VBTkit Reference Manual
A toolkit for Trestle

Edited by Marc H. Brown and James R. Meehan

January 4, 1993

Contributors

The VBTkit software reflects the efforts of many people at SRC. Development of the software began in early 1988, and new pieces are still being developed. The major contributors, in alphabetical order, are as follows:

- **Andrew Birrell** implemented `ListVBT` and preliminary versions of radio buttons and check boxes.
- **Ken Brooks** implemented early versions of the text-related modules, the `Rsrc` interface, and some of the utility routines.
- **Marc H. Brown** implemented the 3-D look and feel (Section 3.1), the multi paradigm (Section 4), the buttons (Section 5), subwindows (Section 6), and some of the utility routines.
- **Mark R. Brown** implemented the Ivy text editor. This isn't part of VBTkit *per se*, but the user interface of VBTkit's text editing facility is based on Ivy. Some of the prose within this document is lifted from the Ivy manual.
- **Pat Chan** implemented an earlier version of `ScrollerVBT`. Pat also implemented Ivy with Mark Brown.
- **Luca Cardelli** implemented a Trestle toolkit named "vbtkit" in 1987. That software was retired in 1991 as part of SRC's infamous "Project '90". However, some snippets of Luca's code do live in the current VBTkit. Perhaps the most noticeable is `ZGrowVBT`'s user interface and algorithm for reshaping a subwindow. Luca was graciously "willing to transfer my proprietary vbtkit(TM)(SM)(C)(R) logo for a small monthly fee. . . . OK, a small yearly fee."
- **John DeTreville** implemented the `VText` package, which is the heart of the VBTkit's text-editing facilities.
- **Steve Glassman** implemented `JoinVBT`, `ScaleFilter`, and `ViewportVBT`. Steve also implemented the tools to read ASCII descriptions of pixmaps and cursors from files. Most importantly, Steve has discovered and fixed numerous infelicities in a variety of interfaces.
- **Mark Manasse** significantly helped implement `ReactivityVBT` and `JoinVBT`. He was a valuable sounding board in the development of the multi-filter and multi-split paradigms.
- **Jim Meehan** ported the text modules (Section 2) from Modula-2 to Modula-3, and greatly enhanced and updated them.
- **Greg Nelson** helped implement `JoinVBT`.
- **Jorge Stolfi** implemented all of the color modules (Section 10).

- **Mary-Claire van Leunen** wrote the Ivy manual, from which words were lifted to describe VBTkit's text-editing facilities.

Contents

1 Introduction

This document is a programmer's reference manual for VBTkit[1], a Modula-3 user interface toolkit based on the Trestle window system toolkit. VBTkit provides a library of "widgets" and the support software that makes it easy to customize these widgets and to construct more widgets.

This reference manual is not self-contained; we assume you are already familiar with Trestle. If you are not, you should read the *Trestle Tutorial* [3] before continuing. You are advised to have a copy of the *Trestle Reference Manual* [2] in hand as you read this. This reference manual (like the *Trestle Reference Manual*) consists of complete Modula-3 interfaces printed in `typewriter font` and interspersed with commentary printed in Roman font.

1.1 VBTkit widgets versus Trestle widgets

There are three key differences between the widgets in VBTkit and those in Trestle. The differences concern the dimensions in which screen-independent sizes are specified, the visual appearances of widgets, and the use of multiple VBTs to form a single widget.

[1]Not to be confused with the "vbtkit" software developed by Luca Cardelli in 1987. At SRC, we have been calling the software described in this manual "lego" (not to be confused with the popular children's toy).

VBTkit widgets	Trestle widgets
Screen-independent sizes are specified in points. VBTkit uses 72 points to the inch[2], and 25.4 millimeters to the inch. The `Pts` interface provides routines to convert between millimeters and points, and between points and screen pixels.	Sizes are specified in millimeters.
Most widgets can appear with a Motif-like, three-dimensional look.	All of the built-in Trestle widgets appear two-dimensional.
Colors are specified with a `Shadow.T` object, a subtype of `PaintOp.ColorScheme` that has painting operations for the shadows that are used with background and foreground colors. A `Shadow.T` also includes the size of the shadow, and the `Shadow.Style` type enumerates the styles of shadows that can be drawn, such as raised or recessed.	Widgets are given background and foreground colors by specifying a screen-independent painting operation, a `PaintOp.T` or a `PaintOp.ColorScheme`, an object that encapsulates a variety of painting operations formed using the background and foreground colors.
Many widgets are built from a collection of `VBT`s, whose internal structure is hidden from the client.	Widgets are all primitive. That is, in Trestle a `VBT` is either a *leaf* or a *split*. A leaf provides "interactors", like scrollbars and typing fields, and a split provides geometric layout for an arbitrary number of children. A *filter* is a split that has a single child; filters modify a child in some ways, such as adding a border or interpreting mouse clicks.

In VBTkit, a composite VBT has some number of *logical* children. These logical children are typically descendants (but not necessarily children) in the VBT hierarchy. For example, a `ViewportVBT` puts scrollbars around a child VBT. If the child's minimum acceptable size is larger than the domain it has been given, the scrollbars are used to see parts of the child. Logically, a `ViewportVBT` has a single child. Internally, a `ViewportVBT` comprises nearly a dozen VBTs, including two `ScrollerVBT`s, three `HVSplit`s, one `JoinVBT`, and two `FlexVBT`s. The `Multi` interface is used to access the logical children of a composite VBT, and the `MultiClass` interface is used for building composite VBTs.

A simpler type of composite VBT that appears frequently is one with no logical children. For example, a `NumericVBT` contains a place to type numbers, and buttons to increment and decrement the number. Internally, this is implemented with eleven VBTs. Logically, however, a client views a `NumericVBT` as a "leaf"; asking for its internal children is discouraged, as it would violate the abstraction that a `NumericVBT` gives.

A final difference between VBTkit widgets and Trestle widgets concerns "callback" procedures that are invoked in response to user action.

VBTkit widgets	Trestle widgets
Callback procedures are invoked in response to *different* types of user actions. For example, the `NumericVBT` callback procedure is invoked when the number is changed—either by clicking with the mouse on the plus or minus buttons, or by typing a number followed by a carriage return. The `AnyEvent` interface is used to encapsulate different types of Trestle events into a single parameter.	A callback procedure is are invoked in response to a *single* type of user action (e.g., a mouse click), passing the event-record that describes the user gesture (e.g., a `VBT.MouseRec`).

1.2 Roadmap

Section 2 describes the text-editing facilities in VBTkit. The primary VBT class is a `TextPort`. It provides a full-screen editor, with the most common commands bound to keys. It is rare to use a `TextPort` directly, however. Typically, one uses a `TextEditVBT`—a `TextPort` with a scrollbar. A popular subclass of `TextEditVBT` is a `TypescriptVBT`. Section 2 also describes a number of interfaces that are of interest to programmers implementing subclasses of text-editors.

Section 3 describes the facilities for giving a "three-dimensional" look. The basic data types are defined in the `Shadow` interface. A `ShadowedVBT` displays a 3-D border around a VBT, and a `ShadowedBarVBT` displays a horizontal or vertical line in 3-D.

Section 4 describes the ways that one can build a VBTkit widget out of multiple internal VBTs. The `Multi` interface allows a client to access the logical children of a composite VBT, and the `MultiClass` interface is needed to implement new composite VBTs. Composite VBTs are at the heart of the buttons that VBTkit provides; you are advised to study the `Multi` interface before reading about VBTkit buttons.

Section 5 describes VBTkit buttons. There are two fundamental differences between Trestle buttons and VBTkit buttons. First, in VBTkit the user-gesture that makes up a button (e.g., does it react on an up-click or a down-click?) is separated from the visual feedback (e.g., when the button is activated, does it change its colors?). The user gesture is defined by a VBT, and the feedback is defined by another VBT. These VBTs are composed using the `Multi` interface. (It is not unusual for a feedback VBT to be a composite VBT itself.) Second, in VBTkit the client action procedure, invoked when the user activates a button, is a method on the VBT subclass. In Trestle, the callback is a procedure that is specified when a button subclass is instantiated. To ensure that there is no confusion, we call VBTkit-style buttons *switches*. VBTkit also adds considerably to the collection of buttons provided by Trestle. The switches provided by VBTkit includes guarded switches (they require a double click to be activated), trill switches (they continue to invoke the callback while the mouse button is down), Boolean check boxes, radio choices, and "drag and drop" with semantic feedback.

Section 6 describes support for subwindows. A subwindow is like a top-level window but it is not installed in the window manager. The bad news is that user gestures for controlling the subwindow (moving, reshaping, closing) are not the same as for top-level windows. The good news is that the user interface is quite obvious, and the subwindows are automatically controlled by the top-level window. For example, when the top level window is iconized, all of its subwindows also disappear (without cluttering up the window manager's icon box). Most of the VBT classes that support subwindows are multis.

Section 7 describes various useful leaf VBTs. (Internally, these VBTs may be composed of many VBTs; however, they do not have any logical children.) These include `ListVBT`, a text browser that displays a scrollable list of items, typically strings; `FileBrowserVBT`, a file browser that displays the files in a directory and allows the user to traverse the directory hierarchy; `PixmapVBT`, a VBT that displays a pixmap; `ScrollerVBT`, a scrollbar; and two widgets for specifying an integer in a given range (a `NumericVBT` is a type-in field with increment and decrement buttons on its sides, and a `NumericalScrollerVBT` is a scrollbar).

Section 8 describes various useful non-leaf `VBT`s. A `FlexVBT` is a filter that places size constraints on its child. You can give an explicit size (just like Trestle's `RigidVBT`), or add some stretch to the child or some shrinkage to the child's preferred size. A `ReactivityVBT` is a filter that can make its child unresponsive to mouse and keyboard activity, optionally also "graying-out" the child. A `SplitterVBT` is like an `HVSplit`, but automatically puts adjusting bars (`HVBars`) between each child. Because a `SplitterVBT` is a multi, the client never deals with the `HVBars` (unless it accesses the children of the `SplitterVBT` using the `Split` interface rather than the `Multi` interface, of course). A `ViewportVBT` is a multi with a single logical child. It puts scrollbars around the child so the entire contents of the child can be viewed, even if the `ViewportVBT` is allocated less screen space than the child has specified that it needs.

Section 9 contains the various utility interfaces that the VBTkit widgets use internally. A couple of the interfaces are needed by clients of certain VBTkit widgets. For example, the `AnyEvent` interface encapsulates different types of Trestle events into a single parameter. It is needed by clients of `NumericVBT` and `FileBrowserVBT`, since those widgets allow different types of user gestures (e.g., a double-click and a carriage return) to invoke the same callback method. The `FlexShape` interface is used in conjunction with a `FlexVBT`. You may need to use routines in `Pts` to convert between points (as used by VBTkit widgets) and millimeters (used by Trestle's widgets). You should use routines in `PaintOpCache` rather than `PaintOp`, to make your application more efficient and peppier. You should probably skim the other utility interfaces, but you probably won't need to use them until you implement a new widget.

Section 10 describes some of the utilities for manipulating color. Most noteworthy is the `ColorName` interface that takes a description like "SomewhatMurkyYellow" and returns the red, green, and blue components that will display the color. This section also defines mappings between various common color models like RGB, HSV, and CIE.

Section **??** provides functions for an application to locate the resources it needs at runtime. These resources are stored in files, and they include cursors, pixmaps, and text files.

2 Text Editing

2.1 The TextPort Interface

A `TextPort` is a `VBT` that allows the user to type and edit single- or multi-line text.

The `TextPortClass` interface defines “models” for keybindings and text-selections; the `TextPort` interface itself supports four models: Ivy, Emacs, Xterm, and Mac. The default depends on the environment variable `TextPortModel`, which defaults to `ivy`. See the `IvyModel`, `EmacsModel`, `XtermModel`, and `MacModel` interfaces in Appendices A.1–A.4 for details on keybindings, mouse clicks, and selections.

`TextPort` uses the terms *primary* and *secondary* for selections; the model may support only one, and it defines the correspondence between the selection-name (e.g., `primary`) with Trestle selections (e.g., `VBT.Target`).

The locking level for all procedures is `LL <= VBT.mu` except as noted.

```
INTERFACE TextPort;

IMPORT Font, PaintOp, ScrollerVBT, VBT, VText;

TYPE
  Points = REAL;
  Model = {Default, Ivy, Emacs, Mac, Xterm};
  SpecificModel = [Model.Ivy .. Model.Xterm];
  T <: Public;
  Public =
    VBT.Leaf OBJECT
    METHODS
      init (singleLine                            := FALSE;
            hMargin, vMargin: Points              := 1.5;
            font                                  := Font.BuiltIn;
            colorScheme     : PaintOp.ColorScheme := NIL;
            expandOnDemand                        := FALSE;
            wrap                                  := TRUE;
            readOnly                              := FALSE;
            turnMargin      : Points              := 2.0;
            model                                 := Model.Default ): T;

      <* LL = VBT.mu *>
      getFont        (): Font.T;
      setFont        (font: Font.T);
      getColorScheme (): PaintOp.ColorScheme;
      setColorScheme (c: PaintOp.ColorScheme);
      getModel       (): SpecificModel;
      setModel       (model: Model);
```

```
      getReadOnly     (): BOOLEAN;
      setReadOnly     (flag: BOOLEAN);
      scrollUpdate    ();

      (* callbacks *)
      returnAction  (READONLY cd: VBT.KeyRec);
      tabAction     (READONLY cd: VBT.KeyRec);
      defaultAction (READONLY cd: VBT.KeyRec);
      focus         (gaining: BOOLEAN; time: VBT.TimeStamp);
      filter        (VAR cd: VBT.KeyRec);
      modified      ();
      error         (msg: TEXT);

    END;
```

The call `v.init(...)` initializes `v` as a `TextPort` and returns it.

If `singleLine` is `TRUE`, then the `TextPort`'s height is set to the height of the tallest character in the current font, and it does not expand vertically. Its default width is 30 times the width of the widest character in the current font. This is intended for "type-in" boxes.

The parameters `hMargin` and `vMargin` indicate how much whitespace to leave around the text.

If `colorScheme` is `NIL`, then `PaintOp.bgFg` will be used.

If `expandOnDemand` is `TRUE`, then multi-line and scrollable `TextPort`s will expand (and contract) vertically as the text requires, so that the entire text is visible in the window. This parameter is ignored for single-line or nonscrollable `TextPort`s.

If `wrap` is `TRUE` and `singleLine` is `FALSE`, then text will be wrapped across line boundaries; otherwise it will be clipped. If it is wrapped, then `turnMargin` specifies the width of the gray bar placed at the end of the first line and the beginning of the second, indicating that the text has been wrapped.

If `readOnly` is `TRUE`, then the text cannot be changed through the user interface (keyboard and mouse). The procedures `Replace`, `Insert`, `SetText`, and `PutText` bypass the read-only protection, but these are not called by internal routines. In all other descriptions in this interface, the words "replace", "insert", "delete", and so on should be understood as having the restriction that `v` is not read-only.

If `model` is `Model.Default`, then `init` will use the current value of `DefaultModel`, defined below.

`v.getModel()` returns the name of the current model; note that the return value cannot be `Model.Default`. The call `v.setModel(...)` changes the current model; its parameter may be `Model.Default`, in which case the value of `DefaultModel` will be used.

The implementation calls `v.scrollUpdate` whenever `v`'s state has changed in a way that might change the associated scrollbar, if any. The default method calls `ScrollerVBT.Update` with the `TextPort`'s scrollbar, if it has one.

The call `v.setFont(font)` changes the font used for displaying the text.

The call `v.setColorScheme(c)` changes the colors used for displaying the text.

The implementation calls `v.focus(gaining, ...)` whenever `v` gains or loses the keyboard focus. If `gaining` is `TRUE`, then `v` is about to gain the keyboard focus; i.e., this method is called *before* the selection feedback is established, so it is reasonable to call `Select` (below) or put up some other indication. If `gaining` is `FALSE`, then `v` has just lost the keyboard focus, so it reasonable to take down whatever indicated that the focus had been acquired. It is not within the power of the `focus` method to prevent `v` from gaining or losing the focus. The default for this method is a no-op.

The implementation calls `v.error(msg)` whenever an exception is raised for which there is no particular remedy, such as an `Rd.Failure`. The value of `msg` will be a short description of the error, typically the name of the procedure where the exception was raised. No method or procedure defined in this interface raises exceptions, but the client may wish to report the error or take some other action, such as beeping or flashing the screen. The default for this method is a procedure that tests whether the environment-variable named `TextPortDebug` is set (to any value); if so, it writes the message to `SmallIO.stderr`.

2.1.1 Keybindings

The `TextPort` interface allows clients a great deal of flexibility in handling keystrokes. `TextPort.T.key(v, cd)` proceeds in four steps. In step 1, it tests whether `cd.wentDown` is true, whether `v` has the keyboard focus, and whether `v`'s domain is non-empty. If all three conditions are true, it proceeds to step 2.

In step 2, it calls `v.filter(cd)`, passing `cd` as a `VAR` parameter. The default method for `filter` is a no-op, but clients may override this. In addition to calling other procedures in the `TextPort` interface, a `filter` method may:

- leave `cd` untouched,
- translate it into some other keystroke, by changing `cd.whatChanged` and/or `cd.modifiers`, or
- suppress it entirely by setting `cd.whatChanged` to `VBT.NoKey`.

If the client's `filter` method handles a key directly, by taking some action, then it should usually set `cd.whatChanged` to `VBT.NoKey`, which prevents further processing of the key.

In step 3, we call the model's filter, which follows the same conventions as the `TextPort`'s filter. (The EmacsModel, for example, has a filter that implements quoted-insert.)

In step 4, there are several mutually exclusive possibilities, tested in this order:

- If the key is Return, then if the `shift` bit is on, we insert a newline; if the `option` bit is on, we insert a newline but leave the cursor in place; otherwise, we invoke `v.returnAction(cd)`, another callback method. Its default method calls `v.defaultAction(cd)` if this is a "single-line" `TextPort`; otherwise it calls `NewlineAndIndent(v, cd)`.

- If the key is Tab, we invoke `v.tabAction(cd)`. This is a callback that the client can override. The default method inserts 4 spaces.
- If the `control` bit is on (i.e., if

  ```
  VBT.Modifier.Control IN cd.modifiers
  ```

  we call the model's `controlChord` method.
- If the `option` bit is on, we call the model's `optionChord` method.
- If the key is an "arrow" key, we call the model's `arrowKey` method.
- If the key is Backspace or Delete, we delete the previous character.
- If it's a graphic character, we insert it into the text.
- Otherwise, we invoke `v.defaultAction(cd)`, a callback. Its default is a no-op.

Finally, we call `Normalize(v)`, except in the `controlChord`, `optionChord`, and `defaultAction` cases.

Clients can specialize the handling of keys, therefore, by overriding the `TextPort`'s `key`, `filter`, `returnAction`, `tabAction`, or `defaultAction` methods, and by overriding the model's `controlChord`, `optionChord`, or `arrowKey` methods.

2.1.2 Other state information

A `TextPort` maintains some internal information that is accessed indirectly:

- a "modified" flag. Any operation that changes the text sets this flag to `TRUE`. If it was previously `FALSE`, then the implementation calls `v.modified()`, **after** the change has already happened to `v`. The default is a no-op. The `SetModified` and `IsModified` procedures set and test this flag, respectively.
- a scrollbar (optional). See the `TextEditVBT` interface in Section 2.3.

2.2 Interface routines

```
VAR DefaultModel: SpecificModel;
```

The initial value of `DefaultModel` *is determined by the environment-variable named* `TextPortModel`*; if that is set to* `ivy`, `emacs`, `mac`, *or* `xterm`, *then* `DefaultModel` *will be set accordingly. If it is not defined, or is defined as some other value, then the initial value of* `DefaultModel` *will be* `Model.Ivy`.

```
PROCEDURE ChangeAllTextPorts (newModel := Model.Default);
```

For each `TextPort p`, *call* `p.setModel(newModel)`. *(The implementation maintains a list of weak references to all initialized* `TextPort`*s.) The principal client is* `FormsVBT`*; see the* `MForm` *interface.*

```
TYPE Scroller <: ScrollerVBT.T;
```

A scrollbar that is to be used with a `TextPort` *must be one of these.*

```
PROCEDURE SetScrollBar (v: T; scrollBar: Scroller);
```

Set the scrollbar associated with `v`.

```
PROCEDURE MarkAndUpdate (v: T);
```

Call `VBT.Mark(v)` *and then update the scrollbar. This is like* `Normalize`*, except that no scrolling is actually performed.*

```
PROCEDURE SetWrap (v: T; wrap: BOOLEAN);
```

Set the "wrap" property of `v`*, and redisplay* `v`.

```
PROCEDURE Length (v: T): CARDINAL;
```

Return the number of characters in `v`*'s text.*

```
PROCEDURE Width (v: T): CARDINAL;
```

Return the number of characters that will fit on a line, given the current size of `v`*. If* `v` *has no width (because its window is iconic, for example), return 0. If* `v`*'s font is not fixed-pitch, the result will be computed in terms of the width of the widest character in the font.*

```
PROCEDURE Height (v: T): CARDINAL;
```

Equivalent to `v.shape(Axis.T.Ver, 0)`.

```
PROCEDURE ShapeInfo (v: T; VAR lineCount, lineLength: INTEGER);
```

Return the number of lines in the text and the number of characters in the longest line (excluding the newline). A client may use this information in conjunction with `VBTShape.Set` *to shape* `v` *to fit its text. Having done so, it should be prepared to do it again on a rescreen to a screen of different density.*

```
PROCEDURE Index (v: T): CARDINAL;
```

Return the current "type-in" position.

```
PROCEDURE Seek (v: T; n: CARDINAL);
```

Set the "type-in" position.

```
PROCEDURE Newline (v: T);
```

Insert a newline character at the type-in point.

```
PROCEDURE NewlineAndIndent (v: T);
```

Insert a newline character and enough spaces to match the indentation of the previous line. As it leaves a blank line, it will delete all spaces from that line so as to leave it truly empty.

2.2.1 Editing commands

```
PROCEDURE GetText (v    : T;
                   begin: CARDINAL := 0;
                   end  : CARDINAL := LAST (CARDINAL)): TEXT;
<* LL = VBT.mu *>
```

Returns a sub-sequence of the text in `v`. *The result will be empty if* `begin >= Length(v)`*; otherwise the range of indexes of the subsequence is*

```
[begin .. MIN (end, Length (v)) - 1]
```

Note that the following four procedures do not test the `readOnly` status of `v`.

```
PROCEDURE SetText (v: T; t: TEXT); <* LL <= VBT.mu *>
```

Replace the current contents of `v` *with* `t`.

```
PROCEDURE PutText (v: T; t: TEXT); <* LL <= VBT.mu *>
```

Append `t` *to the current contents of* `v`.

```
PROCEDURE Replace (v: T; begin, end: CARDINAL; newText: TEXT);
```

Replace the text between positions `begin` *and* `end` *in* `v` *with* `newText`. *If* `begin` *and* `end` *are beyond the end of the text, they are taken to refer to the end of the text.*

```
PROCEDURE Insert (v: T; text: TEXT);
```

If there is a replace-mode selection (see Section A.1.3, page 106), replace it with `text`*; otherwise insert* `text` *at the type-in point. In either case, this is a no-op if* `text` *is the empty string.*

2.2.2 Selection

`TextPort` defines two text-selections, referred to as *primary* and *secondary*. The `TextPort`'s model determines whether these types are supported, and if so, how they correspond to `VBT` selections.

In the Ivy model, for example, primary and secondary correspond to `VBT.Target` and `VBT.Source`, respectively. User actions that establish the primary selection cause `VBT.Target` to be acquired; likewise for the secondary selection and `VBT.Source`.

In the Emacs, Mac, and Xterm models, only primary selection is supported, and it corresponds to `VBT.Source`. Operations involving secondary selections are effectively no-ops.

```
TYPE SelectionType = {Primary, Secondary};
```

A selection is represented by a pair of indexes (`begin` and `end`) into the text. A selection may or may not correspond to a `VBT.Selection`; that is determined by the model. The current selection-indices can be retrieved via the `GetSelection` procedure.

```
PROCEDURE Select (v         : T;
                  time      : VBT.TimeStamp;
                  begin, end: CARDINAL;
                  sel         := SelectionType.Primary;
                  replaceMode := FALSE;
                  caretEnd    := VText.WhichEnd.Right   );
```

Make a selection in `v`, at event-time `time`. If `begin` and/or `end` are beyond the end of the text, they will be clipped to the end of the text. Acquire ownership of the corresponding `VBT.Selection`, and, in case of `SelectionType.Primary`, of the keyboard focus as well.

The values `replaceMode` and `caretEnd` are relevant only if `sel` is `SelectionType.Primary`. If `replaceMode` is `TRUE`, then `Insert` and `VBT.Write` will *replace* the selected text; otherwise, they cause the new text to be *inserted* at whichever end of the primary selection is specified by `caretEnd`.

```
PROCEDURE IsReplaceMode (v: T): BOOLEAN;
```

Return `TRUE` *if the primary selection is in "replace mode".*

```
PROCEDURE HasSelection (v: T; sel := SelectionType.Primary):
  BOOLEAN; <* LL.sup = VBT.mu *>
```

Return `TRUE` *if* `v` *currently owns the* `VBT.Selection` *corresponding to* `sel`.

```
TYPE Extent = RECORD l, r: CARDINAL END;

CONST NotFound = Extent {LAST (CARDINAL), LAST (CARDINAL)};

PROCEDURE GetSelection (v: T; sel := SelectionType.Primary): Ex-
tent;
```

Return the extent of the most recent selection in `v`. *If there is no such selection, return* `NotFound`.

```
PROCEDURE GetSelectedText (v: T; sel := SelectionType.Primary):
  TEXT;                             <* LL.sup = VBT.mu *>
```

Return the text of the most recent selection in `v` *if there is one, or the empty string otherwise.*

```
PROCEDURE PutSelectedText (v   : T;
                           t   : TEXT;
                           sel          := SelectionType.Primary);
<* LL.sup = VBT.mu *>
```

Replace the text of the most recent selection in `v`*, if there is one, with* `t`*. If there is no such selection, this is a no-op.*

```
PROCEDURE TryFocus (v: T; t: VBT.TimeStamp): BOOLEAN;
```

Try to acquire the keyboard focus and the primary selection, and report whether it succeeded.

```
PROCEDURE HasFocus (v: T): BOOLEAN; <* LL.sup = VBT.mu *>
```

Test whether `v` *has the keyboard focus.*

```
PROCEDURE Normalize (v: T; to := -1);
```

Scroll `v` *if necessary to ensure that position* `to` *is visible. If* `to < 0`*, it refers to the current type-in point. If* `to` *is larger than the length of the text, normalizes to the end of the text.*

```
PROCEDURE IsVisible (v: T; pos: CARDINAL): BOOLEAN;
```

Test whether character position `pos` *is visible.*

```
PROCEDURE IsModified (v: T): BOOLEAN;
```

Return the value of the "modified" flag for `v`*. Any change to the text will cause the flag to be set to* `TRUE`*.*

```
PROCEDURE SetModified (v: T; value: BOOLEAN);
```

Set the value of the "modified" flag for `v`*. This will not invoke* `v.modified`*, even if* `value` *is* `TRUE`*.*

```
PROCEDURE GetVText (v: T): VText.T;
```

For wizards only: extract the underlying `vtext`*. It is legal to create and manipulate highlighting intervals on it. It is legal to run readers on it, provided you can be sure that you are locking out concurrent change (for example, by holding* `VBT.mu`*). It is not legal to modify it directly. It is not legal to scroll it directly either, because that will leave the scrollbar incorrect.*

```
END TextPort.
```

2.3 The TextEditVBT Interface

A `TextEditVBT` combines a `TextPort` with a scrollbar. It is a `ShadowedVBT`, which is a filter; if the `TextEditVBT` is scrollable, then the filter's child is an `HVSplit` whose children are a scrollbar, a solid line, and a `TextPort`. Otherwise, the filter's child is just the `TextPort`. The `TextPort` handles all the editing; see Section 2.1. If you need neither a scrollbar nor a shadow, you should probably use a `TextPort` instead of a `TextEditVBT`.

