McGill University
School of Computer Science
ACAPS Laboratory

Advanced Compilers, Architectures and Parallel Systems

Wish++ : The C++ API for Tcl/Tk

Vipul Jain

ACAPS Technical Memo 87
August 5, 1994

ACAPS • School of Computer Science • 3480 University St. • Montréal • Canada • H3A 2A7

Abstract

This paper describes the C++ programmer interface to *wish++, the object oriented windowing shell* based on Tcl/Tk. 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 *wish++* library contains classes that provide convenient yet flexible access to the functionality offered by the Tk toolkit. The library is intended to support the needs of both novice and experienced (window) programmers. It offers widget and graphics classes with a simple interface, but allows more experienced programmers also to employ the Tcl scripting language to define behavior and functionality of widget 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 biasing *wish++* 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.

Contents

1 Introduction

In comparison with ordinary programming (in C++), programming in a window environment (in C++) introduces a number of additional difficulties. First of all, the programmer must become acquainted with the various widgets constituting the (graphical) user interface, such as buttons, menus, messages, canvases, etcetera. And secondly, perha[s 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 toolkit for X11 environment with an interface to C++ do exist already. Well known for example is the InterViews Library, which offers powerful features for defining the layout of graphical user interfaces [LVC89]. However, despite the elegance of its design, InterViews is slightly cumbersome to use and lacks a number of the features and widgets needed for rapidly implementing a graphical user interface.

Commercial packages for GUI programming in C++ are available. The disadvantage of these packages, apart form their price, is primarily they do not offer the flexibility needed in research environment.

A rather different approach to GUI programming has been advocated by Ousterhout [Joh91], which describes the Tcl/Tk toolkit. Tcl is a cshell-like (interpreted) script language that may be embedded in C or C++. Tk is a window and graphics toolkit based on X11, partly implemented in Tcl and partly in C. Tk offers numerous widgets, including a powerful canvas and text widget. Moreover, the scripting language allows 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.

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

Tcl is an interpreted language. Tcl interpreters may be incorporated in C program, hence Tcl may be thought of being embeddable in C/C++. The Tcl/Tk environment defines an interface to C. Lacking thus far, however, for Tcl/Tk is an adequate programmer interface

(API) to C++. This chapter describes the C++ programmer interface to *wish++, the object oriented shell* based on Tcl/Tk. The *wish++* library contains classes that provide convenient yet flexible access to functionality offered by the underlying toolkit. Its contribution is to provide type secure solutions for connecting Tcl and C++ code and that it allows the use of inheritance to refine the behaviour of (compositions of) widgets.

The *wish++* library is intended to support the needs of both novice and experienced (window) programmers also to employ the Tcl scripting language to define the behavior and functionality of widget 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 biasing *wish++* 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.

The structure of this chapter is as follows. First, some background information concerning Tcl/Tk will be given, including a brief example. Then the structure of *wish++* program will be sketched, involving the classes *kit* and *session*. Next, the notion of widgets is explained and how widgets may be used to configure a graphical user interface. Then, we will look at events and how events may be associated with actions and callbacks. Also, we will look at type-secure way to deal with callback client information, involving handler objects. After that, a number of commonly used widgets will be described. Examples will be given to explain to illustrate their use. For the canvas widget in particular it will be explained how graphical items may be created and manipulated, for advanced users, it will be explained how to employ the scripting language directly from C++, and how to use functionality defined in C++ again in a script. As a fine example, it will be explained how to construct a new widget, employing both Tcl and C++.

2 Background-Tcl/Tk

The language Tcl has been presented in [Joh90]. 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 [Joh91]. 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

The Tcl language offers variables, assignments and 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. (Thus far a number of extensions to Tcl have been proposed, including an object-oriented *incr Tcl* and extensions offering facilities for distributed programming).

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 below.

Scripts - *hello world*

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

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, as 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 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.

3 Program structure

The *wish++* library is intended to provide convenient way to program Tcl/Tk based applications in C++. We will start by looking at the structure of programs employing wish++. In an ordinary C++ program the function *main* is used to start the computation. Control is effected by creating objects and calling the appropriate functions. When programming in window based application, at a certain moment control is delegated to the window environment. Consequently, there needs to be some kind of main loop which waits for incoming events, in response to which the control may be delegated to an appropriate component of the program.

