

# 1

## Application development



After studying general issues in the design and software engineering of object-oriented applications and frameworks, it is time to focus in somewhat more detail on actual application development. In this chapter we will look at the *drawtool* application, as a representative of a broader category of interactive editing tools.

### Application development

4

1-1

- the drawtool applications
- guidelines for design
- from specification to implementation

Additional keywords and phrases: *hush framework, interactive editors, law of Demeter, formal specification in Z, abstract systems*

Slide 1-1: Application development

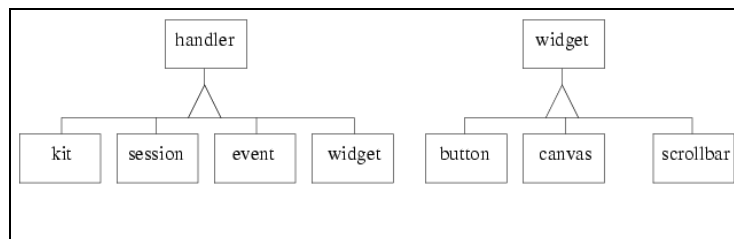
The *drawtool* application is a Java application using the multiparadigm *hush* framework. However, in discussing its development, we will concentrate on specifying the requirements and issues of design. After that we will treat some miscellaneous issues in the design of classes. This chapter will be concluded with a case study, a concise, yet detailed, example of a more formal approach to the development of an object-oriented application.

## 1.1 The *drawtool* application

Interactive editors are an interesting category of applications. Interactive editors, which include word processors and drawing tools, are the kind of applications the average (end) user is most familiar with. From a software engineering perspective, interactive editors are interesting because they combine interactive and functional features. See also [GOF94], which provides many patterns for interactive editors. In the Software Engineering curriculum at the Vrije Universiteit, we have repeatedly used interactive editors as a medium-term assignment for CS2 students (five weeks for groups of four or five students). One example of such an assignment is the *Interactive Design Assistant* discussed in section ???. Another example is the musical score editor (see appendix ??), which has been chosen by a selected group of CS3 and CS4 students as a practical assignment for the Object-Oriented Software Development course. In this section we will look at the *drawtool* application, which is a representative realization of a (rather simple) drawing editor. The implementation of *drawtool* presented here is realized in the Java version of the *hush* framework. The *hush* C++ framework has been used for a number of years in the Software Engineering curriculum, but has recently been replaced by Java with Swing. The *drawtool* application is nevertheless interesting because it acted for many years as the basic example of an interactive editor for quite a number of students. Before studying *drawtool*, we will first look at the realization of a drawing canvas in *hush*.

### A simple drawing canvas in *hush*

The Tcl/Tk toolkit provides a very powerful scripting environment for realizing graphical user interfaces, [Ousterhout91]. The *hush* Java/C++ library gives convenient access to the Tcl/Tk toolkit in an object-oriented style. See also [HUSH].