The scrollbar is implemented by `ScrollerVBT`, and it uses `ScrollerVBT`'s default parameters for the time to delay before starting the auto-repeat, and for the period of auto-repeating. (A client can change this—but only globally—by using `ScrollerVBT.SetScrollingTimes`.)

```
INTERFACE TextEditVBT;

IMPORT Font, PaintOp, Shadow, ShadowedVBT, TextPort;

TYPE Kind = {SingleLine, MultiLine, Scrollable};
```

`TextEditVBT`s *can be single-line editors (*`SingleLine`*), nonscrollable text areas (*`MultiLine`*), or scrollable text areas (*`Scrollable`*).*

```
TYPE
  T <: Public;
  Public = ShadowedVBT.T BRANDED OBJECT
             port: TextPort.T := NIL; (* READONLY *)
           METHODS
             <* LL = VBT.mu *>
             init (kind  := Kind.Scrollable;
                   style := Shadow.Style.Flat;
                   scrollbar     : TextPort.Scroller := NIL;
                   shadow        : Shadow.T          := NIL;
                   expandOnDemand                    := FALSE): T;
             getFont        (): Font.T;
             setFont        (font: Font.T);
             getColorScheme (): PaintOp.ColorScheme;
             setColorScheme (c: PaintOp.ColorScheme);
           END;
```

The call `v.init(...)` initializes `v` as a `TextEditVBT` and returns `v`. If `v.port` is `NIL`, a `TextPort` will be created and initialized from `kind`, `shadow`, and `expandOnDemand` and assigned to `v.port`. The default value used for `shadow` is `Shadow.None`.

If `kind` is `Kind.Scrollable` and `v.port` is not a single-line textport, a scrollbar will be added to the left of the `TextPort`. If no scrollbar is supplied, a default one will be created.

After initialization, the client should not change the value of `v.port`.

```
PROCEDURE GetPort (v: T): TextPort.T;
```

Return `v.port`*.*

```
END TextEditVBT.
```

2.4 The TypescriptVBT Interface

A `TypescriptVBT` is a subtype of `TextEditVBT`, with additional features to make it serve as a "glass teletype" with a memory. To the client, it provides a reader to get input from the user, and a writer to print output. The reader is intermittent and unseekable; the writer is buffered and unseekable. The writer is paired with the reader, such that the writer is flushed whenever a seek blocks on the reader. The writer is not flushed at every newline.

All input to the typescript, once it has been read, and all output, become part of the "history" of the typescript, and is not modifiable; it remains until the user deletes it by calling `ClearHistory`. Selections that lie fully or partially within the history region are never "replace-mode" selections (see Section A.1.3, page 106). Any attempt to type or insert text in the history region becomes an insertion at the end of the typescript instead.

A typescript's textport, `v.port`, must be of type `TypescriptVBT.Port` (which is a subtype of the type `TextPort.T`). The type `TypescriptVBT.Port` overrides `TextPort.T.returnAction`: it makes the text of the current type-in region available to the reader and no longer editable.

The client may, in general, use all the `TextEditVBT` procedures on a `TypescriptVBT`, but should not call

```
v.port.setReadOnly (FALSE)
```

Typescripts do not allow the use of Undo and Redo.

```
INTERFACE TypescriptVBT;

IMPORT Font, PaintOp, Rd, Shadow, TextEditVBT, TextPort,
       Thread, VBT, Wr;

TYPE
  T <: Public;
  Public = TextEditVBT.T OBJECT
           METHODS
             <* LL.sup <= VBT.mu or 0 if v is disconnected *>
             init (shadow: Shadow.T := NIL;
                   style            := Shadow.Style.Flat): T;
             interrupt       (time: VBT.TimeStamp);
             handleInterrupt (time: VBT.TimeStamp);
             setThread       (thread: Thread.T := NIL);
             terminate       ();
           END;
```

The call `v.init(...)` initializes `v` as an empty typescript. The parameter `shadow` defaults to `Shadow.None`.

If `v.port` is `NIL` when `v.init(...)` is called, then a default `TextPort` will be created and assigned to `v.port`. It is a checked runtime error if `v.port` is not of type `TypescriptVBT.Port`, which is a subtype of `TextPort.T`.

The call `v.interrupt(time)` simulates an interrupt by flushing pending type-in, writing the characters `^C`, and then calling `v.handleInterrupt(time)`.

The call `v.handleInterrupt(time)` alerts the thread that is currently waiting on input from `v`'s reader, or if none, the thread that most recently waited. If no thread has ever read from `v`'s reader, this call has no effect.

The call `v.setThread(thread)` sets the thread that is to be alerted on subsequent calls to `v.interrupt(time)`. (So does reading.) `thread` defaults to `Thread.Self()`.

The call `v.terminate()` causes `v` to report end of file on its input stream. This is appropriate when `v` is being thrown away. The call causes `v` to become unresponsive to further user input, but it continues to display output written to its output stream.

```
TYPE
  Points = REAL;
  Port <: PublicPort;
  PublicPort =
    TextPort.T OBJECT
    METHODS
      init (hMargin, vMargin: Points := 1.5;
            font                     := Font.BuiltIn;
            colorScheme: PaintOp.ColorScheme := NIL;
            wrap                             := TRUE;
            readOnly                         := FALSE;
            turnMargin : Points              := 2.0    ): Port;
               END;

  Reader <: PublicReader;
  PublicReader = Rd.T OBJECT METHODS typescript (): T END;

  Writer <: PublicWriter;
  PublicWriter = Wr.T OBJECT METHODS typescript (): T END;

PROCEDURE GetRd (v: T): Reader;
```

Get the input stream for `v`. *Always returns the same reader for a given* `v`.

```
PROCEDURE GetWr (v: T): Writer;
```

Get the output stream for `v`. *Always returns the same writer for a given* `v`.

```
PROCEDURE GetHistory (v: T): TEXT; <* LL <= VBT.mu *>
```

Return the "history" text of `v`.

```
PROCEDURE ClearHistory (v: T);  <*  LL <= VBT.mu *>
```

Clear the "history" text of `v`.

```
END TypescriptVBT.
```

2.5 Subclasses of text-editors: The TextPortClass interface

The `TextPortClass` interface contains the information needed to implement a variety of models for keybinding and text-selection. Four models are currently implemented: Ivy, Emacs, Mac, and Xterm. `TextPort.Model` is an enumeration type for the four names, but `TextPortClass.Model` is the type of the actual object attached to a `TextPort`, to which user-events (keys, mouse clicks, position reports) are sent. The model is responsible for all highlighting.

Unless otherwise noted, the locking level of all procedures in this interface is `TextPort.T.mu`.

```
INTERFACE TextPortClass;

IMPORT Font, KeyboardKey, PaintOp, TextPort, VBT, VTDef, VText;

CONST Primary = TextPort.SelectionType.Primary;
      Secondary = TextPort.SelectionType.Secondary;

REVEAL TextPort.T <: T;

TYPE
  Pixels = CARDINAL;
  T = TextPort.Public OBJECT
        m : Model    := NIL;
        mu: MUTEX;                (* VBT.mu < v.mu *)
        <* LL = v.mu *>
        vtext         : VText.T;
        font          : Font.T;
        scrollBar     : TextPort.Scroller;
        singleLine    : BOOLEAN;
        readOnly      : BOOLEAN;
        expandOnDemand: BOOLEAN;
        visible       : BOOLEAN;
        linesShown    : CARDINAL;
        charWidth     : Pixels             := 0;
        fontHeight    : Pixels             := 0;
        typeinStart   : CARDINAL;
        cur           : UndoRec;
        lastCmdKind   : CommandKind;
        thisCmdKind   : CommandKind;
        wishCol       : CARDINAL;
        hasFocus                           := FALSE;
      METHODS
        <* LL = SELF.mu *>
        getText       (begin, end: CARDINAL): TEXT;
        index         (): CARDINAL;
        length        (): CARDINAL;
```

```
    normalize     (to := -1);
    replace       (begin, end: CARDINAL; newText: TEXT): TextPort.Extent;
    unsafeReplace (begin, end: CARDINAL; newText: TEXT): TextPort.Extent;
    insert        (t: TEXT);
    unsafeInsert  (t: TEXT);
    getKFocus     (time: VBT.TimeStamp): BOOLEAN;
    newlineAndIndent ();

    (* All of these call SELF.error. *)
    vbterror   (msg: TEXT; ec: VBT.ErrorCode);
    vterror    (msg: TEXT; ec: VTDef.ErrorCode);
    rdfailure  (msg: TEXT; ec: REFANY);
    rdeoferror (msg: TEXT);

    (* We release SELF.mu around the following callbacks. *)
    ULreturnAction  (READONLY cd: VBT.KeyRec);
    ULtabAction     (READONLY cd: VBT.KeyRec);
    ULdefaultAction (READONLY cd: VBT.KeyRec);
    ULfocus         (gaining: BOOLEAN; time: VBT.TimeStamp);
    ULmodified      ();
    ULerror         (msg: TEXT);

  END;
```

`v.font` is a local copy of `v.vtext.vOptions.vFontxxx.vFont.font`. `v.charwidth` is the width of a (maximal) character.

`v.fontHeight` is the height of a (maximal) character.

`v.typeinStart` is meaningful only for typescripts, where it indicates the point that divides the "history" part of the transcript, which is read-only, from the current command line, which is not. See the `TypescriptVBT` interface. For non-typescripts, this field is always zero.

`v.thisCmdKind` and `v.lastCmdKind` allow the interpretation of a command to depend on the previous command. Currently, the only commands that depend on context are the "vertical" commands that call `UpOneLine` and `DownOneLine`.

`v.hasFocus` is `TRUE` when `v` has acquired the keyboard focus. The keyboard focus and primary selection are always acquired and released together.

`v.replace` tests `v.readOnly`; if that is `TRUE`, then it returns `TextPort.NotFound`. Otherwise it calls `v.unsafeReplace`, which is the only routine that actually alters the underlying text.

`v.insert` calls `v.replace`; i.e., it is safe.

```
TYPE CommandKind = {VertCommand, OtherCommand};

TYPE
  Arrow = [KeyboardKey.Left .. KeyboardKey.Down];
  Model <: PublicModel;
  PublicModel =
```

```
OBJECT
  v        : T;
  fixed    : CARDINAL;         (* Fixed pivot for highlighting *)
  dragging               := FALSE;
METHODS
  <* LL = SELF.v.mu *>
  init  (cs: PaintOp.ColorScheme): Model;
  close ();

  mouse    (READONLY cd: VBT.MouseRec);
  position (READONLY cd: VBT.PositionRec);
  misc     (READONLY cd: VBT.MiscRec);
  read (READONLY s: VBT.Selection; time: VBT.TimeStamp): TEXT
        RAISES {VBT.Error};
  write (READONLY s: VBT.Selection; time: VBT.TimeStamp; t: TEXT)
         RAISES {VBT.Error};

  cut             (time: VBT.TimeStamp);
  copy            (time: VBT.TimeStamp);
  paste           (time: VBT.TimeStamp);
  clear           ();

  filter          (VAR cd: VBT.KeyRec);
  controlChord    (ch: CHAR; READONLY cd: VBT.KeyRec);
  optionChord     (ch: CHAR; READONLY cd: VBT.KeyRec);
  arrowKey        (ch: Arrow; READONLY cd: VBT.KeyRec);

  getSelection    (sel := Primary): TextPort.Extent;
  getSelectedText (sel := Primary): TEXT;
  putSelectedText (t: TEXT; sel := Primary);
  hasVBTselection (sel := Primary): BOOLEAN;
  seek            (position: CARDINAL);
  select (time       : VBT.TimeStamp;
          begin, end : CARDINAL;
          sel                        := Primary;
          replaceMode                := FALSE;
          caretEnd                   := VText.WhichEnd.Right);
  isReplaceMode (): BOOLEAN;
  takeSelection (sel      : TextPort.SelectionType;
                 time     : VBT.TimeStamp;
                 highlight                           := FALSE): BOOLEAN;
  walkIntervals (p: IProc) RAISES {VTDef.Error};
END;
```

`m.init(...)` initializes a Model `m`. The default method is a no-op and simply returns `m`.

`m.close()` releases `VBT` selections and deletes highlighting intervals.

The type `TextPort.T` overrides the `VBT` `mouse`, `position`, `misc`, `read`, and `write` methods with procedures that lock `v.mu` and call `v.m.mouse`, `v.m.position`, etc. Note that the signatures are not identical to their Trestle counterparts.

Clients must override the `read` method with a procedure that returns a text if `m` owns the selection `s`; otherwise it should call the default method, which calls `VBT.Read(s, time)`. `time` is valid when the caller is a user-event procedure such as `Paste`; it will be 0 when called from `TextPort.T.read`, but in that case, `m` owns the selection, so `time` is not needed.

Similarly, clients must override the `write` method. `write` is called by `TextPort.T.write`, which ensures that `m.v.readOnly` is `FALSE` before calling this.

If there is a primary selection, then `m.copy(time)` arranges for that text to become the source selection. Otherwise, it is a no-op; in particular, if the selection is empty, it will not acquire the source selection.

The default for `m.cut(time)` is `m.copy(time); m.clear()`.

The default for `m.paste(time)` is `m.insert(m.read(VBT.Source, time))`.

`m.clear()` deletes the primary selection. Its default method is

```
m.putSelectedText ("", TextPort.SelectionType.Primary)
```

`m.insert(t)` implements `TextPort.Insert`. The default method replaces the primary selection, if there is one, with `t`; otherwise, it inserts `t` at the type-in point. Clients may wish to override this in order to alter the highlighting.

`m.filter(cd)` is called by the default method for `TextPort.T.key`. The default for `filter` is a no-op.

The `controlChord`, `optionChord`, and `arrowKey` methods are described in the `TextPort` interface. The interfaces for the specific models override `controlChord` and `optionChord`, for which there are no defaults. The default method for `arrowKey` dispatches to `ToPrevChar`, `ToNextChar`, `UpOneLine`, and `DownOneLine`.

`m.seek(position)` sets the type-in point.

```
TYPE IProc = PROCEDURE (i: VText.Interval) RAISES {VTDef.Error};

TYPE
  SelectionRecord = RECORD
                      owned                         := FALSE;
                      interval: VText.Interval;
                      mode    : VText.SelectionMode
                    END;

TYPE UndoRec <: ROOT;

TYPE IRange = RECORD left, middle, right: CARDINAL END;

PROCEDURE GetRange (         v   : T;
                    READONLY cp  : VBT.CursorPosition;
                             mode: VText.SelectionMode ): IRange;
  <* LL = v.mu *>
```

Return an `IRange` *indicating the boundaries of the character, word, paragraph, etc., that contains the position* `cp`*. The* `middle` *field will be equal to either the* `left` *field or the* `right` *field, depending on which end the cursor was closer to.*

```
CONST
  VBTErrorCodeTexts =
    ARRAY VBT.ErrorCode OF TEXT {
     "event not current", "timeout", "uninstalled", "unreadable",
     "unwritable", "unowned selection", "wrong type"};
```

2.5.1 Cursor-motion

```
PROCEDURE ToPrevChar (v: T);
PROCEDURE ToNextChar (v: T);
```

Move cursor (type-in point, caret) left or right one char.

```
PROCEDURE ToStartOfLine (v: T);
PROCEDURE ToEndOfLine (v: T);
```

Move cursor to start or end of line.

```
PROCEDURE ToOtherEnd (v: T);
```

Move cursor to other end of the primary selection.

```
PROCEDURE FindNextWord (v: T): TextPort.Extent;
```

Locate the “next” word: from the current position, we scan right until we reach an alphanumeric character. Then we continue scanning right until we reach the first non-alphanumeric character; that position defines the right end of the extent. Then we scan left until we find a non-alphanumeric character. That position, plus 1, defines the left end of the extent.

If the initial position is in the middle of a word, then the extent actually covers the *current* word, but on successive calls, it covers each following word in turn.

```
PROCEDURE FindPrevWord (v: T): TextPort.Extent;
```

Locate the “previous” word. This is the same as `ToNextWord`*, except that all the scanning directions are reversed.*

```
PROCEDURE UpOneLine (v: T);
PROCEDURE DownOneLine(v: T);
```

Move cursor up or down one line.

2.5.2 Deletion commands

All these procedures return an `Extent` indicating the range of characters that were deleted, or `TextPort.NotFound` if no characters were deleted.

```
PROCEDURE DeletePrevChar (v: T): TextPort.Extent;
PROCEDURE DeleteNextChar (v: T): TextPort.Extent;
```

```
PROCEDURE DeleteToEndOfWord (v: T): TextPort.Extent;
```

Delete from the current position to the end of the "next" word (as defined in `ToNextWord`*).*

```
PROCEDURE DeleteToStartOfWord (v: T): TextPort.Extent;
```

Delete from the current position to the start of the "previous" word (as defined in `ToPrevWord`*).*

```
PROCEDURE DeleteToStartOfLine (v: T): TextPort.Extent;
```

Delete to start of line, or delete the preceding newline if the cursor is at the start of line.

```
PROCEDURE DeleteToEndOfLine (v: T): TextPort.Extent;
```

Delete to end of line. If cursor is at the end, delete the newline.

```
PROCEDURE DeleteCurrentWord (v: T): TextPort.Extent;
```

Delete the word containing the cursor.

```
PROCEDURE DeleteCurrentLine (v: T): TextPort.Extent;
```

Delete line containing the cursor.

2.5.3 Other modification commands

```
PROCEDURE SwapChars(v: T);
```

Swap the two characters to the left of the cursor.

```
PROCEDURE InsertNewline(v: T);
```

Insert a newline without moving the cursor.

2.5.4 Searching

```
TYPE
  Direction = {Forward, Backward};
  Where = {Here, Beginning};

PROCEDURE Find (v         : T;
```

```
pattern   : TEXT;
direction          := Direction.Forward;
fromWhere          := Where.Here;
ignoreCase         := TRUE              ): TextPort.Extent;
```

Search for a pattern. The search proceeds either forward or backward (`direction`) from either the current location or the beginning of the buffer (`fromWhere`). If `ignoreCase` is `TRUE`, the case of letters is not significant in the search.

2.5.5 Scrolling the display

```
PROCEDURE ScrollOneLineUp (v: T);
PROCEDURE ScrollOneLineDown (v: T);
PROCEDURE ScrollOneScreenUp (v: T);
PROCEDURE ScrollOneScreenDown (v: T);
```

Move the displayed text up or down by either a line or screen. This doesn't move the selections or cursor, so the `TextPort` may not be normalized when done. A "screen" contains `MAX(1, n-2)` lines, where `n` is the number of diplayed lines.

2.5.6 Managing the "Undo" stack

The "Undo" stack records all the editing changes made to the `TextPort`. These changes can be undone; once undone, they can be redone. There is no built-in limit to the number of changes that are recorded. A sequence of insertions of graphic characters (i.e., plain typing) counts as one "edit".

```
PROCEDURE AddToUndo (v: T; begin, end: CARDINAL; newText: TEXT);
  <* LL = v.mu *>
```

This is called by `v.unsafeReplace(begin, end, newText)` *to record a change to the underlying text.*

```
PROCEDURE Undo (v: T); <* LL = v.mu *>
```

Reverse the effect of the last editing command.

```
PROCEDURE Redo (v: T); <* LL = v.mu *>
```

Reinstate the effect of the last editing command.

```
PROCEDURE ResetUndo (v: T); <* LL < v.mu *>
```

Clear the "Undo" stack. (Nothing in the implementation calls this procedure.)

```
PROCEDURE UndoCount (v: T): CARDINAL; <* LL < v.mu *>
```

Return the number of changes that can be undone.

```
PROCEDURE RedoCount (v: T): CARDINAL; <* LL < v.mu *>
```

Return the number of undone changes that can be redone.

```
END TextPortClass.
```

2.6 Other interfaces

The VBTkit package also provides a number of interfaces that are geared to help clients define subclasses of text-editors. Here are the interfaces of interest:

- `Key` defines constants for the `VBT.KeySym`s of some common non-graphic keys.
- `KeyTrans` provides some standard mapping between keyboard keys and ASCII characters.
- `MTextUnit` implements tools for treating the underlying text as sequences of characters, lines, or paragraphs.

3 Shadows: 3-dimensional look and feel

3.1 The Shadow Interface

The `Shadow` interface contains the basic definitions for VBT classes that implement a Motif-like, 3-D look.

A *shadow* is like a border, but it gives the visual illusion of "raising" a rectangle above the background, "lowering" a rectangle into the background, drawing a "ridge" above the background, or chisling a "groove" into the background.

These visual effects are actually quite simple to accomplish by drawing parts of the border using a dark variant of the background color, and by drawing other parts using a light variant of the background color.

For example, to give the impression that an object is raised above its background, the north and west borders re drawn using a light color, whereas the south and east border are drawn in a dark color. To draw a "ridge", the north and west shadows start out in the light color, and, halfway, switch to the dark color. Analogously, the south and east shadows start out dark and switch to a light color.

For maximum effectiveness, the child's background should be a color whose saturation level is about 50%, and the light and dark shadows should colors with the same hue and lightness, but with saturation levels of 25% and 75% respectively.

The following chart summarizes how the visual effects are accomplished:

Style	*North/West*	*South/East*
Flat	Background	Background
Raised	Light	Dark
Lowered	Dark	Light
Ridged	Light/Dark	Dark/Light
Chiseled	Dark/Light	Light/Dark

See Figure 3.1.

```
INTERFACE Shadow;

IMPORT PaintOp, VBT;

TYPE
  Points = REAL;
  T = PaintOp.ColorScheme OBJECT
        size : Points;
        light: PaintOp.T;
        dark : PaintOp.T;
      END;
```

Shadows will occupy `ABS(size)` points. A negative size is interpreted by some `VBT` classes as a signal to give feedback in a 2-D style. For instance, when they are activated, `SwitchVBT`s display an inverted image, rather than a recessed shadow.

```
TYPE Style = {Flat, Raised, Lowered, Ridged, Chiseled};

PROCEDURE New (size : Points := 2.0;
              bg            := PaintOp.Bg;
              fg            := PaintOp.Fg;
              light         := PaintOp.Fg;
              dark          := PaintOp.Fg  ): T;
```

Return a newly allocated `Shadow.T` with the specified fields. All of the paint ops should be tints, arranged so that on a monochrome screen `bg` draws in background, while `fg`, `light`, and `dark` draw in foreground.

```
PROCEDURE SetSize  (shadow: T; newSize: Points): T;
PROCEDURE SetBg    (shadow: T; newBg:    PaintOp.T): T;
PROCEDURE SetFg    (shadow: T; newFg:    PaintOp.T): T;
PROCEDURE SetLight (shadow: T; newLight: PaintOp.T): T;
PROCEDURE SetDark  (shadow: T; newDark:  PaintOp.T): T;
```

Return a newly allocated copy of `shadow`, with the specified field replaced with a new value.

```
PROCEDURE Supported (shadow: T; v: VBT.T): BOOLEAN;
```

Return whether `shadow` will appear 3-dimensional on `v`. Two conditions must hold: `v` must be on a color screen and `shadow.size` must be positive.

```
VAR (* CONST *) None: T;
```

This variable is really a constant for

`New(0.0, PaintOp.Bg, PaintOp.Fg, PaintOp.Fg, PaintOp.Fg)`

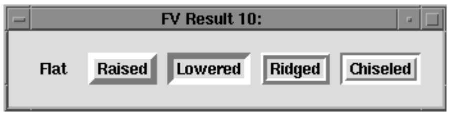

Figure 1: *ShadowStyles, with size = 4 points.*

Because `None` is not a constant, it cannot be the default value of a procedure argument. Therefore, we adopt the following convention: when a parameter whose type is `Shadow.T` has a default value of `NIL`, the procedure will use `Shadow.None` instead.

```
END Shadow.
```

3.2 The ShadowedVBT Interface

A `ShadowedVBT.T` is a filter whose parent's screen consists of the child's screen surrounded by a shadow. The colors and size of the shadow can be set dynamically, as can the style of the shadow. The parent's shape is determined from the child's shape by adding the shadow size.

```
INTERFACE ShadowedVBT;

IMPORT Filter, Shadow, VBT;

TYPE
  T <: Public;
  Public = Filter.T OBJECT
           METHODS
             <* LL.sup <= VBT.mu *>
             init (ch    : VBT.T;
                   shadow: Shadow.T := NIL;
                   style           := Shadow.Style.Flat): T;
           END;
```

The call `v.init(...)` initializes `v` as a `ShadowedVBT` with child `ch` and the given `shadow` and `style`. Each dimension of `v` exceeds the corresponding dimension of `ch` by `2 * ABS(shadow.size)`. If `shadow=NIL`, it defaults to `Shadow.None`.

```
PROCEDURE New (ch    : VBT.T;
               shadow: Shadow.T := NIL;
               style           := Shadow.Style.Flat): T;
<* LL.sup <= VBT.mu *>
```

`New(...)` is equivalent to `NEW(T).init(...)`.

```
PROCEDURE Set (v: T; shadow: Shadow.T);
<* LL.sup = VBT.mu *>
```

Change the size and colors of v's shadow and mark v for redisplay. If the size has in fact changed, notify v's parent that v's shape has changed.

```
PROCEDURE SetStyle (v: T; style: Shadow.Style);
<* LL.sup = VBT.mu *>
```

Change the `style` of v's shadow and mark v for redisplay.

```
PROCEDURE Get (v: T): Shadow.T;
<* LL.sup = VBT.mu *>
```

Return v's shadow.

```
PROCEDURE GetStyle (v: T): Shadow.Style;
<* LL.sup = VBT.mu *>
```

Return v's shadow style.

```
END ShadowedVBT.
```

3.3 The ShadowedBarVBT Interface

A `ShadowedBarVBT.T` is a VBT that displays a horizontally or vertically oriented 3-D rectangle.