Since programming an event dispatch loop by hand is very cumbersome, them main (event dispatch) loop is usually hidden, and instead programmer is required to declare callback functions that will be activated in response to particular event.

```
#include< vsh.h >

void program(kit* tk, int argc, char* argv){
     message * m = new message(".m");
     m ->text("hello world");
     tk ->pack(m);
}

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

To hide the details of activating the main loop and dispatching of events, *wish++* provides a class *session* of which an instance must be created to start computation.

Above an example is given of a simple *hello world* program employing wish++. It resembles the example given previously, except that a message is used instead of button widget. In *main* an instance of *session* is created which is then told to run the function *program*. As a result, *program* is called with, as parameters, a pointer to an instance of *kit* (that is used to communicate with the underlying toolkit) and the original parameters to *main*. The function *program* merely creates a message saying *hello world* and displays this in a window on the screen by telling the toolkit to append it to the root window.

3.1 The interpreter *kit*

Wish++ is meant to provide a parsimonious C++ interface to Tcl/Tk. Neverthless, as with many a toolkit, some kind of API shock seems to be unavoidable. This is specially true for the class *kit* defining the interface. 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. The reader will undoubtly gradually learn the functionality of *kit* by studying the examples.

The *kit* class is derived from the class *vcl* which provides access to the lower level details of (an extended version of) Tcl.

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. The function *after* can used to have a Tcl command evaluated after n milliseconds (It makes use of X time callbacks). The function *send* may be used to evaluate a command in different applications. An application may be given a name when creating the *session* object. The function *selection* delivers the current X selection. The function *update* may be used to process any pending event. These functions are only for experienced users.

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 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();
}
```

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 4.

The function *action* may be used to associate a Tcl command to a command function to handler written in C++. See section 5.2 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.

3.2 A *session*

To initialize an instance of the *kit* and to start the main (event dispatch) loop an instance of *session* has been inspired directly by a similar class in InterViews library. However, programs written with *wish++* may also be used as an interpreter of Tcl Scripts. Consequently, the class *session* offers functions to initialize and install the functionality needed apart from function to start the main loop.

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);
}
```

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*.

4 Configuration 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.

Widgets in Tk are identified by a *path name*. The path name of a widget reflects its possible subordination to another widget. The format of a path name may be expressed in BNF form as

$$path ::= < empty > |'.' < string > < path >$$

where $< empty >$ denotes the empty string. For example "." is the pathname of the root widget, whereas ".quit" is the pathmane 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.

Widgets may be configured in a variety of ways. For example most widgets allow specification of a foreground and a background color. The *wish++* interface delegates most of these issues to the underlying toolkit. Users wishing to improve on the default appearance of the widgets must study the *options* provided by Tk.

Naming widgets in a hierarchical fashion does not imply that widgets behave accordingly. To group widgets, they must be packed in common ancestor widget, that is a *frame* or *toplevel* widget. Alternatively, the Tcl command *pack* may be used. To map a widget to the screen it must either be packed in the *root* widget, using *kit::pack*, or to a *toplevel* widget, using *widget::pack*. [1]

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.

[1] The *pack* command offered by Tcl/Tk is likely to change in the near future. This may affect the corresponding member functions for *kit* and *widgets* as well.

Widget