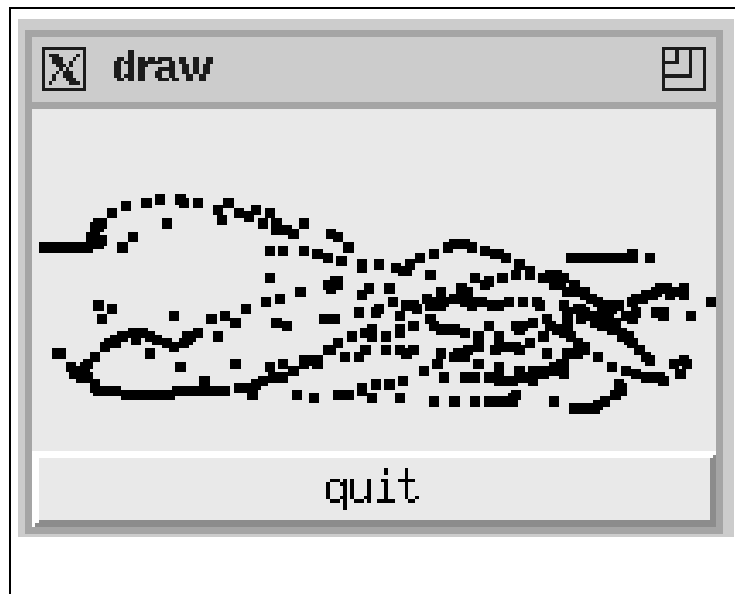


1-2

Slide 1-2: The *hush* class hierarchy

The *hush* library provides three kinds of classes, namely (a) the widget classes, which mimic the functionality of Tk, (b) the handler and event classes, which are involved in the handling of events and the binding of Java/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 on the right in slide ??, the *widget* class 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 Java or C++ terms. It may be used for creating references to widgets defined in Tcl. In contrast, employing the constructor of one of the concrete widget classes results in actually creating a widget.



1-3

Slide 1-3: Drawing canvas

Widgets are the elements from which a GUI is made. 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. The interface of the *widget* class may be defined by the (pseudo) interface below.

```
public interface widget {
    public String path();
    public void eval(String cmd);
    public void pack(String s);
    public void bind(handler h,String s);
    public void bind(String p, handler h,String s);
    public void configure(String cmd);
    public void geometry(int x, int y);
    public void xscroll(widget w);
```

*widget*

```

public void yscroll(widget w);

public widget self(); // to define compound widgets
public void redirect(widget inner);
};

```

The function *path* delivers the path name 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 its background color by a Tcl command like

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

The function *eval* enables 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. As an example look at the code for the drawing canvas widget depicted in slide ??.

```

import hush.dv.api.event;
import hush.dv.widgets.canvas;

class draw extends canvas {
    boolean dragging;

    public draw(String path) {
        super(path);
        dragging = false;
        bind(this);
    }

    public void press(event ev) {
        dragging = true;
    }

    public void release(event ev) {
        dragging = false;
    }

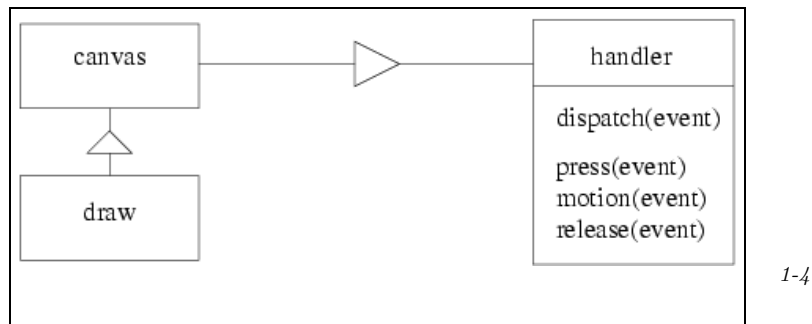
    public void motion(event ev) {
        if (dragging)
            circle(ev.x(),ev.y(),2,"-fill black");
    }
};

```

*draw*

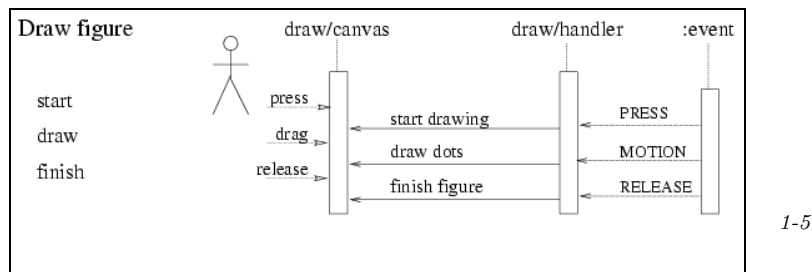
The class *draw* has an instance variable *dragging*, that reflects whether the

user is actually drawing a figure. If *dragging* is true, motions with the mouse will result in small dots on the screen.