If the shadow is `Raised` or `Ridged`, the top (left) half of the `VBT` is drawn in the light color, and the bottom (right) half in the dark color. If the shadow is `Lowered` or `Chiseled`, the colors are reversed. If the shadow is `Flat`, then the entire `VBT` is drawn in the background color.

```
INTERFACE ShadowedBarVBT;

IMPORT Axis, Shadow, VBT;

TYPE
  T <: Public;
  Public = VBT.Leaf OBJECT
           METHODS
             <* LL.sup <= VBT.mu *>
             init (axis  : Axis.T;
                   shadow: Shadow.T := NIL;
                   style           := Shadow.Style.Flat): T;
           END;
```

The call `v.init(...)` initializes `v` as a `ShadowedBarVBT` with the `axis` orientation and with the given `shadow` and `style`. If `shadow=NIL`, it defaults to `Shadow.None`.

If the `shadow.size` along the `axis` dimension results in an odd number of pixels, the extra pixel goes to the top (left) half. The shape of `v` is `ABS(shadow.size)` along the primary axis, and unconstrained in the other dimension.

```
PROCEDURE New (axis  : Axis.T;
               shadow: Shadow.T := NIL;
               style          := Shadow.Style.Flat): T;
<* LL.sup <= VBT.mu *>
```

New(...) is equivalent to NEW(T).init(...).

```
PROCEDURE Set (v: T; shadow: Shadow.T);
<* LL.sup = VBT.mu *>
```

Change the size and colors of v's shadow and mark v for redisplay. If the size has in fact changed, notify v's parent that v's shape has changed.

```
PROCEDURE SetStyle (v: T; style: Shadow.Style);
<* LL.sup = VBT.mu *>
```

Change the style of v's shadow and mark v for redisplay.

```
END ShadowedBarVBT.
```

4 Composing VBTs

4.1 The Multi Interface

A *multi* (`Multi.T`) is a `VBT` with *logical children* that may or may not correspond to its children in the `VBT` hierarchy. A *multi-filter* has a single logical child, and a *multi-split* has any number of logical children. Typically, logical children of multi `m` are also `VBT`-descendants of `m`, but this is not necessary.

A `BooleanVBT` is a good example of a multi-filter. It is implemented as an `HSplit` with 2 children. The first child is a `TSplit` with two `PixmapVBT`s corresponding to the "on" and "off" states; the second child is typically a string or a pixmap. A client views a `BooleanVBT` as having one child—the `HVSplit`'s 2nd child. `Multi.Child` retrieves this `VBT`, and `Multi.ReplaceChild` changes the child.

A `SplitterVBT` is a good example of a multi-split. It is built as an `HVSplit` with `HVBars` automatically inserted between children. The client of a `SplitterVBT` doesn't care about the `HVBars` at all. Invoking `Multi.Nth(v,i)` retrieves child `i` of the `SplitterVBT` (which is actually child `2*i` of the `HVSplit`), whereas `Split.Nth(v,i)` retrieves child `i` of the `HVSplit`. Invoking `Multi.Delete(v, ch)` deletes child `ch` and its adjacent `HVBar`, whereas `Split.Delete(v, ch)` deletes a single child `ch` of the `HVSplit`.

This interface defines the functionality that is common to all clients of multis (e.g., enumerating and deleting children). The `MultiClass` interface is for clients implementing `VBT` classes that are multis.

Warning: Unless you are a wizard, don't use procedures from the `Split` or `Filter` interfaces on a multi; use the corresponding procedures from this interface instead. The procedures here work like their `Split` and `Filter` counterparts if the argument is not a multi.

```
INTERFACE Multi;

IMPORT Point, VBT;

EXCEPTION NotAChild;

TYPE T = VBT.T;
```

A `Multi` *is a* `VBT` *with a* `MultiClass.T` *in its property set.*

4.1.1 Multi-filters

The following procedures can accept either a `Multi.T` or a non-multi `Filter.T` as the first argument. If the first argument is a non-multi `Filter.T`, the procedure just calls the corresponding procedure in the `Filter` interface.

```
PROCEDURE Child (v: VBT.T): VBT.T;
```

Return the child of `v`*, or* `NIL` *if there is no child.*

```
PROCEDURE ReplaceChild (v: VBT.T; ch: VBT.T): VBT.T;
```

Replace `v`*'s child by* `ch`*, detach and return* `v`*'s old child.*

4.1.2 Multi-splits

The following procedures can accept either a `Multi.T` or a non-multi `Split.T`. If `v` is a non-multi `Split.T`, the procedure just calls the corresponding procedure in the `Split` interface. Any `Split.NotAChild` exceptions are re-raised as `NotAChild`.

```
PROCEDURE Delete (v: VBT.T; ch: VBT.T)
  RAISES {NotAChild};
<* LL = VBT.mu *>
```

Delete the child `ch` *of the multi* `v` *and detach* `ch`*. Mark* `v` *for redisplay.*

```
PROCEDURE Replace (v: VBT.T; ch, new: VBT.T) RAISES {NotAChild};
<* LL = VBT.mu *>
```

Replace child `ch` *of* `v` *with* `new`*, detach* `ch` *(which must not be* `NIL`*), and mark* `v` *for redisplay.*

```
PROCEDURE Succ (v: VBT.T; ch: VBT.T): VBT.T RAISES {NotAChild};
<* LL >= VBT.mu *>
```

Return the child of `v` *that follows the child* `ch`*. If* `ch = NIL`*, then return the first child. If the result is* `NIL`*, then* `ch` *was the last child. The successor of* `NIL` *is* `NIL` *iff there are no children. The exception is raised if* `ch` *is not a child of* `v`*.*

The multiclass is expected to implement `Succ` in constant time; `Pred`, `Nth`, and `Index` may take time proportional to the number of children.

```
PROCEDURE Pred (v: VBT.T; ch: VBT.T): VBT.T RAISES {NotAChild};
<* LL >= VBT.mu *>
```

`Pred(v,ch) = x` *iff* `Succ(v,x) = ch`.

```
PROCEDURE Nth (v: VBT.T; n: CARDINAL): VBT.T;
<* LL >= VBT.mu *>
```

Returns child `n` *of* `v` *in succ order (child 0 is first), or* `NIL` *if there are* `n` *or fewer children.*

```
PROCEDURE NumChildren (v: VBT.T): CARDINAL RAISES {NotAChild};
<* LL >= VBT.mu *>
```

Return the number of children of `v`*.*

```
PROCEDURE Index (v: VBT.T; ch: VBT.T): CARDINAL RAISES {NotAChild};
<* LL >= VBT.mu *>
```

Return the value `n` *such that* `Nth(v,n)=ch`*. When* `ch=NIL`*, return the number of children of* `v`*.*

```
PROCEDURE Locate (v: VBT.T; READONLY pt: Point.T): VBT.T;
<* LL = VBT.mu *>
```

Return the child of `v` *that would receive a mouse click at point* `pt`*, or* `NIL` *if there is no such child.*

```
PROCEDURE Move (v: VBT.T; pred, ch: VBT.T) RAISES {NotAChild};
<* LL = VBT.mu *>
```

Move child `ch` *of* `v` *to follow* `pred`*.* `ch` *and, if non-*`NIL`*,* `pred`*, must be children of* `v`*.*

```
PROCEDURE Insert (v: VBT.T; pred, new: VBT.T) RAISES {NotAChild};
<* LL = VBT.mu *>
```

Add `new` *as a child of* `v` *following* `pred`*. If* `v` *doesn't allow more children than it already has,* `pred` *(or the first child, if* `pred = NIL`*) is deleted from the multi and discarded. The precise semantics are defined by the individual multis. The exception is raised if* `pred` *isn't a child of* `v`*, and it's a checked runtime error if* `new` *isn't detached.*

```
PROCEDURE AddChildArray (v: VBT.T; READONLY new: ARRAY OF VBT.T);
<* LL = VBT.mu *>
```

The non-`NIL` *elements of* `new` *become the last children of* `v`*. Equivalent to*

```
pred := Pred(v, NIL);
FOR i := FIRST(new) TO LAST(new) DO
  IF new[i] # NIL THEN
    Insert(v, pred, new[i]);
    pred := new[i]
  END
END
```

```
PROCEDURE AddChild (v: VBT.T;
    n0, n1, n2, n3, n4, n5, n6, n7, n8, n9: VBT.T := NIL);
<* LL = VBT.mu *>
```

Equivalent to

```
AddChildArray(v,
  ARRAY OF VBT.T{n0, n1, n2, n3, n4, n5, n6, n7, n8, n9})
```

```
END Multi.
```

4.2 Implementing your own Multis: The MultiClass Interface

A *multi* is a `VBT` to which a set of methods has been added (`replace`, `insert`, `pred`, etc.) to provide flexibility in the organization of "logical" children, so that it need not be constrained to be a subtype of `VBT.Filter` or `VBT.Split` and follow the rules for "`VBT`-children".

In a language with multiple inheritance, multis would simply inherit different methods from different parent-types. In Modula-3, however, we achieve this affect by creating an instance `mc` of type `MultiClass.T`, and attaching `mc` to a `VBT` `v` by way of `v`'s property set. `mc` points back to `v` via the field `mc.vbt`.

Clients that are defining their own multis can make a `VBT` `v` "into" a multi by calling `Be(v, mc)` during initialization. They should also call `BeChild` on each new child when it is inserted, and `UnChild` when a child of a multi is deleted. `Multi.Replace` calls `BeChild` automatically.

Some clients export their multiclass, e.g., `BooleanVBT.MC`, and their `init` methods take a multiclass argument which defaults to `NIL`. By convention, this means that the `init` method will use a new instance of the exported multiclass, e.g., `NEW(MC)`, as the multiclass.

```
INTERFACE MultiClass;

IMPORT List, Multi, VBT;

TYPE
```

```
T <: Public;
Public =
  MUTEX OBJECT
    vbt: VBT.T;               (* READONLY *)
    (* This field is filled in by the "Be" procedure. *)
  METHODS
    (* All clients should override "replace" *)
    replace (ch, new: VBT.T) RAISES {Multi.NotAChild};

    (* Multi-split clients should override "insert" and
       "move" *)
    insert (pred, new: VBT.T) RAISES {Multi.NotAChild};
    move   (pred, ch: VBT.T) RAISES {Multi.NotAChild};

    (* The following four methods have defaults. *)
    succ  (ch: VBT.T)  : VBT.T RAISES {Multi.NotAChild};
    pred  (ch: VBT.T)  : VBT.T RAISES {Multi.NotAChild};
    nth   (n: CARDINAL): VBT.T;
    index (ch: VBT.T)  : CARDINAL RAISES {Multi.NotAChild};

  END;
```

By definition,

`mc.replace(ch, new)`
implements
`Multi.Replace(mc.vbt, ch, new)`

`mc.replace(ch, NIL)`
implements
`Multi.Delete(mc.vbt, ch)`

`mc.insert(pred, new)`
implements
`Multi.Insert(mc.vbt, pred, new)`

and
`mc.move(pred, ch)`
implements
`Multi.Move(mc.vbt, pred, ch)`

For the `replace` method, `ch` may be `NIL` if `mc.vbt` is a filter.

There are no defaults for `replace`, `insert`, or `move`. Some multiclasses may be able to implement `mc.move(pred,ch)` simply as

```
mc.replace(ch, NIL); mc.insert(pred, ch)
```

The default `succ` method finds the successor of `ch` by a depth-first walk of `mc.vbt`'s descendants, starting after `ch`, and stopping at the first `VBT` `w` for which

```
IsChild(mc.vbt, w)
```

is `TRUE`, or when all `mc.vbt`'s descendants have been visited. `succ` returns `NIL` if `ch` has no successor.

The default `pred` method finds the predecessor of `ch` by repeatedly calling `mc.succ`.

The default `nth` method returns the nth child of `mc.vbt` by repeatedly calling `mc.succ`.

The default `index` method returns the index of `ch` in `mc.vbt` by repeatedly calling `mc.succ`.

```
PROCEDURE Be (v: VBT.T; mc: T);
```

Make `v` into a multi by storing `mc` on `v`'s property set and setting `mc.vbt` to `v`.

```
PROCEDURE Resolve (v: VBT.T): T;
```

Return the multiclass of `v`, that is, the `mc` for which `Be(v,mc)` was previously called. Return `NIL` if there is no such `mc`.

```
PROCEDURE BeChild (v: VBT.T; ch: VBT.T);
```

Make `ch` into one of `v`'s children that is exposed to the client via the `Multi` interface. It is possible for `ch` to be a child of more than one multi, and it is possible that `ch` is not related to `v` in the `VBT` hierarchy.

```
PROCEDURE UnChild (v: VBT.T; ch: VBT.T);
```

Unmark `ch` as one of `v`'s children that is exposed to the client via the `Multi` interface.

```
PROCEDURE IsChild (v: VBT.T; ch: VBT.T): BOOLEAN;
```

Return `TRUE` iff `BeChild(v,ch)` was previously invoked and `UnChild(v,ch)` has not been subsequently called.

```
PROCEDURE Parents (ch: VBT.T): List.T (* of VBT.T *);
```

Return a list of `VBT`s for which `IsChild(v,ch)` is `TRUE`. The list may be `NIL`.

```
END MultiClass.
```

5 Buttons

Buttons in VBTkit differ from buttons in Trestle (i.e., `ButtonVBT` and its subclasses). VBTkit buttons are referred to as *switches*.

Trestle buttons	VBTkit switches
Buttons are passed an `action` procedure.	Switches define a `callback` method, which makes it easier to define a subclass of a switch.
Buttons provide visual feedback as part of their `pre`, `during`, and `post` methods.	Switches assume that their `VBT`-child is a feedback, and the `pre`, `during`, and `post` methods invoke the feedback's `normal` and `excited` methods appropriately. This facilitates a "mix-and-match" of button looks with gestures (e.g., putting a radio button within a menu).
Buttons are filters.	Switches are multi-filters. The multi-child of the feedback is the switch's multi-child. This makes is it easy to create buttons with sophisticated visual feedback, while retaining the model that a button is a "filter with an arbitrary child". Clients should use the `Multi` interface to access the arbitrary child, and the `Filter` interface to access the feedback.

5.1 Providing Visual Feedback

5.1.1 The Feedback Interface

A *feedback* (`Feedback.T`) is a multi-filter that provides visual feedback for its child. The essence of a feedback is its methods for displaying itself in a *normal* or *excited* state. The normal style is drawn to give permanent feedback, and the excited style gives transitory feedback (e.g., while a button is pressed). In addition, a feedback maintains a Boolean flag so that it can distinguish between an "on" and "off" state (e.g., for use by a `BooleanVBT`).

An arbitrary multi-filter is made into a feedback by calling `FeedbackClass.Be`.

```
INTERFACE Feedback;

IMPORT VBT;

TYPE T = VBT.T;
```

A `Feedback` is a multi-filter with a `FeedbackClass.T` in its property set.

```
PROCEDURE Normal (v: T);
```

Invoke `v`'s `normal` method.

```
PROCEDURE Excited (v: T);
```

Invoke v's `excited` method.

```
PROCEDURE SetState (v: T; state: BOOLEAN);
```

Record the `state` and then invoke whichever of v's methods, `normal` or `excited`, was most recently invoked.

```
PROCEDURE GetState (v: T): BOOLEAN;
```

Return the value of the most recent call to `SetState`.

```
END Feedback.
```

5.1.2 Implementing your own Feedbacks

A *feedback* is a `VBT` (specifically, a multi-filter) to which `normal` and `excited` methods are attached. Just as `MultiClass` implements the attachment of methods to any `VBT` to make it a "multi", `FeedbackClass` implements the attachment of `normal` and `excited` methods to any multi-filter to make it a "feedback".

To do this, we create an instance `fc` of `FeedbackClass.T` and attach it to a multi-filter `v` by way of `v`'s property set. `fc` points back to `v` via the field `fc.vbt`.

Clients that define their own feedbacks can make a multi-filter `v` into a feedback by calling `Be(v,fc)` during initialization.

Some clients export their feedback-class, e.g., `BorderedFeedbackVBT.FC`, and their `init` methods take a feedback-class argument which defaults to `NIL`. By convention, this means that the `init` method will use a new instance of the exported multiclass, e.g., `NEW(FC)`, as the feedback-class.

```
INTERFACE FeedbackClass;

IMPORT VBT;

TYPE
  T <: Public;
  Public = OBJECT
             vbt  : VBT.T;        (* READONLY *)
             last           := LastCall.Normal;
             state          := FALSE;
           METHODS
             normal  ();
             excited ();
           END;
```

The default methods for `normal` and `excited` are no-ops.

```
TYPE
```

```
  LastCall = {Normal, Excited};

PROCEDURE Be (v: VBT.T; fc: T);
```

Make `v` *into a* `Feedback.T` *by storing* `fc` *in* `v`*'s property set and settting* `fc.vbt` *equal to* `v`*. The initial state is* `FALSE` *("off").*

```
PROCEDURE Resolve (v: VBT.T): T;
```

Return `fc` *for which* `Be(v,fc)` *was previously called. Return* `NIL` *if there is no such* `fc`*.*

```
END FeedbackClass.
```

5.1.3 The FeedbackVBT Interface

A `FeedbackVBT` is a feedback that provides no visual feedback at all. It's useful as a place-holder or super-type.

```
INTERFACE FeedbackVBT;

IMPORT FeedbackClass, Filter, VBT;

TYPE
  T <: Public;
  Public = Filter.T OBJECT
           METHODS
             init (ch: VBT.T; feedbackclass: FC := NIL): T
           END;

  FC <: FeedbackClass.T;

END FeedbackVBT.
```

5.1.4 The BiFeedbackVBT Interface

A `BiFeedbackVBT` is a feedback that is used for composing other feedbacks. It doesn't have any visual behavior itself. The crux is that it is assumes that the `BiFeedbackVBT`'s child, `ch`, is a feedback, and that `ch`'s multi-child (i.e., the multi-grandchild) is also a feedback.

```
INTERFACE BiFeedbackVBT;

IMPORT Feedback, FeedbackClass, Filter, MultiClass;

TYPE
  T <: Public;
  Public = Filter.T OBJECT
```

```
              METHODS
                init (ch            : Feedback.T;
                      multiclass    : MC           := NIL;
                      feedbackclass: FC            := NIL  ): T;
              END;
```

```
TYPE
  FC <: FeedbackClass.T;
```

The type `FC` overrides the `normal` and `excited` methods of the type `FeedbackClass.T`. The `excited` and `normal` methods invoke the corresponding methods on both child and grandchild.

```
TYPE
  MC <: MultiClass.T;
```

The type `MC` overrides the `replace`, `pred`, and `succ` methods of the type `MultiClass.T`.

```
END BiFeedbackVBT.
```

5.1.5 The BorderedFeedbackVBT Interface

A `BorderedFeedbackVBT` is a feedback that display a border as visual feedback to another `VBT`. In the "on" state, the border is drawn using the foreground color; in the "off" state, the border is drawn using the background color. In an "excited" state, the border is drawn with a gray pixmap; in the "normal" state, the border is drawn with a solid pixmap.

```
INTERFACE BorderedFeedbackVBT;

IMPORT BorderedVBT, FeedbackClass, PaintOp, VBT;

CONST
  PtsPerMM = 25.4 / 72.0;
  Default  = BorderedVBT.Default * PtsPerMM;

TYPE
  Points = REAL;
  T <: Public;
  Public = BorderedVBT.T OBJECT
           METHODS
             init (ch  : VBT.T;
                   size: Points  := Default;
                   op            := PaintOp.BgFg;
                   feedbackclass: FC           := NIL ): T
           END;

  FC <: FeedbackClass.T;
```

The type FC overrides the normal and excited methods of FeedbackClass.T.

```
END BorderedFeedbackVBT.
```

5.1.6 The MarginFeedbackVBT Interface

A MarginFeedbackVBT is a feedback (and also a multi-filter) that provides visual feedback to the left of another VBT. The feedback-class FC displays the exact same pixmap for both normal and excited modes and for both states.

```
INTERFACE MarginFeedbackVBT;

IMPORT FeedbackClass, HVSplit, MultiClass, Shadow, VBT;

TYPE
  T <: Public;
  Public = HVSplit.T OBJECT
           METHODS
             init (ch           : VBT.T;
                   marginVBT    : VBT.T;
                   multiclass   : MC      := NIL;
                   feedbackclass: FC      := NIL  ): T
           END;

  MC <: MultiClass.T;
```

The type MC overrides the replace, pred, and succ methods of MultiClass.T.

```
  FC <: FeedbackClass.T;
```

The following procedures create some popular types of MarginFeedbackVBTs. NewCheck displays a check mark or nothing; NewBox displays a box that is either empty or contains an "x"; NewBullet displays a circle with or without a bullet in its center.

```
PROCEDURE NewCheck  (ch: VBT.T; shadow: Shadow.T := NIL): T;
PROCEDURE NewBox    (ch: VBT.T; shadow: Shadow.T := NIL): T;
PROCEDURE NewBullet (ch: VBT.T; shadow: Shadow.T := NIL): T;

END MarginFeedbackVBT.
```

5.1.7 The ShadowedFeedbackVBT Interface

A ShadowedFeedbackVBT is a feedback that displays a *shadow* as visual feedback to another VBT. The default methods change the style of the shadow to the style indicated in the init method. However, on a monochrome screen or when the shadow's size is a negative number, the `excited` method inverts a highlighter above itself and the `normal` method takes off the highlighting.

```
INTERFACE ShadowedFeedbackVBT;

IMPORT FeedbackClass, Shadow, ShadowedVBT, VBT;

TYPE
  T <: Public;
  Public =
    ShadowedVBT.T OBJECT
    METHODS
      init (ch              : VBT.T;
            shadow          : Shadow.T := NIL;
            onStyle                    := Shadow.Style.Ridged;
            onExcitedStyle             := Shadow.Style.Chiseled;
            offStyle                   := Shadow.Style.Raised;
            offExcitedStyle            := Shadow.Style.Lowered;
            feedbackclass   : FC       := NIL                    ):
            T
    END;

  FC <: FeedbackClass.T;

PROCEDURE NewMenu (ch: VBT.T; shadow: Shadow.T := NIL): T;
```

Create the standard `shadow` used for menu buttons. The specification of the standard for menu buttons is as follows: the normal states are displayed as `Shadow.Style.Flat`; the excited state as `Shadow.Style.Lowered` style.

```
END ShadowedFeedbackVBT.
```

5.2 Basic Switches

5.2.1 The SwitchVBT Interface

A *switch* (`SwitchVBT.T`) is the VBTkit version of a *button* (`ButtonVBT.T`). A switch `s` follows these three conventions:

1. `s` has a `callback` method. This method will be invoked whenever a button would have invoked its `action` procedure.
2. The `VBT`-child of `s` is a feedback, `f`. `s.pre()` invokes `Feedback.Excited(f)`. `s.post()` and `s.cancel()` invoke `Feedback.Normal(f)`.
3. `s` is also a multi-filter. Its multi-child is defined to be the multi-child of `f`.

Although all VBTkit switches follow these conventions and are subtypes of `ButtonVBT.T`, not all are subtypes of `SwitchVBT.T`. A `MenuSwitchVBT`, for example, is a subtype of `MenuBtnVBT.T`.

```
INTERFACE SwitchVBT;

IMPORT ButtonVBT, Feedback, MultiClass, VBT;

TYPE
  T <: Public;
  Public =
    ButtonVBT.T OBJECT
    METHODS
      init     (f: Feedback.T; multiclass: MC := NIL): T;
      callback (READONLY cd: VBT.MouseRec) := DefaultCallback;
    OVERRIDES
      pre    := Pre;
      post   := Post;
      cancel := Cancel;
    END;

  MC <: MultiClass.T;
```

The type `MC` overrides the `replace`, `pred`, and `succ` methods of the type `MultiClass.T`.

```
PROCEDURE Pre    (v: ButtonVBT.T);
PROCEDURE Post   (v: ButtonVBT.T);
PROCEDURE Cancel (v: ButtonVBT.T);

PROCEDURE DefaultCallback (         v : ButtonVBT.T;
                           READONLY cd: VBT.MouseRec );
```

A no-op.

```
END SwitchVBT.
```

5.2.2 The AnchorSplit Interface

An `AnchorSplit` is a multi-split version of `AnchorBtnVBT`. The first child is the *anchor* that is displayed (such as a text string or an icon). The second child is the *menu* that is displayed when the anchor is activated. Attempts to give an anchor-split more than two children cause the extra children to be lost.

At initialization time, the feedback for the anchor is specified. It must be a childless multi-filter. Also at initialization time, a frame is specified that will surround the menu. The frame is also a childless multi-filter.

```
INTERFACE AnchorSplit;
```

```
IMPORT AnchorBtnVBT, Feedback, Multi, MultiClass, VBT;

TYPE
  T <: Public;
  Public = AnchorBtnVBT.T OBJECT
           METHODS
             init (f           : Feedback.T;
                   menuFrame   : Multi.T;
                   n           : CARDINAL     := 0;
                   anchorParent: VBT.T        := NIL;
                   hfudge                     := 0.0;
                   vfudge                     := 0.0;
                   multiclass  : MC           := NIL  ): T
           END;
```

The call `v.init(...)` initializes `v` as an `AnchorSplit`. The feedback `f` and the multi `menuFrame` must have no multi-children. That is, calling `Multi.Child(f)` and `Multi.Child(menuFrame)` must both return `NIL`. The other parameters, `n`, `anchorParent`, `hfudge`, and `vfudge` are the same as in `AnchorBtnVBT`.

```
TYPE
  MC <: MultiClass.T;
```

The type `MC` overrides the `replace`, `insert`, and `succ` methods of the type `MultiClass.T`.

```
END AnchorSplit.
```

5.2.3 The DragSwitchVBT Interface

A `DragSwitchVBT` is a switch version of Trestle's `DragBtnVBT`.