```
interface widget:handler{
    widget(char* p);
    widget(widget& w,char* p);
    char* type();
    char* path();
    int eval(char* cmd);
    char* result();
    char* evaluate();
    virtual void configure(char* cmd);
    widget* pack(char* wp, char* options = "top");
    widget* pack(widget* w, char* options = "top");
    virtual bind(char* b, char* cmd, char* args = "");
    virtual bind(char *s, command proc, client* data = 0, char* args = "" );
    virtual bind(char *s, handler* h, char* args = "" );
    virtual bind(char *s, action& a, char* args = "" );
    void handler(char* cmd, char* args = "");
    void handler(command proc, client* data = 0, char* args = "" );
    void handler(class handler* h, char* args = "" );
    void handler(action& ac, char* args = "" );
    void source(char* type, char* args, char* options="");
    void target(char* type, action& ac, char* args, char* options="");
    void focus(char* options = "");
    void grab(char* options = "");
    void destroy();
    Tk_Window tkwin();
protected;
    virtual install(action&, char* args="");
}
```

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

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

does not result in creating actual widget but only defines a pointer to the widget with 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 path name *path* to the path name of the argument widget *w*.

The function *path* delivers the pathname of a widget object. Each widget created by Tk actually defines a Tcl command associated with the path name 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 background color by a 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 *pack* is used to pack widgets together. The *options* parameter of *pack* may be used to instruct the geometry manager. By default, widgets will be appended to the top of the remaining space in the widget to which it is appended.

The function *bind* is used to associate actions with events. These associations will exists only for the widget for which they are declared. The function *bind* comes in four forms. The first form associates a Tcl command with events specified by *b*. The second and third form associate an action either defined by a command or a *handler* with the events specified by *b*. The fourth version takes an instance of an action as argument. Actually, the first three forms provide an abbreviation for the last form. See section 5.2. For the various bindings allowed consult the manual page for *bind*.

The function *handler* has also four forms, analogously to *bind*. It installs an action for the default bindings of a particular widget. The protected virtual function *install* is assumed to define the default building for each widget. Only for widgets for which *install* is (re)defined, invoking *handler* will have meaning.

The functions *source* and *target* may be used to declare the widget as respectively a source or target for a *drag&drop* operation.

The functions *focus*, *grab* and *display* correspond with Tcl/Tk functions of the same name.

To get hold of the underlying $Tk_W indow$ object, the user may employ the function *tkwin*. This is needed when for example one wishes to manipulate bitmaps directly, by using Tk functions.

The functions *configure*, *pack*, *bind* and *handler* are declared virtual, since user defined widgets consisting of multiple widgets may need to redirect these commands to a component widget.

5 Binding actions 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. This is automatically taken care of by the (toolkit) environment. The standard C interface for Tcl offers only limited support to define callback functions and to deal with events. The $wish++$ C++ interface extends the standard Tcl C interface by *commands* that may ask the *kit* for the latest event, and more importantly by *handler* objects that allow to deal with *client* information and events in a type-secure way.

5.1 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 latesr 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();
}
```

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.

5.2 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**);
```

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.

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();
}
```

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 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 proffered 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 Below, the declaration and use of an *action* is shown in 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.

In the example, the function f is invoked by calling the Tcl scripts command $do-something$ via $kit :: eval$.

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();
};
```

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.

5.3 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 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;
};
```

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 above, 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 7.2 for an example.)

Dispatching

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

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();
};
```

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.

6 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. The *wish* + + 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 *wish*++ 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* widget. In section 9 it will be explained how to create new widgets in C++ and make them available for use in a Tcl script.

In this section only a limited number of the available widgets will be described[2]

Pathname When creating an instance of a widget a *pathname* must be given. In order to allow for a more flexible naming scheme, for each widget class a second constructor has been provided that has a widget pointer as its first parameter. This widget acts as the parent of the newly created widget [3] The pathname of the newly created widget will be the concatenation of the pathname of the (parent) widget and the pathmname given as the second parameter.

Options Both constructors allow for *options* parameters which may be used to set the options for the widget as defined for Tk.The functions provided for each widget classes are merely meant to provide convenient access to the Tk widgets. More demanding programmers may use the constructor *options* parameter, or the $widget :: eval$ or $widget :: configure$ functions instead.

Creation and inheritance To accommodate the use of inheritance to redefine the behavior of widgets or to create compound widgets, the constructors take apart from the *options* parameter a boolean flag (which is default set to true). If the flag is set to false, the initialization code for a widget will not be executed. Apart, from omitting initialization in the context of inheritance, the user may also wish to omit the initialization to create a reference to already existing widget. We will display the boolean $flag$ parameter in the widget interfaces shown here. When a widget class does not correspond to a widget supported by Tk, mostly an additional constructor will be defined that is used create a *handler* to deal with the creation of such widgets in Tcl scripts. For an example, see section 9.

Handlers For a number of widgets a *handler* may be defined. A *handler*, in this context, is an *action* associated with the events related to the default bindings for the widget. Do not confuse the *handler* function defined for a *widget* object with the *handler* objects that may be used to define *action*. The $widget :: handler$ function may itself be given a handler object as parameter.

An example of a *command* function that may be installed as a handler for widget (by means of $widget :: handler$) is the function *browse* defined below.

[2]For additional widgets you can refer to the manual pages accompanying $wish++$.
[3]Currently, the child widget must explicitly be packed.

Browse function