Slide 1-4: Drawing canvas

A structural view of the *draw* class is given in slide ???. The *draw* class is derived from a *canvas*, which is itself (indirectly) derived from a *handler* class. The *handler* class dispatches events to predefined handler methods, such as *press*, *motion* and *release*. For the *draw* class we must distinguish between a *handler* and a *canvas* part. The *handler* part is defined by the methods *press*, *release* and *motion*. The *canvas* part allows for drawing figures, such as a small circle.

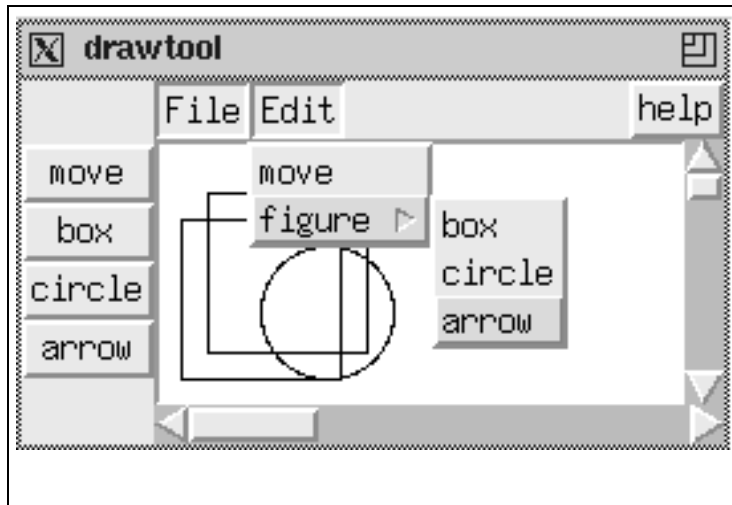


Slide 1-5: Drawing canvas

In slide ?? it is depicted how these two parts interact when the user draws a figure. Actions of the user result in events that activate the handler. Note that the UML sequence diagrams are not completely adequate here, since it is difficult to express information concerning the events and the state of the *draw* instance. Widgets may respond to events. To associate an event with an action, an explicit binding must be specified for that particular widget. Some widgets provide default bindings. These may, however, be overruled. The function *bind* is used to associate handlers with events. The first string parameter of *bind* may be used to specify the event type. Common event types are, for example, *ButtonPress*, *ButtonRelease* and *Motion*, which are the default events for canvas widgets. Also keystrokes may be defined as events, for example *Return*, which is the default event for the *entry* widget.

The function `bind(handler, String)` may be used to associate a handler object with the default bindings for the widget. Concrete widgets may not override the `bind` function itself, but must define the protected function `install`. Typically, the `install` function consists of calls to `bind` for each of the event types that is relevant to the widget. In addition, the widget class offers two functions that may be used when defining compound or mega widgets. The function `redirect(w)` must be used to delegate the invocation of the `eval`, `configure` and `bind` functions to the widget `w`. The function `self()` gives access to the widget to which the commands are redirected. The function `path` will still deliver the path name of the outer widget. Calling `redirect` when creating the compound widget class suffices for most situations. However, when the default events must be changed or the declaration of a handler must take effect for several component widgets, the function `install` must be redefined to handle the delegation explicitly.

### The *drawtool* application



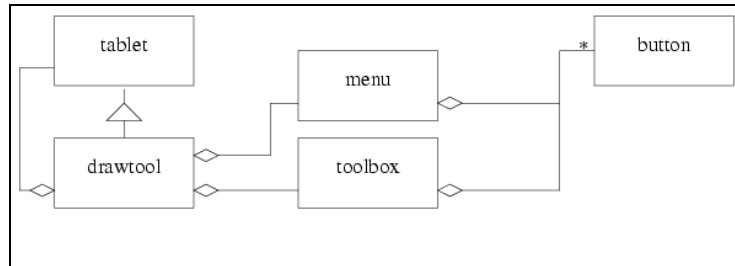
1-6

Slide 1-6: The *drawtool* interface

In this section we will look at the realization of simple drawing tool. The example illustrates how to use the *hush* library widgets, and serves to illustrate in particular

how to construct compound widgets. A structural view of the *drawtool* application is given in slide ??.