Actually, `DragBtnVBT` does not exist. If it existed, it would be a `ButtonVBT` subclass that generates events while a mouse button is down and has moved since the last time an event was generated. Events are generated even when the mouse is outside the `VBT`'s domain, as long as it remains on the screen.

The `pre` method is invoked on the first click in the `VBT`; the `post` method is called on an uncanceled upclick; the `cancel` method is called whenever the mouse is "chorded"; the `during` method is called whenever the mouse has moved since the last call to `during` or `pre`. A new cage containing the current cursor position will be set before calls to `pre` and `during`. The `callback` method is invoked on an uncancelled upclick. The default `during` and `callback` methods are no-ops.

```
INTERFACE DragSwitchVBT;

IMPORT ButtonVBT, Feedback, SwitchVBT, VBT;

TYPE
```

```
  T <: Public;
  Public = ButtonVBT.T OBJECT
           METHODS
             init     (f: Feedback.T; multiclass: MC := NIL): T;
             during   (READONLY cd: VBT.PositionRec);
             callback (READONLY cd: VBT.MouseRec);
           END;

  MC = SwitchVBT.MC;

END DragSwitchVBT.
```

5.2.4 The GuardedBtnVBT Interface

A `GuardedBtnVBT` is a `ButtonVBT` that protects its child against unintentional mouse clicks. While the guard is displayed, mouse clicks are not forwarded. To remove the guard, click on the button. The guard is restored after the next upclick, chord-cancel, or when the mouse leaves the domain of the `VBT`.

Typically, a `GuardedBtnVBT` is placed above a "dangerous" button, like one that terminates an application. This forces the user to click twice to terminate the application—the first time to remove the guard, and the second time to invoke the button that terminates the application.

A `GuardedBtnVBT` is much closer to being a VBTkit switch than a Trestle button. There's a `callback` method (invoked when the guard is removed), and the guard is a multi-filter. However, the client does not provide a feedback; it is hard-wired into the `GuardedBtnVBT` implementation.

```
INTERFACE GuardedBtnVBT;

IMPORT ButtonVBT, MultiClass, PaintOp, VBT;

TYPE
  T <: Public;
  Public = ButtonVBT.T OBJECT
           METHODS
             init (ch        : VBT.T;
                   colors    : PaintOp.ColorScheme := NIL;
                   multiclass: MC                  := NIL  ): T;
             callback (READONLY cd: VBT.MouseRec);
           END;

  MC <: MultiClass.T;
```

The type `MC` overrides the `replace`, `pred`, and `succ` methods of `MultiClass.T`.

```
END GuardedBtnVBT.
```

5.2.5 The MenuSwitchVBT Interface

A `MenuSwitchVBT` is a switch version of Trestle's `MenuBtnVBT`.

```
INTERFACE MenuSwitchVBT;

IMPORT Feedback, MenuBtnVBT, SwitchVBT, VBT;

TYPE
  T <: Public;
  Public = MenuBtnVBT.T OBJECT
           METHODS
             init     (f: Feedback.T; multiclass: MC := NIL): T;
             callback (READONLY cd: VBT.MouseRec);
           END;

  MC = SwitchVBT.MC;

END MenuSwitchVBT.
```

5.2.6 The QuickSwitchVBT Interface

A `QuickSwitchVBT` is a switch version of Trestle's `QuickBtnVBT`.

```
INTERFACE QuickSwitchVBT;

IMPORT Feedback, QuickBtnVBT, SwitchVBT, VBT;

TYPE
  T <: Public;
  Public = QuickBtnVBT.T OBJECT
           METHODS
             init     (f: Feedback.T; multiclass: MC := NIL): T;
             callback (READONLY cd: VBT.MouseRec);
           END;

  MC = SwitchVBT.MC;

END QuickSwitchVBT.
```

5.2.7 The TrillSwitchVBT Interface

A `TrillSwitchVBT.T` is a switch version of Trestle's `TrillBtnVBT`.

Actually, a `TrillBtnVBT` does not exist. If it existed, it would be a button that generates events repeatedly while the mouse is down and in its domain. When the mouse leaves the domain, events generation is suspended until the mouse returns.

The implementation uses the `AutoRepeat` interface for repeatedly generating events. That interface defines the parameters that control how frequently events are generated, and how long to wait before starting to auto-repeat.

```
INTERFACE TrillSwitchVBT;

IMPORT ButtonVBT, Feedback, SwitchVBT, VBT;

TYPE
  T <: Public;
  Public = ButtonVBT.T OBJECT
           METHODS
             init     (f: Feedback.T; multiclass: MC := NIL): T;
             callback (READONLY cd: VBT.MouseRec);
           END;

  MC = SwitchVBT.MC;

END TrillSwitchVBT.
```

5.3 Booleans and Choices

5.3.1 The BooleanVBT Interface

A `BooleanVBT` is a multi-filter. Its `VBT`-child is a VBTkit switch of some sort (e.g., `SwitchVBT`, `MenuSwitchVBT`); the `BooleanVBT` maintains a Boolean state for that switch. When the `callback` method of the switch would normally be invoked, the value of the state of the `BooleanVBT` is toggled and the `callback` method on the `BooleanVBT` is invoked. The visual feedback is provided by the switch's feedback. An appropriate feedback to use is `MarginFeedbackVBT.NewCheck`.

The multi-child of a `BooleanVBT` is defined to be the multi-child of the switch.

```
INTERFACE BooleanVBT;

IMPORT ButtonVBT, Filter, MultiClass, VBT;

TYPE
  T <: Public;
  Public =
    Filter.T OBJECT
    METHODS
      init     (switch: ButtonVBT.T; multiclass: MC := NIL): T;
      callback (READONLY cd: VBT.MouseRec);
    END;
```

The call `v.init(switch, ...)` initializes `v` as a `BooleanVBT` with an initial state of `FALSE`.

Warning: This call modifies the `action` field of `switch`. Clients must not change the `VBT` child of a `BooleanVBT` directly. Clients can, of course, change multi-child of `v` (e.g., some displayed text or pixmap).

Although all VBTkit switches are subtypes of `ButtonVBT.T`, not all are subtypes of `SwitchVBT.T`. A `MenuSwitchVBT`, for example, is a subtype of `MenuBtnVBT.T` and is neither a subtype nor a supertype of `SwitchVBT.T`.

```
TYPE MC <: MultiClass.T;
```

The type `MC` overrides the `replace`, `pred`, and `succ` methods of the type `MultiClass.T`.

```
PROCEDURE Put (v: T; state: BOOLEAN);
```

Set v's state.

```
PROCEDURE Get (v: T): BOOLEAN;
```

Returns v's current state.

```
END BooleanVBT.
```

5.3.2 The ChoiceVBT Interface

A `ChoiceVBT` multi-filter behaves in concert with other `ChoiceVBT`s to implement *radio buttons*. At any given time, all of the related `ChoiceVBT`s have a `FALSE` state, except, at most, one. This one is called the *selection*. If no choices are in the `TRUE` state, the selection is `NIL`.

Structurally, a `ChoiceVBT` is identical to a `BooleanVBT`; refer to that interface for details.

```
INTERFACE ChoiceVBT;

IMPORT BooleanVBT, ButtonVBT, Filter, Radio, VBT;

TYPE
  T <: Public;
  Public = Filter.T OBJECT
           METHODS
             callback (READONLY cd: VBT.MouseRec);
             init (switch    : ButtonVBT.T;
                   group     : Radio.T;
                   multiclass: MC            := NIL): T
           END;
```

The call `v.init(...)` initializes `v` as a `ChoiceVBT` with an initial state of `FALSE` and in the a radio group `group`.

```
TYPE MC = BooleanVBT.MC;
```

The type `MC` *overrides the* `replace`, `pred`, *and* `succ` *methods of* `Boolean-VBT.MC`.

```
PROCEDURE Put (v: T);
```

Let g be v's radio group. Set the state of g's current selection to `FALSE`; *make* `v` *the current selection thereby setting its state to* `TRUE`.

```
PROCEDURE Clear (v: T);
```

Let g be v's radio group. Set the state of g's current selection to `FALSE`; *make* `NIL` *the current selection.*

```
PROCEDURE Get (v: T): T;
```

Return the current selection in v's radio group. Note that this may be `NIL`.

```
PROCEDURE GetGroup (v: T): Radio.T;
```

Return the radio group containing `v`.

```
 END ChoiceVBT.
```

5.3.3 The Radio Interface

The `Radio` data structure supports mutually exclusive selections from a group of objects.

This interface provides the notion of a group of `VBT`s. After a group is created, `VBT`s can be added to and removed from the group. At most one member of the group is designated as the *current selection*. The current selection may be set and retrieved; it may be `NIL`. All routines are self-locking.

This interface could be used by non-`VBT` clients by recompiling with `MemberType` set to something other than `VBT.T`. It's probably a good candidate for Modula-3 generics.

```
INTERFACE Radio;

IMPORT List, VBT;

TYPE
  T <: REFANY;
  MemberType = VBT.T;

EXCEPTION NotAMember;

PROCEDURE New (): T;
```

Creates a new group.

```
PROCEDURE Add (group: T; member: MemberType);
```

Add `member` to `group`. This procedure is a no-op if `member` is `NIL` or if it's already a member of `group`.

```
PROCEDURE Remove (group: T; member: MemberType)
  RAISES {NotAMember};
```

Remove `member` from `group`. If `member` was the selection, set the selection to `NIL`. Raise the exception if `member` is not a member of `group`.

```
PROCEDURE Get (group: T): MemberType;
```

Return the currently selected member of `group`. If no member is selected, return `NIL`.

```
PROCEDURE Put (group: T; member: MemberType) RAISES {NotAMember};
```

Make `member` the current selection of `group`. `member` may be `NIL`. The exception is raised if `member` is non-`NIL` and not a member of `group`.

```
PROCEDURE IsMember (group: T; member: MemberType): BOOLEAN;
```

Return whether `member` belongs to `group`

```
PROCEDURE Members (group: T): List.T;
```

Return a list of members. The elements will appear in the order in which they were added to `group`.

```
END Radio.
```

5.4 Drag'n'Drop

5.4.1 The SourceVBT Interface

A `SourceVBT` is a button that looks for special `VBT`s marked as *targets* while the mouse button remains down. When the mouse is over a target, a procedure posted on the active `SourceVBT` is invoked to see if the target is acceptable for the source. If so, an "excited" method on the target is invoked to give feedback. Eventually, a target's "normal" method is called to remove the feedback.

```
INTERFACE SourceVBT;

IMPORT DragSwitchVBT, Feedback, FeedbackClass, HighlightVBT, VBT;

TYPE
  T <: Public;
  Public = DragSwitchVBT.T OBJECT
```

```
METHODS
  init (f: Feedback.T): T;
  hit (target: VBT.T; READONLY cd: VBT.PositionRec):
       BOOLEAN := NeverHit
END;
```

The call `v.init(...)` initializes `v` as a `SourceVBT`. The type `SourceVBT.T` overrides the `pre`, `post`, `during`, and `cancel` methods of its supertype.

The default `pre` method changes the cursor to a starburst. The default `during` method calls the `hit` method whenever it is on a point controlled by a `VBT` that is a target. If the `hit` method returns `TRUE`, the target's `excited` and `normal` methods are called appropriately. The `post` method invokes the target's `normal` method and restores the original cursor. It's guaranteed that a target's `excited` and `normal` methods are called in non-nested pairs.

```
PROCEDURE GetHighlighter (v: T): HighlightVBT.T;
```

Returns the `HighlightVBT` above the nearest Trestle-installed ancestor of `v`.

```
PROCEDURE GetTarget (v: T): Target;
```

Called by a `callback` method to find out the current target. That is, returns `w` for which the most recent call to `HitProc(v,w)` returned `TRUE`, or `NIL` if `HitProc` returned `FALSE` or if the mouse is not over a valid target.

```
PROCEDURE AlwaysHit (         v     : Public;
                              target: VBT.T;
                     READONLY cd    : VBT.PositionRec): BOOLEAN;
```

Always return `TRUE`.

```
PROCEDURE NeverHit (         v     : Public;
                             target: VBT.T;
                    READONLY cd    : VBT.PositionRec): BOOLEAN;
```

Always returns `FALSE`.

5.4.2 Targets

```
TYPE Target = VBT.T;
```

A target is a `VBT` on which `BeTarget` has been invoked.

```
TYPE TargetClass <: FeedbackClass.T;
```

A target class determines the feedback when a target's `excited` method is called.

```
PROCEDURE BeTarget (w: VBT.T; class: TargetClass);
```

Make `w` into a target for a `SourceVBT`. As a target, `w` may be passed to some `SourceVBT`'s `hit` method.

```
PROCEDURE GetSource (w: Target): T;
```

Called by a target's `normal` or `excited` methods to find out the `SourceVBT` causing the method to be invoked.

```
PROCEDURE NewInserterTarget (): TargetClass;
```

Displays a grid over itself when excited. Appropriate for an adjusting bar in a tiling window manager, like the Trestle window manager on SRC's Firefly. The parent of the target must be an `HVSplit`, and grid has a minimun size in the `HVSplit`'s axis.

```
PROCEDURE NewSwapTarget (): TargetClass;
```

Displays a grid over itself when excited. This target is appropriate for a non-adjusting bar in a tiling window manager.

```
PROCEDURE NewTarget (): TargetClass;
```

Inverts itself when excited. This target class is a general-purpose target.

```
PROCEDURE TargetClassOf (w: Target): TargetClass;
```

Return the `class` argument for which there was a previous call to `BeTarget(w, class)`, or `NIL` if there was no such call.

```
END SourceVBT.
```

6 Subwindows

6.1 The ZChassisVBT Interface

A `ZChassisVBT` multi-filter provides a *chassis* for a subwindow. (A *subwindow* is a non-background child of a `ZSplit`.) The visual display of the chassis is hard-wired into this module; in particular, it won't look like a top-level window of most of the common X window managers. The top of the chassis is a banner containing (from left to right) a *close button*, *draggable title*, and a *grow button*.

Clicking on the close button unmaps the `ZChassisVBT`, thereby causing it to disappear. Clicking on the grow button allows the user to change the size of the `ZChassisVBT`, subject to its size contraints. That is, the user isn't allowed to make the interior of the chassis smaller or larger than its reported bounds along each dimension. Clicking on the title allows the user to reposition the `ZChassisVBT` within its parent.

```
INTERFACE ZChassisVBT;

IMPORT List, MultiClass, Shadow, VBT, ZChildVBT;

TYPE
  T <: Public;
  Public = ZChildVBT.T OBJECT
           METHODS
             init (ch        : VBT.T;
                   title     : VBT.T;
                   shadow    : Shadow.T := NIL;
                   closable             := TRUE;
                   open                 := TRUE;
                   at        : List.T   := NIL;
                   multiclass: MC       := NIL   ): T
           END;

  MC <: MultiClass.T;
```

The type `MC` overrides `MultiClass.T.replace`.

The call `v.init(...)` initializes `v` as a `ZChassisVBT`. It is assumed that `v` will be a subwindow. The interior of the chassis, `ch`, is `v`'s child in the multi-child sense.

A close button is displayed iff `closable` is set. The grow button is implemented with a `ZGrowVBT`. The also functions as a drag bar. It is implemented by a `ZMoveVBT` around `title`. The looks of these buttons is governed by the `shadow` parameter.

If `open` is set, then `v` will be `mapped` when it is inserted as a child of its parent `ZSplit`. It's location is specified in `at`. More details about `open` and `at` appear in the `ZChild` interface.

In the current implementation, a chassis has the following general structure (using FormsVBT notation):

```
(Border
  (VBox
    (HBox (CloseButton "C")
          (ZMoveVBT title)
          (ZGrowVBT "G"))
    (Frame ch)))
```

See Figure 6.1.

However, don't try to traverse the `VBT` tree directly; it is subject to change. To retrieve the contents of a chassis `v`, use `Multi.Child(v)`.

`v.callback(cd)` is invoked when the CloseButton is activated.

A `ZChassisVBT`'s move, grow, and close buttons are not effective unless the `ZChassis` is a non-background child of a `ZSplit`.

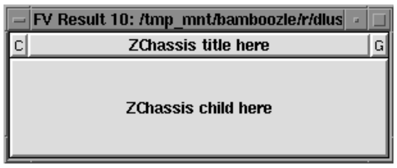

Figure 2: *A ZChassis.*

```
END ZChassisVBT.
```

6.2 The ZChildVBT Interface

An `ZChildVBT.T` is a `VBT` that is typically used as a subwindow. (A *subwindow* is a non-background child of a `ZSplit`.)

A `ZChildVBT` is a subclass of a `HighlightVBT`, thereby insulating any highlighting done in it against highlighting done in other subwindows. You need to use a `ZBackgroundVBT` around the background child in order to insulate highlighting in it against the subwindows.

A `ZChildVBT` has two properties that are specified during initialization: a flag to indicate whether the subwindow is initially "mapped" (displayed); and a list that specifies the default location of the subwindow within the parent `ZSplit`. The list contains two or four numbers, which may be integers or real numbers.

If the list contains two numbers, these specify the center of the subwindow as `(h,v)`. Otherwise, the list has four numbers, and these specify the rectangle of the subwindow, as `(west, north, east, south)`.

If the numbers are integers, these specify distance in points from the upper left corner of the subwindow. This distance will be maintained regardless of how the window is reformatted. Otherwise, the numbers must be real numbers in the range 0.0–1.0, and these specify proportional position of the width or height of the window. Thus the default value, `(0.5 0.5)`, means that the subwindow is centered in the `ZSplit`.

If the two-number form is used, they may be followed by a corner specification, one of the symbols `NW`, `NE`, `SW`, or `SE`. If a corner specification is given, the position applies to that corner of the window rather than to the center. However, the system will not pop up a subwindow with its northwest corner north or west of the visible portion of the `ZSplit`; it will override the specified position as necessary to bring it into view.

It is legal, but not recommended, to mix integers and reals in the same list.

Finally, in order for the reformatting to meet specifications above, the client must call `Inserted` after the `VBT` is inserted as a child of a `ZSplit`; the client must call `Moved` after the `VBT` is repositioned by the user; and the client must call `Grow` after the size of the `VBT` is changed by the user.

```
INTERFACE ZChildVBT;
```

```
IMPORT HighlightVBT, List, VBT;

TYPE
  T <: Public;
  Public =
    HighlightVBT.T OBJECT
    METHODS
      init     (ch: VBT.T; open := TRUE; at: List.T := NIL): T;
      callback (READONLY cd: VBT.MouseRec);
    END;
```

The call `v.init(...)` initializes `v` as a `ZChildVBT`. It is assumed that `v` will be a subwindow.

```
PROCEDURE InitiallyMapped (vbt: VBT.T): BOOLEAN;
```

"Should `vbt` *be mapped initially?" This question is answered as follows: If* `vbt` *is a* `ZChild`*, return the value of* `open` *when it was initialzied. Otherwise, return* `TRUE`*.* `VBT.Parent(vbt)` *must be a subclass of* `ZSplit`*.*

```
PROCEDURE Pop (vbt: VBT.T; forcePlace := FALSE);
```

Map `vbt` *and lift it to the top of its parent's children. If* `forcePlace` *is set, position* `vbt` *at the* `at` *specification.* `VBT.Parent(vbt)` *must be a subclass of* `ZSplit`*.*

```
PROCEDURE Inserted (vbt: VBT.T);
```

Call this procedure after `vbt` *has been inserted into a* `ZSplit`*. This procedure sets a* `ReshapeControl` *object on* `vbt`*. If* `vbt` *isn't a* `ZChild`*, the* `ReshapeControl` *tries to center* `vbt`*, subject to the contraint that its northwest corner is just visible. If* `vbt` *is a* `ZChild`*, the* `ReshapeControl` *will follow* `vbt.at` *until* `vbt` *is moved by the user (usually because* `Moved` *is called). At that point, the northwest corner of* `vbt` *will remain at that position relative to its parent, until the user moves it again.* `VBT.Parent(vbt)` *must be a sublcass of* `ZSplit`*.*

```
PROCEDURE Moved (vbt: VBT.T);
```

Call this procedure after `vbt` *has been moved by a user. If* `vbt` *is a* `ZChild`*, mark that* `vbt` *has been moved by the user so the next time it is reshaped, it will retain its current position relative to its parent. If* `vbt` *isn't a* `ZChildVBT`*, this procedure is a no-op.* `VBT.Parent(vbt)` *must be a subclass of* `ZSplit`*.*

```
PROCEDURE Grow (vbt: VBT.T; w, h: INTEGER);
```

Call this procedure after the size of `vbt` *has been changed to* `w`*-by-*`h` *(in pixels) by a user. If* `vbt` *is a* `ZChild`*, this procedure marks that* `vbt` *has a new shape and calls* `VBT.NewShape` *to tell the parent* `ZSplit`*. From here on,* `vbt` *will report its shape as* `w`*-by-*`h`*. If* `vbt` *is not a* `ZChildVBT`*, this procedure is a no-op.* `VBT.Parent(vbt)` *must be a subclass of* `ZSplit`*.*

```
END ZChildVBT.
```

6.3 The ZBackgroundVBT Interface

A `ZBackgroundVBT` is a filter that should be put around the background child of a `ZSplit`. This filter will clip highlighting that takes place within the background child from the other children of the `ZSplit`. The implementation is merely a `HighlightVBT`, but it's easier to remember the purpose of that highlighter if it's called a `ZBackgroundVBT` instead.

```
INTERFACE ZBackgroundVBT;

IMPORT HighlightVBT;

TYPE
  T = HighlightVBT.T BRANDED OBJECT END;

END ZBackgroundVBT.
```

6.4 The ZMoveVBT Interface

A `ZMoveVBT` is a switch that has the side-effect of repositioning its nearest ancestor that's a non-background child of a `ZSplit`. In other words, a `ZMoveVBT` should be a descendant of a non-background child of some `ZSplit`; call the child `ch`. The `ZMoveVBT` has the side-effect of moving `ch`.

If the initial mouse click is unshifted, child `ch` is lifted to the top of its sibling; otherwise, child `ch` keeps its current top-to-bottom ordering among its siblings. As the mouse is moved, the cursor is changed to give appropriate feedback, and an outline of child `ch` is moved to show where `ch` will be repositioned on an uncancelled upclick. On an uncancelled upclick, the outline is removed and the child is moved if it isn't over an active target. The outline is also removed on a chord-cancel.

As the outline is being dragged, whenever the mouse is over a "target" (as in `SourceVBT.Target`) and the `hit` method returns `TRUE`, the outline is removed.

```
INTERFACE ZMoveVBT;

IMPORT Rect, SourceVBT;

TYPE T <: SourceVBT.T;

PROCEDURE MoveAndHighlight (v: T; READONLY rect: Rect.T);
```

Move the outline of v's rect to `rect`. Highlighting is done in the HighlightVBT returned by `SourceVBT.GetHighlighter(v)`. This procedure is particularly useful to the `during` method of subclasses, such as ZGrowVBT.

```
END ZMoveVBT.
```

6.5 The ZGrowVBT Interface

A `ZGrowVBT` is a switch that has the side effect of reshaing its nearest ancestor that's a non-background child of a `ZSplit`. If the initial mouse click is unshifted, the child is lifted to the top of its sibling; otherwise, the child keeps its current top-to-bottom ordering among its sibling. As the mouse is moved, the cursor changes to give appropriate feedback and an outline of the child is displayed to show the shape the child will acquire on an uncancelled upclick. The shape of the child is not actually changed until an uncancelled upclick. The outline is removed on an uncancelled upclick or on a chord-cancel.

```
INTERFACE ZGrowVBT;

IMPORT Feedback, ZMoveVBT;

TYPE
  T <: Public;
  Public =
    ZMoveVBT.T OBJECT
    METHODS
      init (f: Feedback.T): T;
    END;

END ZGrowVBT.
```

6.6 The ZSplitUtils Interface

The `ZSplitUtils` interface contains utility procedures for working with `ZSplit`s. The `ZChildVBT` interface contains some additional utility procedure that are oriented for children of `ZSplit`s that are used as "subwindows".

```
INTERFACE ZSplitUtils;

IMPORT VBT;

PROCEDURE FindZChild (v: VBT.T): VBT.T;
```

Return the lowest (possibly improper) ancestor of `v` whose parent is a `ZSplit.T` and who is not the `ZSplit.T`'s background child. If no such `VBT` is found, return `NIL`. There's a good chance that the `VBT` returned is a `ZChildVBT.T`, but this is not required.

```
END ZSplitUtils.
```

6.7 The ZTilps Interface

A `ZTilps.T` multi-split is like a `ZSplit`, except that the children are stored from bottom to top. Thus, `Multi.Nth(v,0)` returns the background child of the `ZSplit`.

```
INTERFACE ZTilps;

IMPORT MultiClass, ZSplit;

TYPE
  T <: Public;
  Public = ZSplit.T OBJECT
           METHODS
             init (saveBits       := FALSE;
                   parlim         := -1;
                   multiclass: MC := NIL    ): T
           END;
```

The call `v.init(...)` initializes `v` as a `ZTilps` and returns `v`. See the `ZSplit` interface for a description of `saveBits` and `parlim`,

```
TYPE MC <: MultiClass.T;
```

The type MC overrides the `replace`, `insert`, `pred`, and `succ` methods of the type MultiClass.T.

```
END ZTilps.
```

7 Miscellaneous Leaf VBTs

7.1 The ListVBT Interface

A `ListVBT` defines a `VBT` class for displaying a list (or table) of items. Each item is in a *cell*. All cells are the same size. They are displayed in a single vertical column, with a scrollbar.

The `ListVBT` itself deals with the details of being a `VBT`, maintains a table that maps a cell-number to a cell-value, and maintains the *selection*, a distinguished subset of the cells. It uses subsidiary objects to handle the details of what cells look like on the screen (`Painter`), and how the list responds to mouse clicks (`Selector`).

This interface contains basic versions of each of the subsidiary objects:

- `TextPainter`, which treats cells' values as `TEXT` and paints them.
- `UniSelector`, which maintains at most one selected cell, adjusted by mouse clicks.

- `MultiSelector`, which uses mouse clicks for selection, but permits multiple cells to be selected. The client can subclass these, or provide entirely different ones.

A client that wishes to take actions in response to mouse clicks should subclass a `Selector`. Similarly, a client that wishes to display objects other than text strings should subclass `Painter`.

7.1.1 Locking levels

`ListVBT` is internally synchronized; it can safely be called from multiple threads. All `ListVBT.T` methods have `LL.sup < list`. In addition, `VBT.mu < list` for any `list` of type `ListVBT.T`.