```
int browse(client* data, kit* tk, int argc, char** argv) {
    widget* w = (widget *)data;
    cout << w →type() << " " << w → path() << "::";
    while (argc− > 0) {
    cout << *argv++;
    }
    cout << endl;
    return OK;
}
```

The *browse* function assumes that a *widget* has been defined as *client*. It simply prints the *type* and the *pathname* of the widget, and in addition the arguments given when calling the *action* related to the function. Widgets for whcih the declaration of a handler is meaningful are assumed to have a protected member function *install* defining the default bindings. Naturally, the function *browse* may also be used to define the action related to a user-defined binding.

6.1 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*, and will not be further discussed here.

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

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.)

Example As an example of the definition and use of a button look at the following program fragment.

Example

```
void program(kit* tk,int,char** ){
    tk →trace();
    button* b1 = new button(".b1");
    b1 →text("ok");
    b1 →handler(browse,b1,"first button");
    button* b2 = new button(".b2","-bg blue");
    b1 →bitmap("bitmaps/queen.bm");
    b1 →handler(browse,b2,"second button");
    tk →pack(b1) →pack(b2);
    tk →pack(".quit");
}
```

Two buttons are created. The first button displays some text and the second bitmap. The second button is given a colored background by using the *options* argument of the constructor. For both buttons, the function *brows* is installed as the (default) handler, with as *client* the appropriate pointer. For each button, arguments are defined that will be given to *browse* when invoking the action. Both buttons are subsequently appended to the root. In addition, a third (system-defined) button is appended to the root. This button has been defined in Tk as

```
button .quit -text quit -command destroy .
```

which results in button *.quit* that, when passed, destroys the toplevel root window of the application and all its subordinates.

6.2 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.

```
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);
}
```

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 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 descedent 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 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*.

Example The function *select* shown below merely prints the (index of) the currently active entry for the menu. This may be used in a switch statement to determine the appropriate action.

```
int select(client* data, kit* , int argc, char** argv) {
    menu* m = (menu*) data;
    int n = m →active();
    cout << "entry" << n << endl;
    return OK;
}
```

Note that the *client* parameter is assumed to be a menu. The example below shows how *menus* embedded in a *menubutton* may be organized in *menubar*.

```
void program(kit* tk,int,char** ){
    widget* bar = newmenubar(".bar");
    menubutton* b1 = new menubutton(bar,".b1");
    b1 →text("one");
    menubutton* b2 = new menubutton(bar,".b2");
    b2 →text("two");
    menu* m1 = new menu(b1,".m"); b1 →menu(m1);
    menu* m2 = new menu(b2,".m"); b2 →menu(m2);
    m1 →handler(select,m1);
    m2 →handler(select,m2);
    m1 →add("zero");m1 →add("one");
    m2 →add("second");m2 →add("third");
    bar →append(b1) →append(b2);
    tk →append(bar); tk →append(".quit");
}
```

The pathnames of the menubutton are made dependent on the pathname of the menubar, and, analogously, the pathname of each menu depends on the pathname of the menubutton they are associated with. Before adding the entries, the function *select* is installed as a handler. Entries occur in the menu in the order that they are added. Finally, the buttons are packed in the menubar and the bar and quit buttons are packed in the root widget.

6.3 Dialogues

Interactive applications may require the user to type some input after reading a message or to select an item from a list of alternatives. Among the widgets that may be involved in such an interaction are the *message* widget, the *entry* widget and the *listbox* widget. An example of a (compound) widget combining these widgets is the *filechooser* widget, that is discussed in section 9.

Another example of a compound widget is given in the function *dialog* below. An essential component of (compound) dialogue widgets is usually an *entry* widget, that may be used to obtain (typed) input from the user.

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

The *entry* class shown above 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.

Example below an example is given of a function to create a user-defined compound dialogue widget. The compound widget consists of *toplevel* widget (which makes that the dialogue widget is directly mapped to the screen as a toplevel window), a *message* widget (to display a message), an *entry* widget (to allow the user to enter text) and two *button* widgets (one to acknowledge the answer and the other to cancel the dialogue).

The text typed in the *entry* widget is retrieved by the function *browse* defined below.

Browse-*entry*