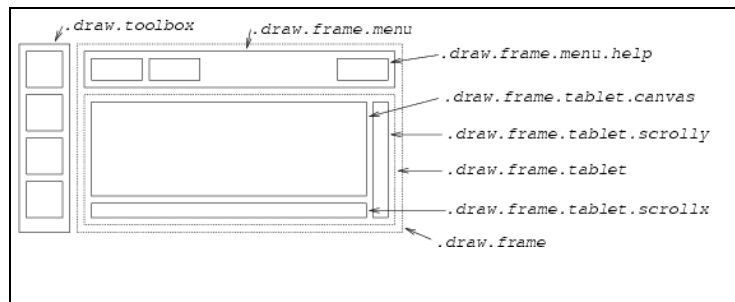
Usually, the various widgets constituting the user interface are (hierarchically) related to each other, such as in the *drawtool* application which contains a canvas to display graphic elements, a button toolbox for selecting the graphic items and a menubar offering various options such as saving the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



1-7

Slide 1-7: A (partial) class diagram

Widgets in Tk are identified by a *path name*. The path name of a widget reflects its possible subordination to another widget. See slide ??.



1-8

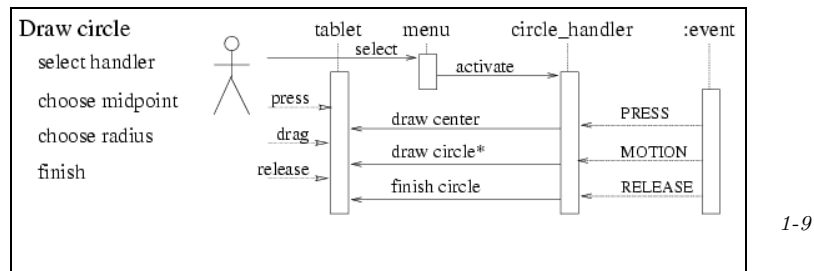
Slide 1-8: Widget containment

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

$$\text{path} ::= '.' \mid '.'\text{string} \mid \text{path}'.\text{string}$$

For example `'.'` is the path name of the root widget, whereas `'..quit'` is the path name of a widget subordinate to the root widget. A widget subordinate to another widget must have the path name of that widget as part of its own path name. For example, the widget `'..f.m'` may have a widget `'..f.m.h'` as a subordinate widget. Note that the widget hierarchy induced by the path names is completely orthogonal to the widget class inheritance hierarchy. With respect to the path name hierarchy, when speaking of ancestors we simply mean superordinate widgets.

Our drawing tool consists of a *tablet*, which is a canvas with scrollbars to allow for a large size canvas of which only a part is displayed, a *menubar*, having a *File* and an *Edit* menu, and a *toolbox*, which is a collection of buttons for selecting from among the drawing facilities. In addition, a help facility is offered.



Slide 1-9: An interaction diagram

A typical interaction (or *use case*) with *drawtool* is depicted in slide ???. On selecting the *circle* menu entry (or toolbox button), the *circle handler* is activated to assist in the drawing of a circle. Details will be given when discussing the *tablet* widget.

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

```
import hush.dv.api.*;
import hush.dv.widgets.frame;
```

```
public class toolbox extends frame {
```

*toolbox*

```
    tablet tablet;
```

```
    public toolbox(widget w, tablet t) {
        super(w,"toolbox");
        tablet = t;
        new toolbutton(this,"draw");
        new toolbutton(this,"move");
        new toolbutton(this,"box");
        new toolbutton(this,"circle");
        new toolbutton(this,"arrow");
    }
```

```
    public int operator() {
        tablet.mode(_event.arg(1)); // reset tablet mode
        return OK;
    }
```

```
};
```

Each button is an instance of the class *toolbutton*.

```
import hush.dv.api.*;
```

```

import hush.dv.widgets.button;

public class toolbutton extends button {
public toolbutton(widget w, String name) {
    super(w,name);
    text(name);
    bind(w,name);
    pack("-side top -fill both -expand 1");
}
};

```

*toolbutton*

When a *toolbutton* is created, the actual button is given the name of the button as its path. Next, the button is given the name as its text, the 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 path name, 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. The constructor further creates the five toolbuttons.