`VBT` methods call `Selector` methods with `LL.sup = VBT.mu`. `Selector` methods are permitted to call `ListVBT.T` methods.

`ListVBT.T` methods call `Painter` methods with the `ListVBT.T`'s internal mutex held. `Painter` methods must not call any of the `ListVBT.T` methods; their locking level is such that `LL.sup = list`.

The `TextPainter` class uses its own internal lock for font information; `TextPainter.setFont(v,font)` has `LL.sup < v`.

7.1.2 The type `ListVBT.T`

```
INTERFACE ListVBT;

IMPORT Font, PaintOp, Rect, VBT;

TYPE Cell = INTEGER;
```

The number of a cell; the first cell-number is 0.

```
TYPE
  T <: Public;
  Private <: VBT.Split;
  Public = Private OBJECT
             painter : Painter  := NIL;
             selector: Selector := NIL;
           METHODS
             init            (colors: PaintOp.ColorQuad): T;
             setValue        (this: Cell; value: REFANY);
             getValue        (this: Cell): REFANY;
             count           (): CARDINAL;
             insertCells     (at: Cell; n: CARDINAL);
             removeCells     (at: Cell; n: CARDINAL);
             selectNone      ();
             selectOnly      (this: Cell);
```

```
    select           (this: Cell; selected: BOOLEAN);
    isSelected       (this: Cell): BOOLEAN;
    getAllSelected   (): REF ARRAY OF Cell;
    getFirstSelected (VAR this: Cell): BOOLEAN;
    scrollTo         (this: Cell);
    scrollToShow     (this: Cell);
  END;
```

In the following descriptions, `v` is an object of type `ListVBT.T`, and a value `n` is said to be *in range* if

$$0 \leq n < v.count()$$

`v.painter` is the list's painter; the client may read but not assign to this field, although the client may provide a value at allocation time. If the actual painter has methods allowing it to be modified, the client is welcome to call them, although the client and painter are then responsible for provoking any necessary repaints.

`v.selector` is the list's selector; client may read but not assign to this field, although the client may provide a value at allocation time. If the actual selector has methods allowing it to be modified, the client is welcome to call them, although the client and selector are then responsible for any necessary adjustments to the set of selected cells.

The call `v.init(colors)` initializes `v` as a `ListVBT` and returns `v`. It must be called before any other method. `colors` is passed intact to the scroller; `colors.fg` is used for a dividing bar. If `self.painter = NIL` when this method is called, `init` will allocate and initialize a `TextPainter`. If `self.selector = NIL`, `init` will allocate and initialize a `UniSelector`. Neither the painter nor the selector need have been initialized before this method is called. The list initially has no cells (and no selection).

In the call `v.setValue(this,value)`, if `this` is in range, then record `value` as the value of the cell `this`; otherwise do nothing.

In the call `v.getValue(this)`, if `this` is in range, then return the previously recorded value of the cell `this`; otherwise return `NIL`.

The call `v.count()` returns the number of cells in the list.

The call `v.insertCells(at,n)` inserts `n` cells, starting at

```
MAX (0, MIN (at, v.count()))
```

Previously existing cells at and beyond `at` are renumbered appropriately, and selections are relocated appropriately. The `VBT` will be repainted in due course. The new cells' values are all `NIL`, and they are not selected.

The call `v.removeCells(at, n)` removes all cells in the range

```
[MAX (0, MIN (at, v.count ())) ..
  MIN (at + n, v.count ()) - 1 ]
```

Subsequent cells are renumbered appropriately. The `VBT` will be repainted in due course.

The call `v.selectNone()` makes the set of selected cells be empty.

In the call `v.selectOnly(this)`, if `this` is in range, make the set of selected cells be exactly `this`; otherwise make the list of selected cells be empty. Equivalent to

```
v.selectNone(); v.select(this,TRUE)
```

In the call `v.select(this,selected)`, if `this` is in range and `selected` is `TRUE`, add `this` to the set of selected cells (without complaint if it's already selected); otherwise if `this` is in range and `selected` is `FALSE`, remove it from the set of selected cells (again without complaint). The `VBT` will be repainted as necessary in due course.

The call `v.isSelected(this)` returns `TRUE` if `this` is in range and is a selected cell; otherwise it returns `FALSE`.

The call `v.getAllSelected()` returns the set of selected cells. If there are none, it returns a non-`NIL` `REF` to an array of length 0.

The call `v.getFirstSelected(this)` assigns to `this` the lowest-numbered selected cell and returns `TRUE`; if there are no selected cells, it returns `FALSE`.

The call `v.scrollTo(this)` adjusts the list's scrolling position to place

```
MAX (0, MIN (this, v.count () - 1) )
```

at the top of `v`'s domain. `this < 0` is treated as `this = 0`.

The call `v.scrollToShow(this)` adjusts the list's scrolling position to make `this` visible.

7.1.3 The Painter

Here is the definition of a `Painter`. In the comments about its methods, `v` is the `VBT` in which the painting is to take place; it is the `ListVBT.T` or a descendant of it. Recall that `LL.sup = list` for all methods, other than `init`.

```
TYPE
  Painter = OBJECT
            METHODS
              init   (): Painter;
              height (v: VBT.T): INTEGER;
              paint  (v       : VBT.T;
                      r       : Rect.T;
                      value   : REFANY;
                      selected: BOOLEAN;
                      bad     : Rect.T   );
              select (v       : VBT.T;
                       r       : Rect.T;
                       value   : REFANY;
                       selected: BOOLEAN );
              erase  (v: VBT.T; r: Rect.T);
            END;
```

The call `p.init()` initializes `p` as a `Painter` and returns `p`.

The call `p.height(v)` returns the pixel height of each cell if painted in `v`. The list caches the result of this call, so it needn't be very efficient. It is called only when the list has a non-empty domain. It gets re-evaluated whenever the list's screen changes.

The call `p.paint(v, r, value, select, bad)` paints the cell in the given rectangle (whose height will equal that returned by `p.height()`, and some part of which will be visible). If `selected` is `TRUE`, highlight the painted cell to indicate that it is in the set of selected cells. `bad` is the subset of `r` that actually needs to be painted; `bad` is wholly contained in `r`.

The call `p.select(v, r, value, selected)` changes the cell's highlighting, according to `selected`, to show whether it is in the set of selected cells. The cell has previously been painted; its selection state has indeed changed. It's OK for this method to be identical to `paint`, but it might be more efficient or cause less flicker, e.g. by just inverting `r`.

The call `p.erase(v, r)` paints the given rectangle to show that it contains no cells. Typically, this just fills it with the background color used when painting cells.

Perhaps the most common type of `Painter` is a `TextPainter`. It displays cells whose values are text strings. Here is its public definition:

```
TYPE
  TextPainter <: TextPainterPublic;

  TextPainterPublic =
    Painter OBJECT
    METHODS
      init (bg       := PaintOp.Bg;
            fg       := PaintOp.Fg;
            hiliteBg := PaintOp.Fg;
            hiliteFg := PaintOp.Bg;
            font     := Font.BuiltIn): TextPainter;
      setFont (v: VBT.T; font: Font.T); <* LL.sup < v *>
    END;
```

The call `p.init(...)` initializes `p` as a `TextPainter` and returns `p`. Unselected cells are painted with `fg` text on `bg`; selected cells are painted with `hiliteFg` text on `hiliteBg`; erased areas are painted with `bg`. Text is drawn using `font`.

After the call `p.setFont(v, font)`, the `TextPainter` uses `font` for subsequent painting of values; the call also marks `v` for redisplay. `v` should be the relevant `ListVBT.T`.

7.1.4 The Selector

Here is the definition of `Selector`. Recall that for all methods (other than `init`) `LL.sup = VBT.mu`.

```
TYPE
```

```
  Selector = OBJECT
             METHODS
               init         (v: T): Selector;
               insideClick  (cd: VBT.MouseRec; this: Cell);
               outsideClick (cd: VBT.MouseRec);
               insideDrag   (cd: VBT.PositionRec; this: Cell);
               outsideDrag  (cd: VBT.PositionRec);
             END;
```

The call `s.init(v)` initializes `s` as a `Selector` and returns `s`. The `ListVBT v` need not have been initialized before this method is called.

`s.insideClick(cd, this)` is called on a `FirstDown` mouse click inside the cell, or on any mouse click inside the cell while we have the mouse focus. On any click other than `LastUp`, the list itself has set a cage so that it receives position reports during subsequent drags.

`s.outsideClick(cd)` is called when there is a `FirstDown` click in the `ListVBT` that is not in a cell, or on any mouse click not in a cell while we have the mouse focus. On any click other than `LastUp`, the list itself has set a cage so that it receives position reports during subsequent drags.

`s.insideDrag(cd)` is called if the list has received a `FirstDown` click and a subsequent position report with the mouse not in any cell. The list itself has set a cage so that it receives further position reports.

`s.outsideDrag(cd)` is called if the list has the mouse focus and receives a subsequent position report with the mouse in this cell. The list itself has set a cage so that it receives further position reports.

One common class of `Selector` is a `UniSelector`. It maintains the invariant that there is at most one selected cell. On an `insideClick` firstDown, or an `insideDrag`, it removes any previous selection then selects this cell. Its other methods do nothing. Here is its declaration:

```
TYPE
  UniSelector <: Selector;
```

The other common class of `Selector` is `MultiSelector`. It permits multiple cells to be selected. On an `insideClick` firstDown, it remembers this cell as the *anchor*; if this is not a shift-click, it calls `selectNone` and inverts the selection state of this cell. On an `insideDrag`, it makes the selection state of all cells between this cell and the anchor be the same as that of the anchor. Here is its declaration:

```
TYPE
  MultiSelector <: Selector;

END ListVBT.
```

7.2 The FileBrowserVBT Interface

A `FileBrowserVBT` displays the files in a directory, and allows the user to traverse the file system and to select one or more files. There are two additional widgets that can be associated with a `FileBrowserVBT`. A *helper* is a type-in field that displays the pathname of the directory and allows the user to type new pathnames. A *directory-menu* is a menu containing the names of each level in the directory tree, with the root at the bottom; you can go to any level in the tree by selecting the appropriate item in the menu.

There are two user actions, selecting and activating.

- The user may *select* items, either by single-clicking on an item to select just that one, or by single-clicking and dragging to select a range. Shift-clicking adds to the selection. A change in selection is reported to the client by invoking the `selectItems` method. The client can read the current selection with `GetFile` or `GetFiles`.

- The user may *activate* an item, either by double-clicking on it, or by typing its name in the helper followed by a carriage return.

  Activation of a *file* is reported to the client by invoking the `activateFile` method, whose default is a no-op.

  Activation of a *directory* is reported by invoking the `activateDir` method, whose default behavior is to call `Set` to display the activated directory.

  The client can distinguish between a double-click and a carriage return by looking at the `AnyEvent.T` passed to the activation method. A double-click will be reported as an `AnyEvent.Mouse`, and the carriage return will be an `AnyEvent.Key`.

Directories are indicated in the display by showing a slash ("/") after the name.

A background thread calls `Refresh(v)` for every open filebrowser `v`, once per second, to see whether it needs to be updated (although the distributed filesystem may cause a substantial delay before the change is noticed). The filebrowser's `redisplay` method also calls `Refresh`.

```
INTERFACE FileBrowserVBT;

IMPORT AnchorSplit, AnyEvent, Font, ListVBT, PaintOp, Shadow,
       TextList, TextPort, VBT;

TYPE
  T <: Public;
  Public =
    ListVBT.T OBJECT
    METHODS
      <* LL.sup <= VBT.mu *>
      init (font                      := Font.BuiltIn;
```

```
        colors: PaintOp.ColorQuad := NIL          ): T;
    <* LL = VBT.mu *>
    selectItems  (event: AnyEvent.T);
    activateFile (filename: TEXT; event: AnyEvent.T);
    activateDir  (dirname: TEXT; event: AnyEvent.T);
    error        (err: E);
  END;
```

The call `v.init(...)` initializes `v` as a `FileBrowserVBT`. If `v.painter` is a subtype of `ListVBT.TextPainter`, `init` calls `v.paint.setFont(font)`. The `selector` field must be either `NIL` (in which case a new selector is created) or a subtype of `FileBrowserVBT.Selector`. The initial state of the filebrowser is the current working directory, as returned by `UnixUtils.GetWD`. If that directory is not readable, '/' is used.

The implementation calls `v.selectItems(...)` when the user changes the selection using the mouse.

The implementation calls `v.activateFile(filename, event)` either when the user double-clicks on a file in the browser (in which case `filename` will be the first selected item), or when the user types Return in the helper (in which case `filename` will be set to the tail of the contents of the helper). To find the full pathname, use

```
GetDir(self) & "/" & filename
```

Don't forget that if `activateFile` is being called because of a double-click, multiple files might be selected in the browser, even though you are given only one in the `filename` parameter.

The implementation calls `v.activateDir(...)` when a new directory is activated. The normal action is simply to set `v` to view that directory, relative to `GetDir(self)`. If an error occurs during the activation, the `error` method is invoked.

The implementation calls `v.error(...)` when an error occurs during user action in `v`, and the `Error` exception cannot be raised (e.g., because it happened in a separate thread). Some examples of errors are as follows: the user has typed a nonexistent directory in the path; the current directory has become inaccessible; the user has no permission to read the directory. By default the response to such errors is that nothing happens. By overriding this method, the client can provide better information to the user.

```
TYPE Selector <: ListVBT.MultiSelector;
```

If you create a subtype of `FileBrowserVBT` *(which is a subtype of* `ListVBT.T`*), and you specify a selector for it, it must be a subtype of* `Selector`.

```
EXCEPTION Error (E);
TYPE
  E = OBJECT
        v         : T;
        text, path: TEXT := ""
```

```
    END;
```

The argument to the `Error` *exception includes the* `FileBrowserVBT` *itself, along with a descriptive message and the pathname in question when the error occurred.*

7.2.1 The Helper

The FileBrowser's helper (see page **??**) is a single-line `TextPort`. Once the user types in the helper, any selected items in the browser are unselected. If the user types Return in the broswer, that will activate the name in the Helper.

If an error occurs during the activation, the `error` method of the filebrowser to which the helper is attached will be invoked. Here's the definition of a helper:

```
TYPE
  Points = REAL;
  Helper <: PublicHelper;
  PublicHelper =
    TextPort.T OBJECT
    METHODS
      init (hMargin: Points   := 1.5;
            vMargin: Points   := 1.5;
            font              := Font.BuiltIn;
            shadow : Shadow.T := NIL           ): Helper;
    END;
```

The arguments to `init` are passed to `TextPort.T.init`. The type `Helper` overrides the `filter` and `modified` methods of the type `TextPort.T`.

```
PROCEDURE SetHelper (v: T; helper: Helper) RAISES {Error};
```

Sets the helper for `v` *to be* `helper`*, and fills it with* `GetDir(v)`.

7.2.2 The Directory-Menu

The directory-menu is described on page 60.

```
TYPE
  DirMenu <: PublicDirMenu;
  PublicDirMenu =
    AnchorSplit.T OBJECT
    METHODS
      init (font              := Font.BuiltIn;
            shadow: Shadow.T := NIL;
            n     : CARDINAL := 0              ): DirMenu
    END;
```

The `font` and `shadow` control the appearance of the text within the menu. As usual, if `shadow` is `NIL`, then `Shadow.None` is used instead. The parameter `n` is used by `AnchorSplit` to determine the `ZSplit` in which to install the menu.

```
PROCEDURE SetDirMenu (v: T; dm: DirMenu);
```

Sets the directory-menu of `v` *to be* `dm`.

7.2.3 FileBrowser options

```
PROCEDURE SetReadOnly (v: T; readOnly: BOOLEAN);
```

If `readOnly` *is* `TRUE`*, then in subsequent calls to* `v.activateFile(filename)`, `filename` *is guaranteed to exist.* `activateFile` *methods is guaranteed to exist. Otherwise, the user can type the name of a non-existing file into the helper. A newly initialized* `FileBrowserVBT` *is not read-only.*

```
PROCEDURE SetSuffixes (v: T; suffixes: TEXT);
```

`suffixes` *provides a filter on the files to be displayed. By default all files in the directory are displayed, but if* `suffixes` *is not the empty string, only files with the specified suffixes (and all directories) will be displayed. The format of* `suffixes` *is a sequence of suffixes (not including the period) separated by non-alphanumeric characters (e.g., spaces). The special suffix* `$` *indicates "files with no suffix". Calling this procedure does not force* `v` *to be redisplayed.*

7.2.4 Setting the displayed directory

```
PROCEDURE Set (v: T; pathname: TEXT; time: VBT.TimeStamp := 0)
  RAISES {Error};
```

Set the display state of v.

The `pathname` may be absolute or relative; if it's relative, it is relative to the current displayed directory. `pathname` is absolute if

```
Text.GetChar(Filename.ExpandTilde(pathname), 0) = '/'
```

If `pathname` refers to a non-existent or inacessible directory, `Error` will be raised. The exception will also be raised if `pathname` refers to a non-existent file and `v` is read-only.

If `time` is not zero and there is a helper, then the helper will take the keyboard focus and will display its new contents in replace-mode, ready for the user to type something in its place.

```
PROCEDURE Unselect (v: T);
```

Put `v` *into the no-selection state, without changing the current directory. Equivalent to* `v.selectNone()`.

```
PROCEDURE Refresh (v: T) RAISES {Error};
```

Update the display without changing the directory. If `v`*'s domain is not empty, and its directory has been* `Set`*, and the directory has changed since the last time it was displayed, then* `v` *will be marked for redisplay.* `Error` *is raised only if the directory has become inaccessible for some reason; in this case, the browser goes to the empty state, so that if the client catches* `Error` *and takes no other action, the browser will be empty but not broken.*

7.2.5 Retrieving from the browser

```
PROCEDURE GetFile (v: T; shortName := FALSE; normalize := TRUE):
  TEXT RAISES {Error};
```

Return the first item selected, or the empty string if no files are selected.

```
PROCEDURE GetFiles (v: T; shortName := FALSE; normalize := TRUE):
  TextList.T RAISES {Error};
```

Return the currently selected files of `v`*, or* `NIL` *if none are selected. The list includes full pathnames unless* `shortName` *is* `TRUE`*, in which case it includes only the name relative to the current displayed directory. Selected objects that are directories will have a "/" on the end.*

Normalizing is done during a call to `Set` and when the user types a carriage return in the helper. It consists of expanding a tilde if present, and checking to see that all directories in the pathname really exist and are accessible. It should be done during `GetFiles` if and only if that `GetFiles` is a synchronous result of some user action, e.g., pressing a button or invoking a menu command. Normalizing can lead to the discovery of errors, raising `Error`; however, no exceptions are raised if `normalize` is `FALSE`.

```
PROCEDURE GetDir (v: T): TEXT;
```

Return the current displayed directory of `v`*. Always reports a full pathname, excluding a "/" on the end. Returns an empty string if* `v` *is in the "empty" state.*

```
PROCEDURE IsDir (filename: TEXT): BOOLEAN;
```

A convenience procedure to test for a "/" on the end of `filename`*, indicating that it is a directory. It does not actually query the filesystem; it only looks at the form of the name.*

```
END FileBrowserVBT.
```

7.3 The NumericVBT Interface

A `NumericVBT` is a `VBT` class for displaying and changing an integer within some range. A `NumericVBT` has three parts (from left to right): a minus button, a type-in field, and a plus button. The type-in field is restricted to be an integer within the range `[min ..`

`max]`; it can be changed by editing or typing carriage return, or by clicking on the plus or minus buttons. The plus/minus buttons are trill buttons, so clicking and holding will cause the value of the `NumericVBT` to continuously increment/decrement.

The `NumericVBT` has a method that is called each time the user changes the value.

```
INTERFACE NumericVBT;

IMPORT AnyEvent, Filter, Font, Shadow, TextPort, VBT;

TYPE
  T <: Public;
  Public = Filter.T OBJECT
           METHODS
             init (min       : INTEGER;
                   max       : INTEGER;
                   allowEmpty: BOOLEAN;
                   naked     : BOOLEAN;
                   font      : Font.T;
                   shadow    : Shadow.T ): T;
             callback (event: AnyEvent.T);
           END;
```

The call to `v.init(...)` initializes `v` as a `NumericVBT` and returns `v`. The integer stored with `v`, referred to as *the value in* `v`, is constrained to be in the range

```
[min .. MAX (min, max)]
```

The initial value in `v` is equal to `min`. If `allowEmpty` is `TRUE`, then "empty" (no text in the type-in area) is a distinct and valid state, and can be tested by `IsEmpty`. The call `Get(v)` in the empty state will return `FIRST(INTEGER)`, regardless of whether this is in the valid range. For `NumericVBT`s of limited range, this may be useful as a simple out-of-band signal that doesn't require the special `IsEmpty` test. If `naked` is `TRUE`, then the numeric interactor appears as just a type-in field, without plus or minus buttons. The values of `font` and `shadow` are used for displaying the type-in area and the buttons.

The implementation calls `v.callback(event)` when the user changes the value in `v`. The `event` parameter reports the details of the event as either an `AnyEvent.Mouse` (clicking on the plus/minus buttons) or an `AnyEvent.Key` (typing carriage return in the type-in area). The value in `v` is changed before `v.callback` is invoked.

```
PROCEDURE Put (v: T; n: INTEGER);
```

Change the value in v to be MIN(v.max, MAX(v.min, n)) and display this value in the type-in area. Note that `v.callback` is not invoked; the value in v has changed, but not due to a user gesture.

```
PROCEDURE Get (v: T): INTEGER;
```

Return the value in v.

```
PROCEDURE PutMin (v: T; minVal: INTEGER);
```

Change `v.min` to be `minVal` and then call `Put(v, v.value)`. The call to `Put` has the effect of projecting the value of v into the new lower bound.

```
PROCEDURE GetMin (v: T): INTEGER;
```

Return `v.min`.

```
PROCEDURE PutMax (v: T; maxVal: INTEGER);
```

Change `v.max` to be `maxVal` and then call `Put(v, v.value)`. The call to `PutMax` has the effect of projecting the value of v into the new upper bound.

```
PROCEDURE GetMax (v: T): INTEGER;
```

Return `v.max`.

```
PROCEDURE SetEmpty (v: T);
```

Set v to the empty state. This is a no-op unless `allowEmpty` was `TRUE` when v was initialized.

```
PROCEDURE IsEmpty (v: T): BOOLEAN;
```

Test whether v is in the empty state. If `allowEmpty` was not `TRUE` when v was initialized, this procedure will always return `FALSE`.

```
PROCEDURE TakeFocus (v          : T;
                     time       : VBT.TimeStamp;
                     alsoSelect                  := TRUE):
  BOOLEAN;
```

Cause the type-in area to grab the keyboard focus. If the focus could be grabbed and if `alsoSelect` is set, the type-in area will make its entire text the primary selection. Returns whether the keyboard focus could be acquired.

```
PROCEDURE GetTypein (v: T): TextPort.T;
```

Return v's type-in area. Clients should manipulate the `TextPort` with extreme care.

```
END NumericVBT.
```

7.4 The PixmapVBT Interface

A `PixmapVBT.T` is a `VBT` that displays a pixmap, possibly colored. Its preferred and minimum sizes are the size of the pixmap, and its maximum size in each axis is very large.

```
INTERFACE PixmapVBT;

IMPORT VBT, PaintOp, Pixmap;

TYPE
  T <: Public;
  Public =
    VBT.Leaf OBJECT
    METHODS
      <* LL.sup <= VBT.mu *>
      init (pm: Pixmap.T;
            op := PaintOp.BgFg;
            bg := PaintOp.Bg): T
    END;
```

The call `v.init(...)` initializes `v` as a `PixmapVBT` displaying pixmap `pm` using the paint-op `op`, and returns `v`. Whenever `v` is made larger than the pixmap `pm`, the pixmap is centered and the background is painting using the tint `bg`.

```
PROCEDURE New (pm: Pixmap.T; op := PaintOp.BgFg; bg := PaintOp.Bg): T;
<* LL.sup <= VBT.mu *>
```

`New(...)` is equivalent to `NEW(T).init(...)`.

```
PROCEDURE Put (v: T; pm: Pixmap.T);
<* LL.sup = VBT.mu *>
```

Change the pixmap displayed by `v` to `pm`, and mark `v` for redisplay. If the size of the pixmap has changed, notify `v`'s parent that its shape has changed.

```
PROCEDURE SetColors (v: T; op: PaintOp.T; bg := PaintOp.Bg);
<* LL.sup = VBT.mu *>
```

Change the `op` and `bg` of `v` and mark `v` for redisplay.

```
END PixmapVBT.
```

7.5 The ScrollerVBT Interface

A `ScrollerVBT` is a scrollbar with an orientation along an *axis*. (For the sake of brevity in this interface, we'll only talk about vertical scrollers. For horizontal scrollers, replace phrases like "top and bottom edges" by "left and right edges", and so on.) The scrollbar contains a *stripe* that spans some fraction of the height of the scrollbar and is slightly narrower than the scrollbar.

Various user-gestures cause methods on the `ScrollerVBT` to be invoked; typically, these methods will move the stripe vertically and possibly change its length.

Clients may find it easier to use the `NumericScrollerVBT` subclass. That provides a "bounded value" abstraction with reasonable user-gestures to change the value (displayed visually by the location of the stripe within the scrollbar) and move the stripe accordingly. Also, it introduces a single "callback" method that is invoked whenever the user changes the value.

Although each `VBT` class that uses a `ScrollerVBT` is free to associate any meaning with the length of the stripe, the following conventions are suggested for using scrollbars to view an object:

> The ratio of the height of the stripe to the height of the scrollbar should be the same as the ratio of the amount of the object visible vertically to its entire height. The position of top of the stripe reflects the position of top of the view of the object within the entire object.