```
int browse(client* data,kit* ,int, char**){
    entry* e = (entry* )data;
    cout << e ->get() << endl;
    return OK;
}
```

The actual *dialogue* function is defined as a handler function that may be attached to any widget for example button. The function *dialog* first creates a *toplevel* widget, and subsequently a *message* widget and an *entry* widget. Both the *message* widget and the *entry* widget are made subroutines to the toplevel widget. Next a *frame* widget is created, to hold the two *button* widgets, *ok* and *cancel*. The *ok* button is given the *browse* function as a handler, with the entry widget as client data. The handler for the *cancel* button is defined directly by a Tcl command which results in destroying the toplevel window. Both buttons must be explicitly packed in the frame widget. After that, the frame widget is packed together with the message widget and entry widget in the toplevel widget. Before leaving the function, the focus is set to entry widget, which effects the cursor to start blinking in the entry widget.

Dialogue - *to be attached to a button*

```
int dialog(client* ,kit* ,int i,char** ){
    toplevel* t = new toplevel(".toplevel");
    message* m = new message(t,".message");
    m →text("Type your answer");
    entry* e = new entry(t,".entry","-relief sunken");
    frame* f = new frame(t,".frame");
    button* ok = new button(f,".ok");
    ok →text("OK");
    ok →handler(browse,e);
    button* cancel = new button(f,".cancel");
    cancel →text("cancel");
    cancel →handler("destroy .toplevel");
    f →pack(cancel,"{left fillx}");
    f →pack(ok,"{padx 5 right fillx filly}");
    t →pack(m); t → pack(e);
    t →pack(f);
    e →focus(); /// to attract the focus
    return OK;
}
```

In the example, occasionally the option parameters are used in order to make the appearance of the widget nicer.

7 Graphics and Hypertext

The Tk toolkit offers powerful facilities and (hyper)text [Joh93]. 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 Class *item*

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 *wish++* library contains a class *item* of which the functionality is shown below.

```
interface item{
    operator int();
    void tag(char* s);
    char *tags();
    void move(int x,int y);
}
```

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 the future, this class *item* can be extended. Also, it is conceivable that more specific *item* classes, corresponding to the item types offered by the canvas widget, will be defined as subclass of *item*.

7.2 The *canvas* widget

The Tk canvas widget offers powerful means for doing structural graphics. The *wish++* 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.

```
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);
      item itemconfigure(int it, char* options);
      item itemconfigure(char* tag, char* options);
}
```

Example As an example of the use of a canvas, consider the handler class defined below. The *canvashandler* class is derived from the class *handler* introduced in section defined for handler, but redefines the (virtual) function *press, motion* and *release*.

```
class canvashandler: public handler {
    public
    canvashandler(canvas* cv) { c = cv; moving = 0 }
    void press(event& e);
        x = e.x(); y = e.y();
        id = c →overlapping(x,y);
        if (id) moving = 1;
    }
    void release(event&) {moving = 0};
    protected:
    canvas* c;
    int x, y;
    int moving;
    item id;
}
```

When creating an instance of *canvashandler*, 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.

```
void program(kit* tk, int, char** ){
    canvas* c = new canvas(".canvas");
    c →handler(new canvas_handler(c);
    c →oval(120,60,180,180,"-fill blue");
    c →bitmap(30,50,"@bishop.bm");
    tk →pack(c) → append(".quit");
}
```

To install an instance of *canvashandler* as a handler for the canvas, the function *widget* :: *handler* is used. Note that the *canvashandler* is given a reference to the canvas.

7.3 The *Hypertext* widget

Both the Tk canvas and the text widget allow 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 $wish++$ 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
}
```

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.

Example An example of hypertext file as may be used by a *help* manager is given below.

```
if you don't wish to read the following text,
click on the %%
button $this.goto -text button -command "$this gotoline $EOT"
$this append $this.goto

            ...help text ...
%%
global EOT
set EOT [expr $thisline-1]
%%
Press %%
button $this.quit -command {destroy .} -text quit
$this append $this.quit
%% to remove the window
```

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.