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

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 upon what is considered the most convenient way in which to access the resources needed. As a guideline, exploit the regularity of the application.

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

```

import hush.dv.api.widget;

public class menubar extends hush.dv.widgets.menubar {

public menubar(widget w, tablet t, toolbox b) {
    super(w,"bar");
    configure("-relief sunken");

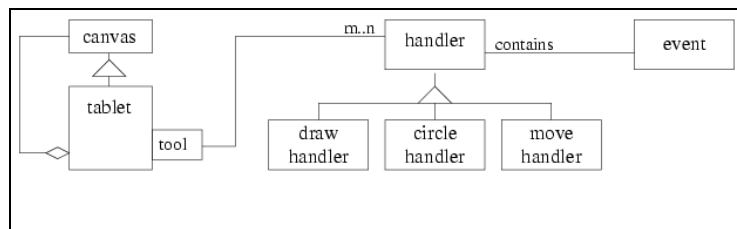
    new FileMenu(this,t);
    new EditMenu(this,b);
    new HelpButton(this);
}
};

```

*menubar*

The class *menubar*, given above, is derived from the *hush* widget *menubar*. Its constructor requires an ancestor widget, a *tablet* and a *toolbox*. The *tablet* is passed as a parameter to the *file\_menu*, and the *toolbox* to the *edit\_menu*. In addition, a *help\_button* is created, which provides online help in a hypertext format when pressed. 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 second button of the *menubar* is defined by the *EditMenu*. The *EditMenu* 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 *EditMenu* coincide.



1-10

Slide 1-10: Tablet

**The *tablet* component** The most important component of our *drawtool* application is defined by the *tablet* widget class given below.

```
import hush.dv.api.*;
import hush.dv.widgets.*;
```

```
public class tablet extends canvas {
```

<i>tablet</i>
---------------

```
    int _mode;
    canvas canvas;
    handler[] handlers;
```

```
    final int DRAW = 0;
    final int MOVE = 1;
    final int CIRCLE = 2;
    final int BOX = 3;
    final int ARROW = 5;
```

```
    public tablet(widget w, String name, String options) {
```

```
        super(w,name,"");
```

```

        handlers = new handler[12];

        init(options);
        redirect(canvas); // to delegate to canvas
        bind(this); // to intercept user actions

        handlers[DRAW] = new DrawHandler(canvas);
        handlers[MOVE] = new MoveHandler(canvas);
        handlers[BOX] = new BoxHandler(canvas);
        handlers[CIRCLE] = new CircleHandler(canvas);
        handlers[ARROW] = new ArrowHandler(canvas);

        _mode = 0; // drawmode.draw;
    }

    public int operator() {
        handlers[_mode].dispatch(_event);
        return OK;
    }

    public int mode(String s) {
        int m = -1;
        if ("draw".equals(s)) m = DRAW;
        if ("move".equals(s)) m = MOVE;
        if ("box".equals(s)) m = BOX;
        if ("circle".equals(s)) m = CIRCLE;
        if ("arrow".equals(s)) m = ARROW;
        if (m != 0) _mode = m;
        return _mode;
    }

    void init(String options) {

        widget root = new frame(path(),"-class tablet");

        canvas = new canvas(root,"canvas",options);
        canvas.configure("-relief sunken -background white");
        canvas.geometry(200,100);

        scrollbar scrollx = new Scrollbar(root,"scrollx");
        scrollx.orient("horizontal");
        scrollx.pack("-side bottom -fill x -expand 0");

        scrollbar scrolly = new Scrollbar(root,"scrolly");

        scrolly.orient("vertical");
        scrolly.pack("-side right -fill y -expand 0");
    }

```

```

        canvas.pack("-side top -fill both -expand 1");

        canvas.xscroll(scrollx); scrollx.xview(canvas);
        canvas.yscroll(scrolly); scrolly.yview(canvas);
    }

};

```

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

Although the *tablet* must act as a canvas, the actual *tablet* widget is nothing but a *frame* that contains a canvas widget as one of its components. This is reflected in the invocation of the canvas constructor (*super*). By convention, when the options parameter is \* 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 right away, the *tablet* constructor creates a top frame, initializes the actual component widgets, and redirects the *eval*, *configure* and *bind* invocations to the subordinate *canvas* widget. It then binds itself to be its own handler, which results in binding itself to be the handler for the canvas component. Note that reversing the order of calling *redirect* and *bind* 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.