Here is some terminology and user-interface guidelines:

- To *scroll* means to left-click or right-click in the scrollbar.
- You need to release the button ("upclick") relatively quickly, or else you'll start *continuous scrolling*. You stop continuous scrolling by releasing the button, by *chord-cancelling* (clicking on any other mouse button), or by moving the mouse.
- When you move the mouse, you are then using *proportional scrolling*. This means that the more pixels that you move the mouse vertically, the more the stripe should be moved in the direction of the mouse movement. You stop proportional scrolling by upclicking or chord-cancelling.
- In general, the left and right buttons should be inverses: the left button should move the stripe downward and the right button should move the stripe upward.
- You *thumb* with a middle-click. The top of the stripe should be moved to the position of the cursor. Thus, middle-click above the top of the stripe moves the stripe up, and middle-click below the top moves the stripe down.
- Middle-drag causes *continuous thumbing*. As you drag to a new position, the top of the stripe should be moved to match the current cursor position. You stop continuous thumbing by middle-upclicking or chord-canceling.

```
INTERFACE ScrollerVBT;

IMPORT Axis, PaintOp, Pixmap, VBT;

TYPE
  Points = REAL;
  T <: Public;
  Public = VBT.Leaf OBJECT
           METHODS
             <* LL.sup = VBT.mu *>
             init (axis: Axis.T; colors: PaintOp.ColorQuad): T;
             <* LL = VBT.mu *>
             scroll (READONLY cd        : VBT.MouseRec;
                              part      : INTEGER;
                              height    : INTEGER;
                              towardsEOF: BOOLEAN       );
             autoScroll (READONLY cd           : VBT.MouseRec;
                                  linesToScroll: CARDINAL;
                                  towardsEOF   : BOOLEAN       );
             thumb (READONLY cd    : VBT.MouseRec;
                             part  : INTEGER;
                             height: INTEGER       );

           END;
```

The call to `v.init(axis, colors)` initializes `v` as a `ScrollerVBT` in the `axis` orientation, and returns `v`. It is displayed using `colors`.

The default methods for `scroll`, `autoScroll`, and `thumb` are no-ops: the stripe within the scroller doesn't change.

When the user scrolls, the implementation calls

```
v.scroll(cd, part, height, towardsEOF)
```

on the up-click. `cd` is the mouse event; `height` is the number of pixels in the domain of `v` in the `v.axis` orientation. `part` is number of pixels away from the top/left edge that the upclick happened. `towardEOF` is `TRUE` when invoked from a left-upclick, `FALSE` when invoked from a right-upclick. (Of these, only `cd` is really needed; the others can be computed from `v` and `cd`.)

While the user is in continuous or proportional scrolling, the implementation calls `v.autoScroll(...)` repeatedly. The `linesToScroll` is somewhat of a misnomer (but kept for historical purposes). For continuous scrolling, the value is always 1. For proportional scrolling, the value is the number of pixels the mouse has moved. Think of `linesToScroll` as simply the "amount that should be scrolled." The `cd` field in proportional scrolling is fine, except it's really a position event, not a mouse event, that caused the action (this is a good use for an `AnyEvent`, but for historical reasons...). For continuous scrolling, `cd` is set to be the mouse record for the down-click that initiate the scrolling, but with `cd.time = 0`.

Finally, `v.thumb(cd, part, height)` is called when the user thumbs or continuous thumbs. `height` is the number of pixels in the domain of `v` in the `v.axis` orientation. `part` is the distance in pixels between the mouse and the top/left edge. The `cd` always has a valid time, cursor position, and modifier fields. (OK, it isn't the real event, since continuous thumbing is a position event whereas thumbing is a mouse event. Again, a good potential client of `AnyEvent`.)

By and large, these methods will change the position and size of the stripe. This is done using the following procedure:

```
PROCEDURE Update (v: T; start, end, length: CARDINAL);
<* LL.sup < v *>
```

Set new values of the stripe and mark v for redisplay. The coordinate system for `start`, `end`, and `length` is as follows (we'll consider just a horizontal scrollbar): The left edge of the domain of v is at coordinate 0, and the right edge consider to be at `length`. The stripe extends from `max(start,0)` to `max(min(end,length),start)`. The implementation will draw a stripe to represent these quantities, subject to constraints such as the minimum number of pixels that the length of the stripe.

The visual appearance of a `ScrollerVBT` is governed by the following data structures and procedures:

```
TYPE
  Attributes = RECORD
                 axis                : Axis.T;
                 margin              : Points;
                 scrollPaintOps      : ARRAY Axis.T OF PaintOp.T;
                 scrollPixmaps       : ARRAY Axis.T OF Pixmap.T;
                 minStripeLen        : Points;
                 stripeWidth         : Points;
                 stripePaintOps      : ARRAY Axis.T OF PaintOp.T;
                 stripePixmaps       : ARRAY Axis.T OF Pixmap.T;
                 stripeBorder        : Points;
                 stripeBorderPaintOp: PaintOp.T;
                 stripeBorderPixmap : Pixmap.T;
               END;
```

```
PROCEDURE GetAttributes (v: T): Attributes;
```

Return the attribute currently in effect for v.

```
PROCEDURE SetAttributes (v: T; READONLY a: Attributes);
<* LL.sup = VBT.mu *>
```

Change the attributes on v to be `a`. Mark v for redisplay and notify v's parent that its shape might have changed.

```
PROCEDURE Colorize (v: T; colors: PaintOp.ColorQuad);
<* LL.sup = VBT.mu *>
```

Sets the paint op of all the scroller's textures and borders to be `colors.bgFg`*. Mark v for redisplay.*

```
END ScrollerVBT.
```

7.6 The NumericScrollerVBT Interface

A `NumericScrollerVBT` provides a *bounded-value* abstraction using a `ScrollerVBT` for user interaction.

A bounded value is guaranteed to stay in the range `[min .. max-thumb]`. Visually, the value is represented by the position (left/top edge) of a stripe in the scroller, and thumb by the length of the stripe. The amount that value should change when continuous scrolling is called the *stepping* amount.

```
INTERFACE NumericScrollerVBT;

IMPORT Axis, PaintOp, ScrollerVBT, VBT;

TYPE
  T <: Public;
  Public = ScrollerVBT.T OBJECT
           METHODS
             <* LL.sup = VBT.mu *>
             init (axis  : Axis.T;
                   min   : INTEGER;
                   max   : INTEGER;
                   colors: PaintOp.ColorQuad;
                   step  : CARDINAL             := 1;
                   thumb : CARDINAL             := 0  ): T;
             <* LL = VBT.mu *>
             callback (READONLY cd: VBT.MouseRec);
           END;
```

The call to `v.init(...)` initializes `v` as a `NumericScrollerVBT` in the `axis` orientation. It is displayed using `colors`.

The implementation calls `v.callback(cd)` after `v`'s value has been changed by the user; it is not called when the value is changed as the result of calls to `Put` or `PutBounds`.

The type `NumericScrollerVBT.T` overrides the `scroll`, `autoScroll`, and `thumb` methods of the type `ScrollerVBT.T`.

```
PROCEDURE Put (v: T; n: INTEGER);
```

Change the value of v, projected to `[min .. max-thumb]`*. Mark v for redisplay.*

```
PROCEDURE PutBounds (v    : T;
                     min  : INTEGER;
                     max  : INTEGER;
                     thumb: CARDINAL  := 0);
```

Set the bounds and project v's value into `[min .. max-thumb]`*. Mark v for redisplay*

```
PROCEDURE PutStep (v: T; step: CARDINAL);
```

Change the amount that v's value should change while continous scrolling to `step`*. If* `step = 0`*, scrolling will be disabled.*

```
PROCEDURE Get      (v: T): INTEGER;
PROCEDURE GetMin   (v: T): INTEGER;
PROCEDURE GetMax   (v: T): INTEGER;
PROCEDURE GetThumb (v: T): CARDINAL;
PROCEDURE GetStep  (v: T): CARDINAL;
```

Return the current `value`*,* `min`*,* `max`*,* `thumb`*, and* `step`*.*

```
END NumericScrollerVBT.
```

7.7 The VBTSnap Interface

`VBTSnap` produces a screen dependent pixmap containing the window's current image.

```
INTERFACE VBTSnap;

IMPORT Image, VBT;

PROCEDURE Photo (v: VBT.T; widthMM, heightMM: REAL): Image.T;
<* LL = VBT.mu *>
```

Return a `widthMM` *by* `heightMM` *screen-independent image of* `v`*.*

In the current implementation, `v` will be rescreened and redisplayed at least twice within `Photo`: once to take the phote and once to restore it to its original appearance.

```
END VBTSnap.
```

7.8 The VBTAlbum Interface

`VBTAlbum` manages a photo album of snapshots of VBTs. The snapshots are taken by `VBTSnap`.

```
INTERFACE VBTAlbum;

IMPORT Axis, VBT;

TYPE
  T <: TPublic;
  TPrivate <: VBT.Split;
  TPublic = TPrivate OBJECT
            METHODS
              <* LL <= VBT.mu *>
              init (layoutAxis   : Axis.T;
                    size         : CARDINAL;
                    width, height: REAL       := 30.0): T;

              <* LL = VBT.mu *>
              add   (w: VBT.T);
              clear ();
            END;
```

The call `v.init(layoutAxis, size, width, height)` initializes `v` to be an `VBTAlbum.T`. A `VBTAlbut.T` is a `VBT.Split` that manages the layout of a series of images produced by taking snapshots of VBTs using `VBTSnap`. The images in `v` are all shaped `v.width` by `v.height` millimeters and laid out in a grid.

If `v.layoutAxis` is horizontal, then there are `v.size` images laid out horizontally in each row, with rows laid out top to bottom. If `v.layoutAxis` is vertical, then there are `v.size` images per column and the columns are laid out left to right.

The method call

```
v.add(w)
```

adds a new snapshot to the album. It occupies the next slot either vertically or horizontally (depending on `v.layoutAxis`).

The method call

```
v.clear()
```

removes all snapshots from the album.

```
END VBTAlbum.
```

8 Miscellaneous non-leaf VBTs

8.1 The FlexVBT Interface

The `FlexVBT.T` is a filter whose shape is based on a *natural* size with some *stretch* and *shrink*. If a natural amount is left unspecified, the stretch and shrink are applied relative to the `FlexVBT`'s child's size. If a stretch or shrink is left unspecified, 0 is assumed. All units are specified in points (see the `Pts` interface).

More precisely, the shape of a `FlexVBT.T` is computed as follows:

```
preferred := natural,
lo        := natural - shrink
hi        := natural + stretch + 1
```

However, if any of these values are left unspecified, a size range is computed consistent with its child's shape according to the following rules:

- if natural is missing, use child's preferred;
- if shrink is missing, use 0 if natural is set, or child's lo (but no more than v's preferred) otherwise;
- if stretch is missing, use 0 if natural is set, or child's hi (but no less than v's preferred) otherwise.

This interface is similar to `RigidVBT`, but more powerful in that one can specify a size based on a child's size and dynamically change the size specification. Also, it presents a slightly different model to the client: In `RigidVBT`, one thinks in terms of the low and high bounds of some range. Here, one thinks in terms of the amount the a value can stretch and shrink. Finally, don't forget that values in this interface are specified in points, whereas `RigidVBT` uses millimeters.

```
INTERFACE FlexVBT;

IMPORT Axis, Filter, VBT;
```

The following types are used to specify the shape of a `FlexVBT.T`:

```
CONST
  Large    = 99999.0;
  Missing  = -Large;
  Infinity = Large;

TYPE
  Points = REAL;
  SizeRange = RECORD natural, shrink, stretch: Points END;
  Shape = ARRAY Axis.T OF SizeRange;
```

Some useful shapes are defined at the end of this interface.

```
TYPE
  T <: Public;
  Public = Filter.T OBJECT
           METHODS
             init (ch: VBT.T; sh := Default): T
           END;
```

The call `v.init(...)` initializes `v` as a `FlexVBT` with child `ch` and shape specification `sh`. The default shape causes `v` to be a (costly) no-op: it will simply return the shape of its child as its own.

```
PROCEDURE New (ch: VBT.T; sh := Default): T;
```

`New(...)` *is equivalent to* `NEW(T).init(...)`.

```
PROCEDURE FromAxis (ch: VBT.T;
                    ax: Axis.T;
                    sh            := DefaultRange): T;
```

Return a `FlexVBT` *and whose shape specification in the* `ax` *dimension is* `sh` *and whose shape in the other dimension is that of* `ch`.

```
PROCEDURE Set (v: T; sh: Shape);
```

Change the shape of `v` *to* `sh`*, and notify* `v`*'s parent that* `v`*'s size has changed.*

```
PROCEDURE SetRange (v: T; ax: Axis.T; sr: SizeRange);
```

Change the shape of `v` *to* `sr` *along the* `ax` *axis, and notify* `v`*'s parent that* `v`*'s size has changed.*

The rest of this inteface defines some useful shapes: `Default` uses child's size info. `Fixed` uses child's preferred, removing all shrink and stretch. `Stretchy` uses child's preferred and shrink, giving infinite stretch.

```
CONST
  Default  = Shape{DefaultRange, DefaultRange};
  DefaultRange =
    SizeRange {natural := Missing,
               shrink  := Missing,
               stretch := Missing};

  Fixed   = Shape{FixedRange, FixedRange};
  FixedRange =
    SizeRange {natural := Missing,
               shrink  := 0.0,
               stretch := 0.0};
```

```
    Stretchy = Shape{StretchyRange, StretchyRange};
    StretchyRange =
      SizeRange {natural := Missing,
                 shrink  := Missing,
                 stretch := Infinity};

PROCEDURE RigidRange (natural: Points): SizeRange;
```

Return a `SizeRange` *with the specified natural amount and with no stretch or shrink. Equivalent to*

```
SizeRange {natural, 0.0, 0.0}
```

```
END FlexVBT.
```

8.2 The JoinVBT Interface

A `JoinVBT.T` allows many parent `VBT`s to view the screen of a single child `VBT`. One parent's view of the child's domain may overlap with another parent's view, and it might not cover the entire child domain. In this implementation, all of the parents must share an identical `VBT.ScreenType`.

Any portion of a parent screen that lies outside the `JoinVBT`'s screen will be not get painted, so this should be avoided. The screentype of the join is the common screentype of the parents.

Event-time operations on the child are performed on the event-time parent (the parent that last received a mouse click, or an arbitrary parent if none of the current parents got the last mouse click). When the event-time parent changes, all selections held by the old event-time parent are released (if the parents are on different Trestles). These event-time operations are handled by making the parent an `ETAgent` and inserting a filter between the parent and the join. When a parent receives a `VBT.Deleted` code, it removes itself (and the filter). Other miscellaneous codes are handled as an `ETAgent`.

The value of `v.parent` is `NIL` if the `JoinVBT v` has no parents; otherwise it is an arbitrary parent (or rather the filter below the parent—these will not be explicitly mentioned hereafter).

Those who are familiar with the rules for using concurrency while reformatting splits will note that two parents of a join may be reformatted concurrently. These reformat procedures may wish to call `MoveParent`, `Reshape`, or `Redisplay`. Therefore, an internal lock is used by the join to serialize these operations; their locking level is `LL <= VBT.mu` instead of `LL = VBT.mu`.

```
INTERFACE JoinVBT;

IMPORT ETAgent, Filter, Rect, VBT;

TYPE
  T <: Public;
```

```
  Public = Filter.T OBJECT
           METHODS
             <* LL.sup <= VBT.mu *>
             init (ch: VBT.T): T
           END;
```

The call `v.init(ch)` initializes `v` as a `JoinVBT` with child `ch`, and returns `v`.

```
TYPE
  Points = REAL;
  ParentT <: ParentPublic;
  ParentPublic =
    ETAgent.T OBJECT
    METHODS
      <* LL.sup <= VBT.mu *>
      init (join: T; north, west: Points := 0.0): ParentT;
      <* LL.sup = VBT.mu *>
      translate (READONLY r: Rect.T): Rect.T;
    END;
```

The call `v.init(join, north, west)` initializes `v` as a `JoinVBT.ParentT` with child `join`, and returns `v`.

The northwest corner of the parent's domain is positioned at point `(north, west)` in `join`'s domain; this will be maintained as the parent is reshaped or rescreened.

For a given parent `p`, `p.translate(r)` returns the rectangle in `p`'s coordinate space corresponding to `r` in the coordinate space of the child.

```
PROCEDURE New (ch: VBT.T): T;
<* LL.sup <= VBT.mu *>
```

`New(...)` is equivalent to `NEW(T).init(...)`.

```
PROCEDURE NewParent (v: T; north, west: Points := 0.0): ParentT;
<* LL.sup <= VBT.mu *>
```

`NewParent(...)` is equivalent to `NEW(ParentT).init(...)`.

```
PROCEDURE RemParent (v: T; prnt: ParentT);
<* LL = VBT.mu *>
```

Cause v to lose parent `prnt`, which is reshaped to `Rect.Empty`. Also, v is marked for redisplay since `RemParent` can change the screentype of the join. It is a checked runtime error if `prnt` is not a parent of v.

```
PROCEDURE Child (prnt: ParentT): T;
<* LL = VBT.mu *>
```

Return the JoinVBT for which `prnt` is a parent.

```
PROCEDURE MoveParent (v: T; prnt: ParentT; north, west: Points);
<* LL = share(VBT.mu) *>
```

Change the position in v's domain of the northwest corner of parent `prnt` to be `(north, west)` and mark `v` for redisplay. Note that this procedure has a locking level that allows it to be called from within a parent's `reshape` or `rescreen` method.

```
PROCEDURE Succ (v: T; prnt: ParentT): ParentT;
<* LL = VBT.mu *>
```

Return the successor to parent `prnt` of `v`, in front-to-back order. It is a checked runtime error if `prnt` is not a parent of `v`. As with `Split.Succ`, the successor of `NIL` is the first parent; the successor the last parent is `NIL`; and the successor of `NIL` is `NIL` if `v` has no parents. Note that `Succ(v, NIL), Succ(v, Succ(v, NIL)), ...` is an enumeration of the parents of `v` in front-to-back order.

```
END JoinVBT.
```

8.3 The ReactivityVBT Interface

A `ReactivityVBT` is a filter that can make its child active, passive, dormant, and invisible. The state *Active* does nothing; mouse and keyboard events are relayed to child. *Passive* doesn't allow mouse or keyboard events to go to the child; in addition, the cursor is changed to `Cursor.NotReady`. *Dormant* doesn't send mouse or keyboard events to the child; it also grays out the child. *Vanish* also doesn't send mouse or keyboard events to go to the child; in addition, it draws over the child in the background color, thereby making it invisible.

If a descendant of a `ReactivityVBT` is painted, it will appear correctly. That is, if the `ReactivityVBT` is in the vanished state, the descendant will not appear until the state changes, or if the `ReactivityVBT` is in a dormant state, the descendant will be grayed out.

A `ReactivityVBT` also has the effect of passing on any miscellanous events to take the keyboard focus to the descendant that last acquired the keyboard focus.

```
INTERFACE ReactivityVBT;

IMPORT ETAgent, PaintOp, Rect, VBT;

TYPE
  State = {Active, Passive, Dormant, Vanish};
  T <: Public;
  Public =
    ETAgent.T OBJECT
    METHODS
```

```
    <* LL.sup = VBT.mu *>
    init (ch: VBT.T; colors: PaintOp.ColorScheme := NIL): T;
    <* LL = VBT.mu *>
    paintDormant (r: Rect.T; colors: PaintOp.ColorScheme);
  END;
```

The call `v.init(..)` initializes `v` as a `ReactivityVBT` with child `ch` and with an initial state of `Active`. If `colors` is `NIL`, then `PaintOp.bgFg` is used instead. The `colors` are used to draw the vanished and dormant states.

The implementation calls `v.paintDormant(r, colors)` to paint the part of `ch` bounded by rectangle `r` when `v`'s state is `Dormant`. The "current colors" of `v` are passed as `colors`. Initially, the current colors are those that were specified when the `ReactivityVBT` was initialized. They can be changed using the `SetColors` procedure. The default method paints a `Pixmap.Gray` texture using `colors.transparentBg`.

```
PROCEDURE New(ch: VBT.T; colors: PaintOp.ColorScheme := NIL): T;
```

`New(...)` is equivalent to `NEW(T).init(...)`.

```
PROCEDURE Set (v: T; state: State);
```

Change v's state and mark v for redisplay.

```
PROCEDURE Get (v: T): State;
```

Retrieve v's current state.

```
PROCEDURE SetColors (v: T; colors: PaintOp.ColorScheme);
```

Change the colors that v uses for the `Dormant` and `Vanish` states. If v is currently in the `Dormant` or `Vanish` state, mark v for redisplay.

```
END ReactivityVBT.
```

8.4 The ScaleFilter Interface

A `ScaleFilter` is a filter whose child's screentype is the same as the parent's except that the resolution is scaled.

```
INTERFACE ScaleFilter;

IMPORT VBT;

TYPE
  T <: Public;
  Public = Private OBJECT METHODS init (ch: VBT.T): T END;
  Private <: VBT.T;
```

The call `v.init(ch)` initializes `v` as a `ScaleFilter` with child `ch` and whose horizontal and vertical scale factors are 1.0.

A `ScaleFilter` is *not* defined as subclass of `Filter.T` because `Filter.Replace` requires both parent and the new child to have the same screentype. Indeed, breaking this invariant is the purpose of a `ScaleFilter`. This is not a serious limitation: you can initialize a `ScaleFilter` with `NEW(Filter.T)`, and change the grandchild of the `ScaleFilter`.

```
PROCEDURE New (ch: VBT.T): T;
<* LL.sup = VBT.mu *>
```

`New(...)` is equivalent to `NEW(T).init(...)`.

```
PROCEDURE Scale (t: T; hscale, vscale: REAL);
<* LL.sup = VBT.mu.t *>
```

Set `t`'s horizontal and vertical scale factors to `hscale` and `vscale` respectively.

```
PROCEDURE Child(t: T): VBT.T;
```

Returns `t`'s child

```
PROCEDURE Get(t: T; VAR (* OUT *) hscale, vscale: REAL);
```

Return `t`'s horizontal and vertical scale factors.

```
END ScaleFilter.
```

8.5 The Scale Interface

```
<*PRAGMA LL*>

INTERFACE Scale;
```

`Scale` provides a filter which uses a `ScaleFilter` to automatically rescale its contents to fill the window.

```
IMPORT ScaleFilter;

TYPE
  T <: PublicT;
  PublicT = ScaleFilter.T OBJECT keepAspectRatio := TRUE;  END;
```

A `Scale` is used exactly like a `ScaleFilter`. The only difference is that is automatically rescales its contents as the window is reshaped.

If `keepAspectRatio` is false, then x and y are scaled independently so that the contents just fit in each direction. If `keepAspectRatio` is true, then x and y are both scaled by the same amount such that the contents just fit in the larger direction and fit fine in the other direction.

```
END Scale.
```

8.6 The SplitterVBT Interface

A `SplitterVBT.T` is a parent window that partitions its screen into a row or column of children windows, depending on the *axis* of the split, with adjusting bars between all children. The adjusting bars allow the user to adjust the allocation of screen real estate among the splitter's children, subject to the size constaints of each child.

A `SplitterVBT` is subclass of an `HVSplit`, but through the `Multi` interface, only the "interesting" children of the `HVSplit` are exposed. That is, adjusting bars are never exposed to the client: they are inserted automatically when a new child is added, and removed as necessary. To access all children, including the adjusting bars, use the `Split` interface instead. The `HVSplit` routines `Move`, `Adjust`, `FeasibleRange`, `AvailSize`, and `AxisOf` can be used.

```
INTERFACE SplitterVBT;

IMPORT Axis, HVSplit, MultiClass, PaintOp, Pixmap;

TYPE Points = REAL;

CONST DefaultSize: Points = 5.0;

TYPE
  T <: Public;
  Public = HVSplit.T OBJECT
           METHODS
             init (hv       : Axis.T;
                   asTargets           := FALSE;
                   size     : Points   := DefaultSize;
                   op                  := PaintOp.BgFg;
                   txt                 := Pixmap.Gray;
                   saveBits            := FALSE;
                   parlim              := -1            ): T;
           END;
```

The call `v.init(...)` initializes `v` as a `SplitterVBT` with no children. See the `HVSplit` interface for an explanation of parameters `saveBits` and `parlim`. See the `HVBar` interface for an explanation of the `size`, `op`, and `txt` parameters.

When the `asTargets` flag is `TRUE`, each child—including the `HVBars`—of the `SplitterVBT` will be marked as a potential target for a `SourceVBT`. Also, there will be an `HVBar` both before and after all children.

```
TYPE MC <: MultiClass.T;
```

The type `MC` overrides the `replace` and `insert` methods of the type `MultiClass.T`.

Be sure to use the routines in `Multi`, not `Split`, for accessing the children of a `SplitterVBT`.

```
PROCEDURE New (hv        : Axis.T;
               asTargets          := FALSE;
               size      : Points := DefaultSize;
               op                 := PaintOp.BgFg;
               txt                := Pixmap.Gray;
               saveBits           := FALSE;
               parlim             := -1           ): T;
```

`New(...)` is equivalent to `NEW(T).init(...)`.

```
END SplitterVBT.
```

8.7 The ViewportVBT Interface

A `ViewportVBT` is a multi-filter that displays multiple views of a child `VBT`, with optional horizontal and vertical scrollbars. When the child's preferred size is larger than the viewport's *interior* (that is, the screen of the viewport minus the scrollbars), the child is reformatted to its preferred size. Since only part of the child is visible, the user can pan the child using the scrollbars. When the child's preferred size is smaller than the viewport's screen, the child is reformatted to the size of the viewport interior, and the scrollbars are ineffective.

Views may be added or deleted under program control or by appropriate gestures in the scrollbar: Option Left click adds a new view after the view in which the user clicked. Option Right click removes the view (unless, of course, it would leave the viewport with zero views!).

```
INTERFACE ViewportVBT;

IMPORT Axis, HVSplit, MultiClass, Rect, Shadow, VBT;

TYPE
  T <: Public;
  Public =
    HVSplit.T OBJECT
    METHODS
      init (ch             : VBT.T;
            axis                      := Axis.T.Ver;
            shadow         : Shadow.T := NIL;
            step           : CARDINAL := 10;
            adjustableViews           := TRUE;
```

```
             scrollStyle     := ScrollStyle.AlaViewport;
             shapeStyle      := ShapeStyle.Related;
             multiclass : MC := NIL                     ): T;
    END;
```

The call to `v.init(..)` initializes `v` as a `ViewportVBT.T`. The `axis` parameter says whether the views are arranged vertically or horizontally. `step` is the number of pixels to move while auto-scrolling. `shadow` gives the shadow for displaying scrollbars, resets and hvbars. When `adjustableViews` is `TRUE`, an `HVBar` will be inserted between views so the user can adjust the screen allocated to each view. `scrollStyle` and `shapeStyle` are explained below.

The initial view of `ch` is numbered 0 for use by `RemoveView`.

```
TYPE
  MC <: MultiClass.T;
```

The type MC overrides the `replace`, `pred`, and `succ` methods of `MultiClass.T`.

The internal structure of a viewport is a rather complex collection of `JoinVBT`s, `HVSplit`s, `ScrollerVBT`s, and others. It depends on the options with which the viewport was created. A viewport is a multi-filter; that is, to the `Multi` interface, it gives clients an abstraction of a single child. Be sure to use the `Multi` interface to get at the child.

```
TYPE
  View = INTEGER;
```

A `View` is an internal ID for a view. The value is valid for the life of a view (i.e., until it is removed by a user gesture or by a call to `RemoveView`). Thereafter, the ID may be reused. The initial view created by the `init` method has a value of 0.

