

[Str93] H. Strickland. Odmg-93 -the object database standard for c++. In *C++ Report*, pages 45–70, October 1993.

[VL90] J. Vlissides and M. Linton. Unidraw: A framework for building domain-specific graphical editors. *ACM Transactions of Information Systems*, 8(2):237–268, 1990.

[VT91] J. Vlissides and S. Tang. Ibuild: A unidraw-based user interface builder. *In Proceedings ACM SIGGRAPH Symposium on User Interface Software amd Technology*, 1991.

[WGM88] A. Weinand, E. Gamma, and R. Marty. Et++, an object-oriented application framework in c++. In *OOPSLA'88*, pages 56–77, 1988. Springer.