**The *drawtool* class** Having taken care of the basic components of the drawing tool, that is the *toolbox*, *menubar* and *tablet* widgets, all that remains to be done is to define a suitable *file\_handler*, appropriate handlers for the various drawing modes and a *help\_handler*.

We will skip these, but look at the definition of the *drawtool* class instead. In particular, 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.

```

import hush.dv.api.*;
import hush.dv.widgets.frame;

```

```

import hush.dv.widgets.canvas;

public class drawtool extends canvas {
    widget root;
    tablet tablet;

    public drawtool() { System.out.println("meta handler created"); }

    public drawtool(String p, String options) {
        super(p,"*");          // create empty tablet
        init(options);
    }

    public int operator() {
        System.out.println("Calling drawtool:" + _event.args(0) );
        String[] argv = _event.argv();
        if ("self".equals(argv[1])) tk.result(self().path());
        else if ("drawtool".equals(argv[0]))
            create(argv[1],_event.args(2));
        else if ("path".equals(argv[1])) tk.result(path());
        else if ("pack".equals(argv[1])) pack(_event.args(2));
        else self().eval( _event.args(1) ); // send through
        return OK;
    }

    void create(String name, String options) {
        drawtool m = new drawtool(name,options);
    }

    void init(String options) {
        root = new frame(path(),"-class Meta");

        frame frame = new frame(root,"frame");

        tablet = new tablet(frame,"tablet",options);

        toolbox toolbox = new toolbox(frame,tablet);
        menubar menubar = new menubar(root,tablet,toolbox);

        toolbox.pack("-side left -fill y -expand 0");
        tablet.pack("-side left -fill both -expand 1");

        menubar.pack();
        frame.pack("-expand 1 -fill both");

        redirect( tablet ); // the widget of interest
    }
}

```

*drawtool*

```
    }
};
```

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()* 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.

The functionality offered by the interpreter defined by the *operator()* function in (II) is kept quite 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*'s 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 *eval* is invoked for *self()*, with the remaining arguments flattened to a string. This allows for using the *drawtool* almost as an ordinary canvas.

```
Canvas c = new DrawTool("draw","");
tk.bind("drawtool",c);
c.circle(20,20,20,"-fill red");
c.rectangle(30,30,70,70,"-fill blue");
c.pack();
```

In the program fragment above, the Tcl command *drawtool* is declared, with an instance of *drawtool* as its handler. (It is assumed that the *tk* variable refers to an instance of *kit*.) In this way, the *drawtool* widget is made available as a command when the program is used as an interpreter. In this case, the actual *drawtool* widget is made the handler of the command, to allow for a script to address the *drawtool* by calling *drawtool self*.

## 1.2 Guidelines for design

Computing is a relatively young discipline. Despite its short history, a number of styles and schools promoting a particular style have emerged. However, in contrast to other disciplines such as the fine arts (including architecture) and musical composition, there is no well-established tradition of what is to be considered as *good taste* with respect to software design. There is an on-going and somewhat pointless debate as to whether software design must be looked at as an *art* or must be promoted into a *science*. See, for example, [Knuth92] and [Gries]. The debate has certainly resulted in new technology but has not, I am afraid, resulted in universally valid design guidelines.

The notion of *good design* in the other disciplines is usually implicitly defined by a collection of examples of good design, as preserved in museums or (art or

music) historical works. For software design, we are still a long way from anything like a museum, setting the standards of good design. Nevertheless, a compendium of examples of object-oriented applications such as [Pinson90] and [Harmon93], if perhaps not setting the standards for good design, may certainly be instructive.