A viewport can be created with a number of different styles of scrollbars:

```
TYPE
  ScrollStyle =
    {HorAndVer, HorOnly, VerOnly, NoScroll, AlaViewport, Auto};
```

The styles are as follows:

- `HorAndVer` puts a horizontal and vertical scrollbar on every view. In addition, nestled between the scrollbars in the southwest corner, there's a little "reset" button that moves the northwest corner of the child to the northwest corner of the view.
- `HorOnly` and `VerOnly` place a scrollbar at the bottom (for horizontal scrolling) or the left side (for vertical scrolling).
- `NoScroll` specifies that views will not have scrollbars.

- `AlaViewport` specifies that there is a scroll bar in the same axis as the viewport. Thus, `AlaViewport` for a vertical viewport is equivalent to `VerOnly`.
- `Auto` specifies that scroll bars appear only when the preferred size of the child exceeds the size of the viewport (in that dimension).

There are two styles of shape relationships between a viewport and its child:

```
TYPE
  ShapeStyle = {Unrelated, Related};
```

`Unrelated` makes the shape of the viewed child equal to the child's preferred shape—completely unrelated to the viewport's current shape. `Related` makes the shape of the viewed child's shape equal to the the viewport's shape in the non-axis direction of the viewport. In the viewport's axis direction, the child's preferred shape is used. For example, the width of the child in a `Vertical` viewport is the width of the viewport.

Here's how to change which part of child is within the viewport:

```
PROCEDURE ScrollTo (          v    : T;
                    READONLY r    : Rect.T;
                             view : View     := 0;
                             force           := TRUE);
<* LL = VBT.mu *>
```

Scroll the viewport `v` so that rectangle `r` is visible in view `view`. Rectangle `r` will be roughly centered within `v`, but if `r` is too big to be seen entirely, its northwest corner will be made visible. If `force` is `FALSE` and `r` is already entirely visible, this procedure is a no-op.

```
PROCEDURE Normalize (v: T; w: VBT.T; view: View := 0);
<* LL = VBT.mu *>
```

If the domain of `w` is non-empty and it's entirely visible, do nothing. Otherwise, do a `ScrollTo` to `w`'s domain in view `view`.

At first blush, `Normalize` seems to be just a call to

```
ScrollTo(v, VBT.Domain(w), FALSE)
```

However, if `w` doesn't have a domain, as is the case when `w` has been recently installed and the `VBT` tree has not been redisplayed, a thread is forked to wait until it can acquire `VBT.mu` (recall that `Normalize` and `ScrollTo` have `LL = VBT.mu`). After the lock is acquired, all pending redisplays are satisfied, and then `ScrollTo` of `w`'s domain is invoked. Since the thread executes outside event-time, it explicitly causes all marked `VBT`s to be redisplayed after it calls `ScrollTo`.

Finally, here's how to add and delete views from a program:

```
PROCEDURE AddView (v: T; after: View := -1; split := TRUE): View;
<* LL = VBT.mu *>
```

Add another view after the view `after` *(-1 means add as the first view) of the child. If* `split` *is* `TRUE`*, then the new view and the view* `after` *will occupy the area previously occupied by the view* `after`*. The area of all other views will be unchanged. The value returned is an internal ID for the view. This value may be reused after the view has been removed.*

```
PROCEDURE RemoveView (v: T; view: View);
<* LL = VBT.mu *>
```

Remove the view `view` *from v's child. The ID for the initial view created by the* `init` *method is 0.*

```
END ViewportVBT.
```

9 Utilities

This section describes various utilities that VBTkit provides.

9.1 The AnyEvent Interface

An `AnyEvent.T` is an object that can hold any of the Trestle event-time events. This object type is useful for `VBT` methods that are called in response to multiple styles of user gestures. For instance, the `callback` method of a `NumericVBT` is invoked either because a user clicked on the plus or minus button or because the user typed a carriage return in the type-in field. The Trestle event is passed to the `callback` method as an `AnyEvent.T`, and the `callback` method can then use a `TYPECASE` to differentiate button clicks from carriage returns, and to retrieve the data specific to each type of event.

The locking level is arbitrary for all procedures in this interface.

```
INTERFACE AnyEvent;

IMPORT VBT;

TYPE
  T = BRANDED OBJECT END;
  Key = T OBJECT key: VBT.KeyRec END;
  Mouse = T OBJECT mouse: VBT.MouseRec END;
  Position = T OBJECT position: VBT.PositionRec END;
  Misc = T OBJECT misc: VBT.MiscRec END;
```

The four subtypes of `AnyEvent.T` *correspond to the four event-time Trestle events: keyboard, mouse, position, and miscellaneous.*

```
PROCEDURE FromKey (READONLY event: VBT.KeyRec): Key;
PROCEDURE FromMouse (READONLY event: VBT.MouseRec): Mouse;
PROCEDURE FromPosition (READONLY event: VBT.PositionRec): Posi-
tion;
PROCEDURE FromMisc (READONLY event: VBT.MiscRec): Misc;
```

Return `event` *as an appropriate subtype of* `AnyEvent.T`.

```
PROCEDURE TimeStamp (anyevent: T): VBT.TimeStamp;
```

Return the timestamp of the `anyevent`. *It is a checked runtime error if* `anyevent` *is not a proper subtype of* `AnyEvent.T`.

```
END AnyEvent.
```

9.2 The AutoRepeat Interface

The `AutoRepeat` interface provides support for calling a procedure repetitively. Auto-repeating typically takes place while a key or mouse button is held down, although there is no direct relation between `AutoRepeat` and `VBT`s.

When an auto-repeat object `ar` is activated, it forks a *timer thread* that calls `ar.repeat()` after `firstWait` milliseconds, and every `period` milliseconds thereafter. However, there is a flow control mechanism: if the call to `ar.repeat()` has not returned by the time the next repetition should take place, the timer thread will wait. That is, repetitions cannot queue up more than one deep.

An auto-repeat object `ar` is activated by a call to `Start(ar)`, terminated by a call to `Stop(ar)`, and resumed by a call to `Continue(ar)`.

All locking is handled within `AutoRepeat`; calls to `Start(ar)`, `Stop(ar)`, and `Continue(ar)` are serialized on a per-`ar` basis. These procedures may be called by a `repeat` method. Clients must not call the `repeat` method directly; it is called by the timer thread subject to client-calls to `Start`, `Stop`, and `Continue`. The `AutoRepeat` interface will never call a `repeat` method re-entrantly.

```
INTERFACE AutoRepeat;

TYPE
  Milliseconds = CARDINAL;

CONST
  DefaultFirstWait: Milliseconds = 500;
  DefaultPeriod   : Milliseconds = 66;

TYPE
  T <: Public;
  Public =
    Private OBJECT
    METHODS
      init (firstWait: Milliseconds := DefaultFirstWait;
            period   : Milliseconds := DefaultPeriod): T;
      repeat ();
    END;
  Private <: ROOT;
```

The call `ar.init(firstWait, period)` initializes `ar` as an `AutoRepeat.T`, and it returns `ar`. The `firstWait` and `period` parameters are stored internally for use by the `Start` and `Continue` procedures.

An auto-repeat object `ar` does not have a default `repeat` method; it must be supplied by the client.

```
PROCEDURE Start (ar: T);
```

Initiate auto-repeating for `ar`.

`Start(ar)` forks a timer thread that will wait `ar.firstWait` milliseconds before calling `ar.repeat()` the first time, then `ar.period` milliseconds between subsequent calls to `ar.repeat()`. This procedure is a no-op if `ar` is already running.

```
PROCEDURE Stop (ar: T);
```

Stop auto-repeating as soon as possible.

After calling `Stop(ar)`, `AutoRepeat` will not call `ar.repeat()` again until a call to `Start(ar)` or `Continue(ar)` restarts auto-repeating. This procedure is a no-op if `ar` is already running.

It is possible (but unlikely) that `ar.repeat()` is called one more time after a call to `Stop(ar)` returns. This can happen because the call to `ar.repeat` is not serialized with repect to the call to `Stop(ar)`. They are not serialized in order to allow a `repeat` method to call `Stop`.

```
PROCEDURE Continue (ar: T);
```

Resume auto-repeating immediately.

`Continue(ar)` is like `Start(ar)`, except rather than waiting `ar.firstWait` milliseconds as in the call to `Start(ar)`, the timer thread calls `ar.repeat` without waiting at all. Subsequent calls to `ar.repeat()` happen every `period` milliseconds, as usual. This procedure is a no-op if `ar` is already running.

```
END AutoRepeat.
```

9.3 The Image Interface

An `Image.T` is a screen-independent specification of an *image*. An image is a pixmap that includes specifications for both color and resolution. It is rendered consistently across screen types in terms of its colors and size.

```
INTERFACE Image;

IMPORT Pixmap, Rd, ScrnPixmap, Thread, TrestleComm,
  VBT, Word, Wr;

TYPE T = Pixmap.T;

EXCEPTION Error;
```

An `Image.Raw` is an array of pixels, with both resolution and color information. (It is like a `Pixmap.Raw`, with the addition of resolution and color information.) An `Image.T` is built from one or more `Image.Raw` pixmaps.

There are three types of `Image.Raw` pixmaps: The `RawBitmap` represents bitmaps (1-bit deep pixmaps); the `RawPixmap` represents pixmaps that do not have a color table; and the `RawPixmapCMap` represents pixmaps that use a color table.

```
TYPE
  Pixel = Word.T;
  Mode = {Stable, Normal, Accurate};
  RGB = RECORD r, g, b: REAL END;

TYPE
  Raw = OBJECT
      width, height: INTEGER;
      xres, yres: REAL := 75.0; (* in pixels per inch *)
    METHODS
      get (h, v: INTEGER): Pixel;
      set (h, v: INTEGER; pixel: Pixel);
    END;

  RawBitmap = Raw BRANDED OBJECT END;

  RawPixmap = Raw OBJECT
      needsGamma := TRUE;
      colorMode  := Mode.Normal;
    END;

  RawPixmapCMap = RawPixmap OBJECT
      colors: REF ARRAY OF RGB;
    END;
```

If `pm` is a `Raw` pixmap, then `pm` contains `pm.height` rows, and each row contains `pm.width` elements. These fields are read-only after they are initialized. The pixels are accessed with (0,0) in the northwest corner and `(width-1,height-1)` in the southeast corner. The `pm.xres` and `pm.yres` fields specify the resolution at which `pm` was designed. The `get` and `set` methods retrieve and store individual elements of the pixmap.

Each of the three subtypes of `Raw` interprets a "pixel" as follows:

- If `pm` is a `RawBitmap` pixmap, then it is guaranteed that the method `pm.get` will return a 0 or 1. In the call `pm.set(h,v,pixel)`, only the least significant bit of `pixel` is used.

- If `pm` is a `RawPixmap`, the pixels in `pm` encode an RGB value each of whose components is 8 bits. A `(r,g,b)` triple is stored as

  ```
  r * 256 * 256 + g * 256 + b
  ```

  and each of `r`, `g`, and `b` is between 0 and 255. The field `pm.needsGamma` indicates whether to let Trestle gamma-correct the colors. The `pm.colorMode` field determines how each RGB value in the pixmap should be displayed on color-mapped display.

- If `pm` is a `RawPixmapCMap`, the pixels in `pm` are used as in index into the color table stored in the field `pm.colors`.

The colors used to display a colored pixmap `p` depends on a number of factors. The `p.colorMode` field is used to match colors in the pixmap with colors in the color table, as described in the `ScrnPixmap` interface. The matching depends on other applications running, on other pixmaps being displayed, and on the depth of the screen.

The current implementation does not perform any dithering, except on monochrome screens. On monochrome screen, a very crude "thresholding" is performed: if the brightness of the color is more than 50% of the maximum brightness, the screen's foreground color is used. Otherwise, the screen's background color is used.

9.3.1 Retrieving, storing, and creating "raw" pixmaps

An `Image.Raw` can be built from a reader containing an image in Jeff Poskanzer's "portable anymap file" ("pnm") format, and a "pnm" description of an `Image.Raw` can be stored into a writer.

There are many tools available in the public domain for manipulating images in "pnm" format and for converting between that format and other formats (e.g., GIF, X11, Macintosh PICT, HP PaintJet, and so on).

There are three types of "pnm" files:

- "pbm" – portable bitmap file
- "pgm" – portable graymap file
- "ppm" – portable pixmap file

Each of these format has two variants: "raw" and "ascii". In the "ascii" version, pixel values are stored as ASCII decimal numbers. In the "raw" version, pixel values must be less than 256 and are stored as plain bytes.

```
PROCEDURE FromRd (rd: Rd.T): Raw
  RAISES {Thread.Alerted, Rd.Failure, Error};
<* LL = arbitrary *>
```

Returns an `Image.Raw` *from the reader* `rd` *containing an image in "pnm" format. Pixels in "ppm" files are normalized to 8 bits per channel and intensity values of "pgm" files are normalized to 8 bits.*

```
PROCEDURE ToWr (raw: Raw; wr: Wr.T)
  RAISES {Thread.Alerted, Wr.Failure};
<* LL = arbitrary *>
```

Store an ascii description of `raw` *into the writer* `wr` *using "pnm" format.*

Procedures `FromRd` and `ToWr` are not guaranteed to be idempotent because pixel values are normaized by `FromRd` to be 8 bits. Also, the "pnm" format produced by `ToWr` is either ascii "pbm" for subtypes of `RawBitmap` or ascii "ppm" for subtypes of `RawPixmap`, whereas procedure `FromRd` can accept these formats as well as the "raw" variants and grayscale formats ("pgm").

The more serious limitation of using "pnm" format is that "pnm" does include any information about the pixmap resolution or color matching. `FromRd` will use the default reslution of a `Raw` and the default color parameters of a `RawPixmap`; `ToWr` simply ignores the resolution and color parameters.

Finally, the following procedure is used to create an `Image.Raw` from a screen-dependent pixmap:

```
PROCEDURE FromScrnPixmap (
    spm: ScrnPixmap.T;
    st: VBT.ScreenType): Raw RAISES {TrestleComm.Failure};
<* LL.sup <= VBT.mu *>
```

Returns a screen-independent `Raw` *which describes the pixmap* `spm` *when displayed on* `st`. *Any field of* `Raw` *that cannot be computed from* `spm` *and* `st` *is given its default value. For example, the* `needsGamma` *and the* `colorMode` *fields of pixmaps that are deeper than 1-bit.*

9.3.2 Building an image from "raw" pixmaps

The remaining procedures in this interface create an `Image.T` from one or more `Image.Raw` pixmaps:

```
PROCEDURE Unscaled (raw: Raw): T;
<* LL.sup <= VBT.mu *>
```

Returns a pixmap that will display as `raw`. *The pixels in* `raw` *will not be scaled regardless of the screen's resolution.*

So, for example, consider a pixmap `p` whose dimensions are 150 wide by 50 high. On a 75dpi screen (a typical 1993 vintage monitor), pixmap `p` would appear 2 inches wide and 2/3 inches high. On a high-resolution monitor of 300 dpi, pixmap `p` would appear .5 inches wid and 1/6 inches high. The `p.xres` and `p.yres` fields are ignored.

If you wanted `p` to always appear as 2 inches by 2/3 inches, regardless of the pixel density of the monitor, you'd use `Scaled` instead:

```
PROCEDURE Scaled (raw: Raw): T;
<* LL.sup <= VBT.mu *>
```

Return a pixmap that will display as `raw`, *scaled for the screen's resolution. The horizontal and vertical dimensions are scaled independently.*

The current implementation scales pixmaps by non-negative integer amounts: horizontally by `ROUND(dpiX/p.xres)` and vertcially by `ROUND(dpi.x/p.yres)`, where `dpiX` and `dpiY` are the horizontal and vertical resolution of the screen, respectively, expressed in dots-per-inch.

In the example above, suppose that `p.xres` and `p.yres` were both 75. On a 300 dpi screen, `p` would appear 2 inches wide and 2/3 inches high. Each pixel in `p` would appear as a block of 4x4 screen pixels. If the screen were 250 dpi horizontally and 175 dpi vertically, then `p` would appear 1-1/2 inches wide and 1-1/3 inches high. Each pixel in `p` would appear as a block of 3x2 screen pixels.

```
END Image.
```

9.4 The Pts Interface

The `Pts` interface contains utilities to convert between points and pixels. VBTkit uses 72 points per inch, following the Postscript convention [1, page 151]. There are 25.4 millimeters to the inch.

The locking level is arbitrary for all procedures in this interface.

```
INTERFACE Pts;

IMPORT Axis, VBT;

PROCEDURE ToScreenPixels (v: VBT.T; pts: REAL; ax: Axis.T):
  INTEGER;
```

Return the number of screen pixels that correspond to `pts` *points on v's screentype in the axis* `ax`*; or return 0 if v's screentype is NIL. Equivalent to*

```
        ROUND (ToPixels (v, pts, ax))
```

```
PROCEDURE ToPixels (v: VBT.T; pts: REAL; ax: Axis.T): REAL;
```

Return the number of pixels that correspond to `pts` *points on v's screentype in the axis* `ax`*; or return 0 if v's screentype is NIL.*

```
CONST
  PtsPerInch = 72.0;
  MMPerInch  = 25.4;

PROCEDURE FromMM (mm: REAL): REAL;
```

Convert from millimeters to points.

```
PROCEDURE ToMM (pts: REAL): REAL;
```

Convert from points to millimeters.

```
END Pts.
```

9.5 The VBTColors Interface

The `VBTColors` interface provides a way to associate a `VBT`'s background and foreground colors with the `VBT`. This information can be retrieved by some other `VBT` to compute a related color.

```
INTERFACE VBTColors;

IMPORT PaintOp, VBT;

PROCEDURE Put (v: VBT.T; colors: PaintOp.ColorScheme);
<* LL.sup < v *>
```

Store `colors` *with* `v`.

```
PROCEDURE Get (v: VBT.T): PaintOp.ColorScheme;
<* LL.sup < v *>
```

Return the colors stored by the most recent call to `Put`. *If* `Put` *has never been called on* `v`, *return* `PaintOp.bgFg`.

```
END VBTColors.
```

9.6 The XParam Interface

The `XParam` interface provides utilities for handling X-style `-display` and `-geometry` arguments. If your application installs a single top-level window, the `XTrestle` interface may be more appropriate than this interface.

```
INTERFACE XParam;

IMPORT Point, Rect, Trestle, TrestleComm;
```

Here are routines for manipulating the `-display` option:

```
TYPE
  Display = RECORD
              hostname: TEXT     := "";
              display : CARDINAL := 0;
              screen  : CARDINAL := 0;
              DECnet  : BOOLEAN  := FALSE
            END;

PROCEDURE ParseDisplay (spec: TEXT): Display RAISES {Error};
<* LL = arbitrary *>
```

Return a parsed version of the `-display` *parameter in* `spec`. *For example, if* `spec` *contains the string* `myrtle.pa.dec.com:0.2`, *the record returned would be*

```
            Display{hostname := "myrtle.pa.dec.com",
                    display := 0, screen := 2, DECnet := FALSE}
```

```
PROCEDURE UnparseDisplay (READONLY d: Display): TEXT;
<* LL = arbitrary *>
```

Return the text-version of the `-display` *parameter* `d`.

Here are routines for manipulating the `-geometry` option:

```
CONST Missing = Point.T{-1, -1};

TYPE
  Geometry =
    RECORD
      vertex := Rect.Vertex.NW;  (* corner for displacement *)
      dp     := Point.Origin;    (* displacement *)
      size   := Missing;         (* width, height *)
    END;

PROCEDURE ParseGeometry (spec: TEXT): Geometry RAISES {Error};
<* LL = arbitrary *>
```

Return a parsed version of the `-geometry` *parameter in* `spec`*. For example, if* `spec` *contains the string* `1024x800-0-10`*, the returned record would be*

```
            Geometry {Rect.Vertex.SE,
                      Point.T {0, 10},
                      Point.T {1024, 80}}
```

The `size` *field defaults to* `Missing`*. The horizontal and vertical displacements default to* `Point.Origin` *(no displacement). The displacements are always positive values; use the* `vertex` *field to find out from which corner they are to be offset.*

```
PROCEDURE UnparseGeometry (READONLY g: Geometry): TEXT;
<* LL = arbitrary *>
```

Return the text-version of the `-geomtry` *switch* `g`.

```
PROCEDURE Position (          trsl: Trestle.T;
                              id  : Trestle.ScreenID;
                     READONLY g   : Geometry          ): Point.T
  RAISES {TrestleComm.Failure};
<* LL.sup = VBT.mu *>
```

Return the position specified by `g` *in the screen coordinates for the screenID* `id` *on the window system connected to* `trsl` *(cf.* `Trestle.GetScreens`*). The value of* `g.size` *must not be* `Missing`*, unless* `g.vertex` *is the northwest corner.*

Here is the definition of the `Error` exception:

```
TYPE
  Info = OBJECT
           spec : TEXT;
           index: CARDINAL
         END;
  GeometryInfo = Info BRANDED OBJECT END;
  DisplayInfo = Info BRANDED OBJECT END;

EXCEPTION Error(Info);
```

Parsing errors are reported with the text (`spec`) and position (`index`) of the first illegal character in the text.

9.6.1 An example

Here is an example of how to use this interface to install a VBT `v` as a top level window, obeying the `-display` and `-geometry` switches given to the application.

First, here are the variables we'll be using:

```
VAR
  display, geometry: TEXT := NIL;
  d: XParam.DisplayRec;
  g: XParam.Geometry;
  trsl: Trestle.T;
BEGIN
```

Next, parse the command line arguments:

```
  ParseParams.BeginParsing(Stdio.stderr);
  TRY
    IF ParseParams.KeywordPresent("-display") THEN
      display := ParseParams.GetNext();
      d := XParam.ParseDisplay(display);
    END;
  EXCEPT Scan.BadFormat, XParam.Error =>
    display := NIL;
    Wr.PutText(
      Stdio.stderr, "Illegal -display argument ignored");
  END;

  TRY
    IF ParseParams.KeywordPresent("-geometry") THEN
      geometry := ParseParams.GetNext();
      g := XParam.ParseGeometry(geometry)
      IF g.size = XParam.Missing THEN
        WITH shapes = VBTClass.GetShapes(v, FALSE) DO
          g.size.h := shapes[Axis.T.Hor].pref;
```

```
              g.size.v := shapes[Axis.T.Ver].pref;
            END
          END
        END;
      EXCEPT Scan.BadFormat, XParam.Error =>
        geometry := NIL;
        Wr.PutText(
          Stdio.stderr, "Illegal -geometry argument ignored");
      END;
```

At this point, if `display` is non-`NIL`, then `d` contains the information from the `-display` switch. Similarly, if `geometry` is non-`NIL`, then `g` contains the information from the `-geometry` switch. If the window-size specificiation was missing, the preferred shape of the window is used.

Finally, we now process the `display` and `geometry` information:

```
      trsl := Trestle.Connect(display);
      Trestle.Attach(v, trsl);
      TrestleImpl.SetDefault(trsl);
      Trestle.Decorate(v, ... );
      IF geometry = NIL THEN
        Trestle.MoveNear(v, NIL)
      ELSE
        StableVBT.SetShape(v, g.size.h, g.size.v)
        Trestle.Overlap(
          v, d.screen, XParam.Position(trsl, d.screen, g))
      END;
    END;
```

The `TrestleImpl` and `StableVBT` interfaces are part of Trestle. The call to `TrestleImpl.SetDefault` establishes the value of the `-display` switch as the default Trestle connection. The call to `StableVBT.SetShape` is used to control the size of a top-level window.

```
END XParam.
```

9.7 The XTrestle Interface

`XTrestle` checks for X-style `-display` and `-geometry` command-line switches and installs a top-level window accordingly. If your application install more than one top-level window, you may find the routines in the `XParam` interface helpful.

```
INTERFACE XTrestle;

IMPORT TrestleComm, VBT;
```

```
EXCEPTION Error;

PROCEDURE Install (v           : VBT.T;
                   applName    : TEXT    := NIL;
                   inst        : TEXT    := NIL;
                   windowTitle: TEXT     := NIL;
                   iconTitle  : TEXT     := NIL;
                   bgColorR    : REAL    := -1.0;
                   bgColorG    : REAL    := -1.0;
                   bgColorB    : REAL    := -1.0;
                   iconWindow : VBT.T    := NIL   )
  RAISES {TrestleComm.Failure, Error};
<* LL.sup = VBT.mu *>
```

Like `Trestle.Install` *except that the locking level is different and the command line is parsed for X-style* `-display` *and* `-geometry` *options.*

The syntax of these switches is described in the X man page and in *X Window System* by Scheifler and Gettys.

If there is a `-display` argument, it will be made the default Trestle connection for those procedures in the `Trestle` interface that take a `Trestle.T` as a parameter.

The arguments to the command line are parsed using the `ParseParams` module. That module maintains a global state of which arguments have been parsed; this routine has the side-effect of resetting that global state.

The `TrestleComm.Failure` exception is raised if a call to `Trestle` raises that exception. The `Error` exception is raised if the string following `-display` or `-geometry` contains any syntax errors (or is missing).

```
END XTrestle.
```

10 Color Utilities

This section describes the utilities that VBTkit provides for specifying colors. The `Color` interface defines two color models, RGB (Red, Green, Blue) and HSV (Hue, Saturation, Value), and contains procedures to convert between the color models. The `ColorName` interface provides routines to translate a color name, such as "VeryPaleCornflowerBlue," into an RGB triple. The locking level is arbitrary for all procedures in these interfaces.

10.1 The Color Interface

A `Color.T` describes a color as a mixture of the three color TV primaries (Red, Green and Blue), in a linear scale (proportional to luminous power), where 0.0 = black and 1.0 = maximum screen intensity.

The set of all colors with RGB coordinates in the range 0.0–1.0 is the *unit RGB cube*. The colors along the main diagonal of the unit cube (from (0,0,0) to (1,1,1)) contain equal amounts of all three primaries; they represent gray levels. RGB triples outside the unit cube cannot be displayed on typical color monitors, but are still legal as far as this interface is concerned, make perfect physical sense, and are useful in some color computations.