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

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 $wish++$ library offers the class var.

```
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 ();
}
```

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.)

Example The function inc, shown below, 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.

```
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;
}
```

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

```
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;
}
```

The function *program* 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.

```
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");
```

No doubt, the reader will need to experiment to get a feel for the interaction between C++ functions and Tcl scripts.

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

To complete the example, the function *main* using *install* and *program* is pictured above.

9 Creating new widgets

By virtue of the class structure and inheritance mechanism of C++ new widgets may easily be created. An example is provided by the *filechooser* widget provided by *wish++*. The *filechooser* widget merely groups an entry, button and listbox in an appropriate way and allows the user to install a handler to retrieve the selected filename.

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

The filechooser has been implemented by a mixture of Tcl and C++ code. In order to make the *filechooser* also available as a widget to be used in Tcl scripts some additional work is required. Recall that when creating a widget in Tcl/Tk, the pathname of the widget defines a command that may be used to perform operations (such as *configure*) on the widget, in an object-like manner.

Dispatching For a widget (not already defined by Tk) to act as a command it must be able to figure as a *handler* object, for which the function *dispatch* is invoked whenever it is treated as a command. The *dispatch* function for the *filechooser* class is shown below.

```
private:
filechooser* f;

int dispatch(kit* tk, int argc, char** argv){
     f = this;
     if (!strcmp("configure", * argv))
          return options(tk, argc, ++argv);
     else if(!strcmp("get",*argv)){ f →get(); return OK; }
     else cout << "no such option" << endl;
     argv++;
     }
     return OK;
}
```

In other words, the class *filechooser* combines three roles, namely in the first place that of a widget class defining the functionality of a widget, secondly it implements the command *filechooser* that is used to create a widget, and thirdly it embodies the command associated with (the pathname of) the widget.

For the *dispatch* function there is an ambiguity with respect to what object is referenced. When creating a new widget, the object referenced must still be created. However, if the widget already exists it will reference itself when treated as a command. To resolve this ambiguity, an additional pointer *f* is introduced that will reference the object itself unless the reference is changed by invoking *create*.

```
int create(kit* tk, int argc, char** argv) {
     f = new filechooser(* argv);
     tk →action(*argv,f);
     tk →result(*argv);
     return options(tk, –argc, ++argv);
}
```

The function *create* creates a new (filechooser) widget and associates the pathname of the widget with an action taking the newly created widget as handler object. Then (in conformance with Tk conventions) it sets the result of the action to the pathname of the newly created widget.

Example The *filechooser* widget command may now be used in a same way as the standard Tk widget commands.

```
filechooser .ff -command {puts stdout [.ff get] }
pack append . .ff {top}
go-back
```

In order to make the *filechooser* command available as a Tcl command, an instance of *filechooser* must be created to be associated with the command of that name as a handler object.

```
void install(kit* tk, int, char**){
    handler* h = new filechooser;
    tk →action("filechooser",h);
}

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

In the function *main* above, no program function is given to *run*. This restricts the executable to be useful for interpreting scripts only. When after interpreting a script, also a program function must be executed, the script must contain the command $go - back$ as its last command.

10 Conclusion

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

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

This approach has a great extent been inspired by the InterViews library. However, although the class structure is reminiscent of the InterViews library class structure, it is considerably less complex. In comparison with InterViews, the *wish++* library provides more ready-to-use graphical interface widgets, yet offers no resolution-independent graphics and provides no pre-defined classes for complex interaction such as *Observer* class provided by InterViews.

References

[Joh90] Ousterhout John. Tcl : An embeddable command language. In *Inproceedings of the winter 1990 USENIX Conference*, pages 133–146, Berkeley (CA), USA, January 1990. Usenix Association.

[Joh91] Ousterhout John. An x11 toolkit based on tcl language. In *Proceedings of Winter 1991 USENIX Conference*, pages 105–115, Berkely (CA),USA, January 1991. Usenix Association.

[Joh93] Ousterhout John. Hypertext and hypermedia. *Usenix Conference*, October 1993.

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