<p><b>Development process – <i>cognitive factors</i></b></p> <ul style="list-style-type: none"><li>• model → realize → refine</li></ul> <p><b>Design criteria – <i>natural, flexible, reusable</i></b></p> <ul style="list-style-type: none"><li>• abstraction – <i>types</i></li><li>• modularity – <i>strong cohesion</i> (class)</li><li>• structure – <i>subtyping</i></li><li>• information hiding – <i>narrow interfaces</i></li><li>• complexity – <i>weak coupling</i></li></ul>	1-11
--	------

Slide 1-11: Criteria for design

The software engineering literature abounds with advice and tools to measure the quality of good design. In slide ??, a number of the criteria commonly found in software engineering texts is listed. In software design, we evidently strive for a high level of abstraction (as enabled by a notion of types and a corresponding notion of *contracts*), a modular structure with strongly cohesive units (as supported by the class construct), with units interrelated in a precisely defined way (for instance by a client/server or subtype relation). Other desirable properties are a high degree of information hiding (that is narrowly defined and yet complete interfaces) and a low level of complexity (which may be achieved with units that have only weak coupling, as supported by the client/server model). An impressive list, indeed.

Design is a human process, in which *cognitive factors* play a critical role. The role of cognitive factors is reflected in the so-called *fractal design process model* introduced in [JF88], which describes object-oriented development as a triangle with bases labeled by the phrases *model*, *realize* and *refine*. This triangle may be iterated at each of the bases, and so on. The iterative view of software development does justice to the importance of human understanding, since it allows for a simultaneous understanding of the problem domain and the mechanisms needed to model the domain and the system architecture.

Good design involves taste. My personal definition of good design would certainly also involve cognitive factors (*is the design understandable?*), including subjective criteria such as *is it pleasant to read or study the design?*

### 1.2.1 Individual class design

A class should represent a faithful model of a single concept, and be a reusable, plug-compatible component that is robust, well-designed and extensible. In slide

??, we list a number of suggestions put forward by [McGregor92].

**Class design – guidelines**

- only methods public – *information hiding*
- do not expose implementation details
- public members available to all classes – *strong cohesion*
- as few dependencies as possible – *weak coupling*
- explicit information passing
- root class should be abstract model – *abstraction*

1-12

Slide 1-12: Individual class design

The first two guidelines enforce the principle of *information hiding*, advising that only methods should be public and all implementation details should be hidden. The third guideline states a principle of *strong cohesion* by requiring that classes implement a single protocol that is valid for all potential clients. A principle of *weak coupling* is enforced by requiring a class to have as few dependencies as possible, and to employ explicit information passing using messages instead of inheritance (except when inheritance may be used in a type consistent fashion). When using inheritance, the root class should be an abstract model of its derived classes, whether inheritance is used to realize a partial type or to define a specialization in a conceptual hierarchy.

The properties of classes, including their interfaces and relations with other classes, must be laid down in the design document. Ideally, the design document should present a complete and formal description of the structural, functional and dynamic aspects of the system, including an argument showing that the various models are consistent. However, in practice this will seldom be realized, partly because object-oriented design techniques are as yet not sufficiently matured to allow a completely formal treatment, and partly because most designers will be satisfied with a non-formal rendering of the architecture of their system. Admittedly, the task of designing is already sufficiently complex, even without the additional complexity of a completely formal treatment. Nevertheless, studying the formal underpinnings of object-oriented modeling based on types and polymorphism is still worthwhile, since it will sharpen the intuition with respect to the notion of behavioral conformance and the refinement of contracts, which are both essential for developing reliable object models. And reliability is the key to reuse!

### 1.2.2 Inheritance and invariance

When developing complex systems or class libraries, reliability is of critical importance. As shown in section ??, assertions provide a means by which to check the runtime consistency of objects. In particular, assertions may be used to check that the requirements for behavioral conformance of derived classes are met.