This interface also provides routines to convert colors between the HSV (Hue, Saturation, Value) and RGB color models.

```
INTERFACE Color;

TYPE
  T = RECORD r, g, b: REAL;  END;

CONST
  (* The vertices of the unit RGB cube: *)
  Black   = T{0.0, 0.0, 0.0};
  Red     = T{1.0, 0.0, 0.0};
  Green   = T{0.0, 1.0, 0.0};
  Blue    = T{0.0, 0.0, 1.0};
  Cyan    = T{0.0, 1.0, 1.0};
  Magenta = T{1.0, 0.0, 1.0};
  Yellow  = T{1.0, 1.0, 0.0};
  White   = T{1.0, 1.0, 1.0};
```

The following procedures are useful for converting a color into a shade of gray:

```
PROCEDURE Brightness (READONLY rgb: T): REAL;
```

Return the intensity of `rgb` in a linear scale. The formula used is

*0.239 * rgb.r + 0.686 * rgb.g + 0.075 * rgb.b*

clipped to the range 0.0–1.0.

An `HSV` is a color represented as a (Hue, Saturation, Value) triple. The HSV color model is somewhat more intuitive than the RGB color model. It's based on mimicking the way that an artist mixes paint: "He chooses a pure hue, or pigment and lightens it to a *tint* of that hue by adding white, or darkens it to a *shade* of that hue by adding black, or in general obtains a *tone* of that hue by adding some mixture of white and black."

So, varying hue corresponds to selecting a pure color along a color wheel where 0 is red, .167 is yellow, .333 is green, .5 is cyan, .667 is blue, and .833 is magenta, and 1.0 is red again. Decreasing the saturation (from 1 down to 0) corresponds to adding white. Decreasing the value (from 1 down to 0) corresponds to adding black.

This interface provides procedures to map between RGB and HSV color models. Note that white and black have indeterminate hue and saturation. Pure colors have saturation=1 and value=1, whereas grey levels have saturation=0, value=brightness, and indeterminate hue.

```
TYPE
  HSV = RECORD h, s, v: REAL END;
```

The following procedures convert between RGB and HSV color models:

```
PROCEDURE ToHSV (READONLY rgb: T): HSV;
```

Convert from RGB to HSV coordinates. By convention, gray colors (including white and black) get hue=0.0. In addition, black gets saturation=0.0.

```
PROCEDURE FromHSV (READONLY hsv: HSV): T;
```

Convert from HSV to RGB coordinates. If value=0 (black), saturation and hue are irrelevant. If saturation=0 (gray), hue is irrelevant.

```
END Color.
```

10.2 The ColorName Interface

The `ColorName` interface provides a standard mapping between color names and linear RGB triples. The implementation recognizes the following names, based on those found in `/usr/lib/X11/rgb.txt`:

`AliceBlue`	`ForestGreen`	`MintCream`	`SandyBrown`
`AntiqueWhite` †	`Gainsboro`	`MistyRose` †	`SeaGreen` †
`Aquamarine` †	`GhostWhite`	`Moccasin`	`Seashell` †
`Azure` †	`Gold` †	`NavajoWhite` †	`Sienna` †
`Beige`	`Goldenrod` †	`Navy`	`SkyBlue` †
`Bisque`	`GoldenrodYellow`	`NavyBlue`	`SlateBlue` †
`Black`	`Gray` ‡	`OldLace`	`SlateGray` †
`BlanchedAlmond`	`Green` †	`OliveDrab` †	`SlateGrey`
`Blue` †	`GreenYellow`	`OliveGreen` †	`Snow` †
`BlueViolet`	`Grey` ‡	`Orange` †	`SpringGreen` †
`Brown` †	`Honeydew` †	`OrangeRed` †	`SteelBlue` †
`Burlywood` †	`HotPink` †	`Orchid` †	`Tan` †
`CadetBlue` †	`IndianRed` †	`PapayaWhip`	`Thistle` †
`Chartreuse` †	`Ivory` †	`PeachPuff` †	`Tomato` †
`Chocolate` †	`Khaki` †	`Peru`	`Turquoise` †
`Coral` †	`Lavender`	`Pink` †	`Violet`
`CornflowerBlue`	`LavenderBlush` †	`Plum` †	`VioletRed` †
`Cornsilk` †	`LawnGreen`	`Powderblue`	`Wheat` †
`Cyan` †	`LemonChiffon` †	`Purple` †	`White`
`DeepPink` †	`LimeGreen`	`Red` †	`WhiteSmoke`
`DeepSkyBlue` †	`Linen`	`RosyBrown` †	`Yellow` †
`DodgerBlue` †	`Magenta` †	`Royalblue` †	`YellowGreen`
`Firebrick` †	`Maroon` †	`SaddleBrown`	
`FloralWhite`	`MidnightBlue`	`Salmon` †	

The dagger (†) indicates that the implementation recognizes a name along with the suffixes `1`–`4`; e.g., `Red`, `Red1`, `Red2`, `Red3`, and `Red4`.

The double dagger (‡) indicates that the implementation also recognizes the names with the suffixes `0` through `100`. That is, `Gray0`, `Gray1`, ..., `Gray100`, as well as `Grey0`, `Grey1`, ..., `Grey100`.

In addition, the name of a color C from this list can be prefixed by one or more of the following modifiers:

Term	*Meaning*
`Light` `Pale`	1/3 of the way from C to white
`Dark` `Dim`	1/3 of the way from C to black

`Drab`
`Weak`
`Dull`
	1/3 of the way from C to the gray with the same brightness as C

`Vivid`
`Strong`
`Bright`
	1/3 of the way from C to the purest color with the same hue as C

`Reddish`	1/3 of the way from C to red
`Greenish`	1/3 of the way from C to green
`Bluish`	1/3 of the way from C to blue
`Yellowish`	1/3 of the way from C to yellow

Each of these modifiers can be modified in turn by the following prefixes, which replace "1/3 of the way" by the indicated fraction:

Term	*Degree*	*% (approx.)*
`VeryVerySlightly`	1/16 of the way	6%
`VerySlightly`	1/8 of the way	13%
`Slightly`	1/4 of the way	25%
`Somewhat`	3/8 of the way	38%
`Rather`	1/2 of the way	50%
`Quite`	5/8 of the way	63%
`Very`	3/4 of the way	75%
`VeryVery`	7/8 of the way	88%
`VeryVeryVery`	15/16 of the way	94%

The modifier `Medium` is also recognized as a shorthand for `SlightlyDark`. (But you cannot use `VeryMedium`.)

```
INTERFACE ColorName;

IMPORT Color, List;

EXCEPTION NotFound;

PROCEDURE ToRGB (name: TEXT): Color.T RAISES {NotFound};
```

Give the `RGB.T` value described by `name`, ignoring case and whitespace. A cache of unnormalized names is maintained, so this procedure should be pretty fast for repeated lookups of the same name.

```
PROCEDURE NameList (): List.T;
```

Return a list of all the "basic" (unmodified) color names known to this module, as lower-case `TEXT`s, in alphabetical order.

```
END ColorName.
```

A The TextPort User Interfaces

A.1 The IvyModel Interface

```
INTERFACE IvyModel;

IMPORT TextPortClass;

TYPE T <: TextPortClass.Model;

END IvyModel.
```

The Ivy model, which was the only model that TextPort originally included, supports two text-selections. `TextPort`'s primary selection corresponds to `VBT.Target`, which is the X server's *secondary* selection. `TextPort`'s secondary selection corresponds to `VBT.Source`, which is the X server's *primary* selection.

There are two ways of acquiring the source selection. The usual way is to control-click to select a point, word, line, paragraph, or buffer. The second way is to use the Copy command (option-C) or the Cut command (option-X); these commands make a copy of the *primary* selection, which becomes the *source* selection. If there was already a source (secondary) selection, its highlighting will be removed.

The following list shows the Ivy keybindings. The modifiers (control, option, and shift) are part of the `VBT.KeyRec` passed to `TextPort` by Trestle. The key that Trestle says is the "option" key may actually be labeled as something else, such as "Compose Character" on an LK201 keyboard.

Return	invoke the `returnAction` method
shift-Return	call `Newline`
option-Return	insert a newline after the cursor
Backspace	delete primary selection or the previous character
option-Backspace	swap the two previous characters
control-A	delete previous character
control-B	delete whole line
control-C	delete to start of line
option-C	**Copy**
control-D	delete to the start of the current word
control-E	**Move**: replace target with source, and clear source
control-F	delete to the end of the current word
control-G	delete whole word
control-H	swap the selection *boundaries*
control-I	move to the next word
control-J	move to previous character
control-K	move to next character
control-L	move to first non-blank and select line
control-M	find previous occurrence
option-M	find previous occurrence of primary

control-N	find next occurrence of primary
option-N	find first occurrence of primary
control-O	move up 1 row in the current column
control-P	move down 1 row in the current column
control-Q	**Clear**
control-R	**Swap**: exchange the selected *text*
control-S	delete the next character
control-U	move to the previous word
control-V	delete to end of line
option-V	**Paste**
control-W	**Paste**
option-X	**Cut**
control-Y	move to opposite end of selection
control-Z	**Undo**
control-shift-Z	**Redo**
control-,	find next occurrence
control-;	move to end of line and select line
control-Space	normalize

A.1.1 The Ivy selection model

`TextPort` was originally designed after an editor called Ivy [?] that was developed at SRC. Ivy was written in Modula-2 and included a wealth of features; `TextPort`'s Ivy model implements only a small subset of them.

The best description of the selection model used by the Ivy model is in documentation for Ivy itself, from which the following section is adapted.

`TextPort` uses two selections, the primary selection and the secondary selection. Briefly, the primary selection is *established* by clicking the left mouse button; it is *extended* by dragging the right mouse button. The secondary selection is established and extended the same way, but while holding down the control key.

The primary selection is used for almost all editing, and for other things as well. `TextPort` highlights the primary selection either by underlining the selected characters with a heavy black line or by displaying the selected characters in reverse video.

The secondary selection is used for Cut, Copy, Paste, Move, and Swap. `TextPort` highlights the secondary selection by underlining the selected characters with a grey line.

When the primary and secondary selections overlap, the text in common is indicated by an ugly stippled background. Fortunately, this does not happen very often.

No matter how many `TextPort` windows you have, you can have at most one primary and one secondary selection at any one time.

The type-in point tells you where `TextPort` is going to insert characters when you type. It is always located at one end or the other of the primary selection. It is indicated by a blinking vertical bar.

A.1.2 Making a single-unit primary selection

You use the Left and Middle buttons on your mouse to select a unit: a point, a word, a line, a paragraph, or the entire file.

To primary-select a point between characters, point the cursor below or at that point, and click Left. A blinking vertical bar (the type-in point) appears at the point you selected.

A word is a maximal non-empty character sequence containing (1) only letters and digits, or (2) one or more space and tab characters, or (3) a single character that is not a letter, a digit, a space, or a tab. E.g., in this sequence:

The cat sat on
/proj/packages.

the words are:

the initial spaces on the first line
`The`
space
`cat`
space
`sat`
space
`on`
newline
the initial spaces on the second line
`/`
`proj`
`/`
`packages`
`.`

To primary-select a word, point the cursor at that word and click Middle. A black underline appears under the word you selected.

You can also select a word more deliberately by holding down Middle and mousing around; the underline will track the mouse. When you reach the word you're interested in, release Middle. The effect is the same as clicking Middle at the final cursor position.

A line is a non-empty character sequence containing at most one newline, whose first character immediately follows either start-of-file or a newline, and whose final character is either a newline or the character immediately preceding end-of-file; if the line ends with end-of-file it contains no newline. This sounds weird; mostly, a line is what you think it is—all the characters from immediately after one newline up to and including the next newline. If the line doesn't need to be turned, it looks like a line; if it's turned, it looks like two or more lines with a grey vertical bar at the far-right extreme of each line except the last.

To select a line, double-click Left; a black underline appears under the line you selected. Just as for points and words, the underline of the selected line tracks the cursor while Left is down; so you can make a click-and-a-half and then hold Left down while you move the cursor to select just the line you are after.

A paragraph is a sequence of lines—either a maximal sequence of non-blank lines or a maximal sequence of blank lines. (A blank line contains only spaces, tabs, and at most one newline.)

To select a paragraph, double-click Middle; a black underline appears under the paragraph you selected.

To select all of the characters in a `TextPort`, triple-click Left or Middle; a black underline appears under the entire text.

Fine point: Unlike points, words, lines, and paragraphs, the underline showing that you have selected an entire text does not track the cursor while you hold Left down. (In typescripts, for instance, you can triple-Left-click to select one file and then drag to another; the first file is still selected.)

A.1.3 Using the mouse to adjust a selection

You use the cursor and the Right mouse button to lengthen or shorten a selection. You can make these adjustments at any time—immediately after you've made the initial selection or at any time thereafter—and you can make as many adjustments as you like.

Once you've chosen a unit with the Left or Middle button, the Right button uses that same unit for adjusting the selection. So for instance if you made your selection by double-clicking Middle, which gives you a paragraph, then Right adjusts the selection by increments of paragraphs. If you made your selection by clicking Left, which gives a point, then Right adjusts the selection in increments of one character.

To extend a primary selection, point to where you want the selection to end and click Right.

The highlight for an adjusted primary selection depends on whether the text containing the selection is read-only or editable. In read-only text, the highlight is the usual black underline. In editable text, an adjusted primary selection is called a *replace selection* (short for "replace-mode primary selection") and highlighted with reverse video. As you can guess from the word "replace" and from the garishness of the reverse-video highlight, replace selections have interesting properties.

Often you will click to adjust a selection, but you can also adjust a selection more deliberately by pressing Right, adjusting the cursor position until the correct selection is highlighted, and then releasing Right. In dragging, you may notice that `TextPort`'s adjustment behavior varies subtly according to whether you press Right before, inside, or after the old selection.

You can adjust either end of the selection, but not both ends with the same Right click or Right drag. Most users find it natural to adjust the selection by extending it toward the end of the buffer, and the adjustment mechanism is biased toward that case.

It is perfectly OK to adjust a selection more than once.

A.1.4 Changing the selection unit

(Special section only for the nimble-fingered:)

Occasionally you want to start a selection in one unit, but adjust it in another. `TextPort` allows you to decrease the selection unit while you are holding down on the Right button by "tapping" Right (making a quick up-down button movement). Each tap decreases the selection unit size: file goes to paragraph, paragraph to line, line to word, and word to point.

Suppose, for instance, that you want to select a paragraph and the first sentence of the following paragraph. You start by double-clicking Middle somewhere inside the first paragraph. Then you hold down on Right and tap to reduce the selection unit from paragraphs to lines; tap again to reduce the selection unit from lines to words. Now move the cursor to the end of the sentence at the top of the second paragraph and click Right. Tapping is something you resort to when you haven't planned ahead, especially in dealing with selections that are too big to fit in a single view.

A.1.5 Typing at a replace selection

As you'll recall, if you adjust a primary selection in an editable `TextPort`, the highlight is reverse video instead of the usual black underline. A primary selection in replace mode—"replace selection" for short—stands ready for you to replace it with something else.

The simplest use you can make of a replace selection is to type at it. The selection disappears and the new characters appear. As you type them, the type-in point moves along apace.

A.1.6 Type-in point

The type-in point is that blinking vertical bar that shows where characters are inserted when you type. If you have a primary selection, the type-in point is located at one end or the other of the primary selection. If the primary selection is empty (because, for instance, you've just deleted it), the type-in point is located by itself in your text, in between characters.

You set the type-in point whenever you make a single-unit selection using Left or Middle. It is at the beginning of the primary selection unless you made the selection by single-clicking Middle. In that case it is at the end of the primary selection closer to where you let the Middle button up. If you point near the middle of a nice long word with Middle down and move the cursor back and forth, you will see the type-in point jump from one end of the word to the other. This is called the "balance-beam" rule for locating the type-in point.

You can reduce the need for careful pointing by using selection modes to set the type-in point. Often you want to put your type-in point at the start of something that is a selection unit; just point at the unit rather than pointing between characters. For instance, to place the type-in point at the start of a paragraph, point anywhere in the paragraph and double-click Middle. `TextPort` treats words specially because you place the type-in point at the end of a word just as often as you place it at the beginning. To place the type-in point at the end of a word, point near the end of the word and click Middle.

If the type-in point is on the screen and typing (or deleting) a character would move it offscreen, `TextPort` scrolls in order to keep it onscreen. If the type-in point is offscreen, typing (or deleting) a character makes `TextPort` scroll to put it onscreen.

A.1.7 Making secondary selections with the mouse

Minor selection with the mouse works just like primary selection except that you hold the Control key down as you're clicking or dragging.

The highlight for secondary selection is a grey underline. A point-sized secondary selection is a black vertical bar between characters.

Hold the Control key down and use the Left and Middle buttons to secondary-select the same units as for primary selection:

click	Left	to select a point between characters
double-click	Left	to select a single line
triple-click	Left	to select an entire file
click	Middle	to select a single word
double-click	Middle	to select a single paragraph
triple-click	Middle	to select an entire file

Just as with the primary selection, you can secondary-select by clicking or by using a more deliberate dragging motion.

To extend a secondary selection, hold the Control key down, point to where you want the selection to end, and click Right. (You're probably holding the Control key down already, since you've probably just made the secondary selection. But if you want you can adjust either the primary or the secondary selection at any time.)

A.1.8 Using the secondary selection and the type-in point

To *copy* text from one place to another, place the type-in point where you want the text to go. Minor-select the text to be copied. Then give the Copy command, which is bound to the keystroke control-W. Or use the other order: First secondary-select, then place the type-in point, then do Copy. A copy of the text will appear where the type-in point was, the type-in point will go down to the end of the copied text, and the copied text will be primary-selected. The highlight disappears from the secondary selection.

Moving text from one place to another is just like copying it: Place the type-in point where you want the text to go, secondary-select the text to be moved and give the Move command, which is bound to the keystroke control-E. Or use the other order: First secondary-select, then place the type-in point, then do Move. The text will appear where the type-in point was, the type-in point will go down to the end of the moved text, and the moved text will be primary-selected.

When you use Copy, the highlight disappears from the secondary selection. But the secondary selection is still defined; only its highlight is gone. This means that you can use Copy to copy the same text to different places without re-selecting it.

It may seem strange that the highlight disappears but the selection is still defined. Most of the time you make a secondary selection and use it only once. This argues for taking down the highlight, since it is annoying to have the extra highlight cluttering up the screen. But sometimes the ability to copy the same text repeatedly without re-selecting it is very useful. This argues for leaving the selection defined.

A.1.9 Making a secondary selection from the primary selection

Sometimes the text that you want to copy or move is primary-selected, rather than secondary-selected, at the time you want to use it. So there's a SwitchSelections command, which is bound to the keystroke control-H. When you type control-H, the primary-selection highlight will be replaced by the grey secondary-selection underline, and the grey secondary-selection underline will change to the primary selection highlight.

There are several things that you can do to a primary selection that you can't do to a secondary, so a useful idiom is control-H do-something-to-the-selection control-H. For instance, you can't normalize the secondary; but you can do control-H control-SpaceBar (Normalize) control-H.

And suppose that your secondary selection highlight has disappeared because you've used the secondary in a Copy or a Swap. The way to get the highlight back is to do control-H control-H.

A.1.10 Making a replace selection into a non-replace selection

If you have a replace selection and want to make it a non-replace selection, type control-Y. This moves the cursor from one end of the primary selection ot the other, and it also takes the selection out of replace mode.

A.1.11 Restoring the primary selection to a window

Sometimes you make a primary selection in one window, then shift your attention to another window and make a primary selection there. The first window loses the primary selection, but remembers where it was. If you want to return to what you were doing in that window, just Right-click anywhere inside of it. This re-selects the most recent primary selection that was made there. If this selection is not visible, the window scrolls to make it visible.

A.2 The EmacsModel Interface

```
INTERFACE EmacsModel;
```

```
IMPORT TextPortClass;

TYPE T <: TextPortClass.Model;

END EmacsModel.
```

In the Emacs model, there is only a primary selection, and it corresponds to `VBT.Source`.

The model supports a single *region*, which is delimited by the "mark" and the "point". Control-space and control-@ set the mark; the point is the same as the current cursor position, which is changed both by mouse-gestures and by control-keys. When the region is defined by control-keys, it is not highlighted.

A left-downclick also sets the point. If you then drag the mouse, the location of the downclick becomes the mark, and the point is set to the current position of the mouse. When the region is defined by dragging, it is highlighted.

The Cut and Copy commands make a copy of the text in the region; it becomes the source selection.

The control- and meta-keys in the Emacs model are not case-sensitive; control-shift-a, for example, has the same effect as control-a. "meta" keys are those whose `modifiers` field contains `VBT.Modifiers.Option`, as well as those that follow the escape key. For example, the two-key sequence of escape followed by "b" is "meta-b", just like the single, modified key that Trestle reports as "option-b".

control-space	set the mark
control-a	move to the beginning of the line
control-b	move to the previous character
meta-b	move to the previous word
control-d	delete the next character
meta-d	delete the next word
control-e	move to the end of the line
control-f	move to the next character
meta-f	move to the next word
control-h	delete the previous character, and move left
meta-h	delete to the start of the current word
control-i	invoke the `tabAction` callback
control-j	insert a newline
control-k	delete to the end of the line, and make that the source selection
control-m	invoke the `returnAction` callback
control-n	move down one line
control-o	insert a newline without moving the cursor
control-p	move up one line
control-q	insert the next character ("quoted insert")
control-r	search backward for the current source selection
control-s	search forward for the current source selection
control-t	swap the current and previous characters
control-v	scroll up one screen
meta-v	scroll down one screen

control-w	**Cut**
meta-w	**Copy**
control-y	**Paste**
control-z	scroll up one line
meta-z	scroll down one line
control-_	**Undo**
meta-_	**Redo**
meta-<	move to the beginning of the buffer
meta->	move to the end of the buffer
meta-leftArrow	move to the previous word (like meta-b)
meta-rightArrow	move to the next word (like meta-f)

A.3 The MacModel Interface

```
INTERFACE MacModel;

IMPORT TextPortClass;

TYPE T <: TextPortClass.Model;

END MacModel.
```

The Mac model is taken from Apple's *Human Interface Guidelines* [**?**, pages 106-114]. It supports a single, primary selection that corresponds to `VBT.Source`.

The first unmodified downclick establishes the *anchor point*. If the user then drags the mouse, the upclick establishes the *active end*; the range between the anchor point and the active end is called the selection, and it is highlighted. If the user releases the mouse without dragging, that establishes the *insertion point*, and there is no selection or highlighting.

Shift-downclick extends (or reduces) the selection and establishes the new active end.

Double-clicking selects a word; dragging after a double-click extends the selection in word-size increments.

The Apple guidelines state:

> When a Shift-arrow key combination is pressed, the active end of the selection moves and the range over which it moves becomes selected. . . . Option-Shift-Left Arrow selects the whole word that contains the character to the left of the insertion point (just like double-clicking on a word.)
>
> In a text application, pressing Shift and either Left Arrow or Right Arrow selects a single character. Assuming that the Left Arrow key was used, the anchor point of the selection is on the right side of the selection, the active end on the left. Each subsequent Shift-Left Arrow adds another character to the left side of the selection. A Shift-Right Arrow at this point shrinks the selection.

> Pressing Option-Shift and either Left Arrow or Right Arrow ... selects the entire word containing the character to the left of the insertion point. Assuming Left Arrow was pressed, the anchor point is at the right end of the word, the active end at the left. Each subsequent Option-Shift-Left Arrow adds another word to the left end of the selection...
>
> When a block of text is selected, either with a pointing device or with cursor keys, pressing either Left Arrow or Right Arrow deselects the range. If Left Arrow is pressed, the insertion point goes to the beginning of what had been the selection. If Right Arrow is pressed, the insertion point goes to the end of what had been the selection.
>
> [From page 83] When the user chooses Cut, ... the place where the selection used to be becomes the new selection. ... In text, the new selection is an insertion point [and the highlighting is removed].
>
> Paste ... inserts the contents of the Clipboard [VBT.Source] into the document, replacing the current selection [i.e., selections are always replace-mode]. If there is no current selection, it's inserted at the insertion point. ... After a Paste, the new selection is ... an insertion point immediately after the pasted text. [In either case, there is no highlighting.]

If there is a selection, then Copy (and therefore Cut) call

```
SELF.takeSelection(TextPort.SelectionType.Primary, ...)
```

to acquire `VBT.Source`, and the selection is copied to a local buffer ("clipboard"). No other procedure in this interface acquires `VBT.Source`. MacModel selections are always in replace-mode.

The Mac model supports only a few keybindings. `meta` refers to the `option` modifier.

control-c, meta-c	**Copy**
control-v, meta-v	**Paste**
control-x, meta-x	**Cut**
control-z	**Undo**
control-shift-z	**Redo**

A.4 The XtermModel Interface

```
INTERFACE XtermModel;

IMPORT TextPortClass;

TYPE T <: TextPortClass.Model;

END XtermModel.
```

The Xterm model supports a single selection and mouse-events in the style of `xterm(1)`. It is a simple model; it is not "programmable", nor is it influenced by commands in the user's `.Xdefaults` file.

The implementation supports only the primary selection, which corresponds to `VBT.Source` (the same as X's primary selection).

A single-left-click establishes the keyboard focus and insertion point, but it does not change (acquire) the selection. A double-left-click selects the current word; a triple-left-click selects the current line. More clicks rotate among these three options.

Single-left-click and drag selects a range of characters. Double-left-click and drag selects a range of words, and triple-left-click and drag selects a range of lines.

Middle-click pastes the current source selection at the insertion point, which need not be at the end of the text (as it would be for a "typescript"). The xterm selection is never in replace-mode.

Right-click extends the current selection, re-highlighting it if needed.

The shift key has no effect on the mouse in xterm-mode; it is ignored, so that shift-left-click, for example, has the same effect as left-click. The control and meta ("option") keys, however, are not ignored; they cause the mouse-clicks to be no-ops, and they have different keybindings. Control-left-click, for example, has no effect.

The only keybindings that are supported are these:

control-u	delete everything from the current position to the beginning of the line
control-z	**Undo**
control-shift-z	**Redo**
meta-x	**Cut**
meta-c	**Copy**
meta-v	**Paste**

`meta` refers to the `option` modifier. Note that Copy does very little: as soon as there is a selection (e.g., via double-clicking or dragging the mouse), it automatically becomes the source selection; there is no separate clipboard in this model.

References

[1] Adobe Systems Inc. *Postscript Language Reference Manual, second edition*. Addison-Wesley Publishing Co., 1990.

[2] Mark S. Manasse and Greg Nelson. Trestle reference manual. Technical Report 68, DEC Systems Research Center, December, 1991.

[3] Mark S. Manasse and Greg Nelson. Trestle tutorial. Technical Report 69, DEC Systems Research Center, May 1, 1992.

Index