eSWT User Interface - High Level Design

Abstract

There is a need for a standardized UI API fit for embedded devices with less than desktop level resources and screen sizes. Herein is proposed a subset of the Standard Widget Toolkit which meets this and other embedded device design requirements.

Document Information

Author: Mark Rogalski
Email: Rogalski@us.ibm.com
Phone: 512-838-3512

Change History

0.9.0	7/12/04	First draft
0.9.1	8/23/04	Incorporation of recommendations from SWT team, alphabetized widgets
0.9.2	9/01/04	Incorporation of feedback from Nokia and Motorola
0.9.3	9/16/04	Additional feedback from SWT and eSWT teams
0.9.4	9/24/04	Incorporate results of eSWT kick-off meeting
0.9.5	9/28/04	Accept text changes, remove satisfied comments

1.1 Introduction

The purpose of this document is to define an embedded device oriented UI functional design that applications use to interact with the end-user. This includes rendering to the screen and receiving input from the user. To some extent it may also specify ancillary features used to obtain UI data about devices, for example, determining screen capabilities. After careful consideration of the requirements of various embedded devices, we propose that a subset of the Standard Widget Toolkit (SWT) can mostly fulfill this need, the working name of which shall be Embedded SWT (eSWT). Function that is not a subset of SWT is set forth in *the Mobile Extensions for SWT* document.

1.1.1 Scope

The API selection is primarily concerned with meeting the functional requirements of embedded devices and with the size and performance of the API. Secondary concerns are support for existing applications, cross-platform portability, and extendibility to support device features. The usability and performance of applications is affected by the choices made in this specification. Application development efficiency, tools capabilities, and developer acceptance are also affected.

The goal is to produce a complete specification that satisfies all high level cross platform requirements outlined below and to support a well-defined mechanism for extending the API for lower level requirements which may not be of interest across all platforms. The specification does not address operational management concerns such as application install/uninstall or inter-application communication. Neither does this document address the user interface of the device or system applications.

1.2 Use Cases or Scenarios

1.2.1 Requirements Summary

Rich UI – support high function/high response widgets

Suitable to target devices – must be low resource, high performance; must adapt to different input devices and screen sizes

Portability - allow apps to run (unmodified) on varied device classes

Compatibility – do not preclude the use of AWT or LCDUI applications, utilize existing competencies and tools

Extendable – allow custom/shared widget/API libraries, but also provide a predictable and usable set of APIs that can be assumed to be present all the time.

Adaptability – provide automatic adaptation for screen size, keypad, etc. for devices in same class and allow programmatic adaptation for devices in different classes

Look and Feel – provide look and feel of native platform

Graphics – support basic and advanced 2D graphics, image manipulation

Multimedia – support display/playback of media content

Multimodality – support speech recognition and output (lower priority)

Internationalization – support apps running using varied languages

1.3 High-Level Model

The eSWT UI architecture is designed so that the API can be implemented as OSGi bundles, thereby allowing managed updates of API packages. However, it should be understood that eSWT does not require OSGi to be present. Updates might include revisions/additions to existing API or completely new API components. However, eSWT always provides a core API that applications can rely on and can be used in any application architecture that has support for the full Java VM specification.

eSWT is separated in to manageable packages that provide different UI widget sets and UI functionalities:

- Core eSWT
- Expanded eSWT

There is also separate package called *Mobile Extensions for SWT* that does not belong to eSWT but is closely related as it is dependent on the Core eSWT package. Mobile Extensions are defined in a separate document: *Mobile Extensions for SWT- High Level Design*.

1.3.1 UI Architecture

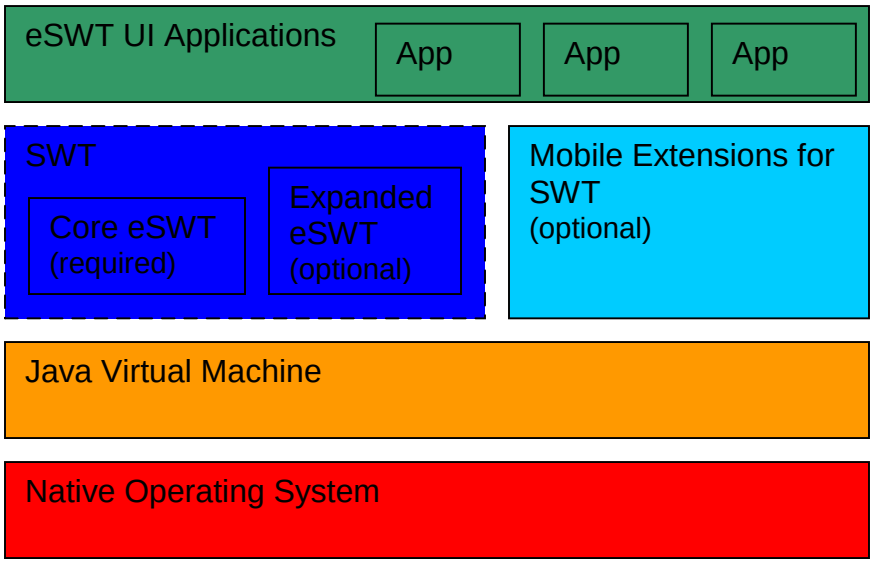

Figure 1— UI Architecture

The above diagram represents how eSWT and Mobile Extensions for SWT fit into the overall programming architecture. It shows that eSWT is a subset of SWT made up of two components: the Core portion which is always present on devices supporting eSWT applications, and the Expanded portion which is optionally present. Mobile Extensions for SWT is a separate optional component.

1.4 Package Decomposition

eSWT is the subset of SWT API which is sufficient to support AWT libraries and MIDP containers, plus a few additional widgets which will be driven back into base SWT.

The minimum Java platform required to run eSWT is CLDC 1.1, however, the footprint requirements of eSWT may make its use very limited on devices that only support this Java configuration. The following sections describe the major aspects of eSWT.

1.4.1 Component Styles

A widget can have different styles or style combinations when it is constructed. The use of styles provides benefits in terms of application flexibility and portability, because the same GUI widget can have different styles in look and feel.

Styles are widget or context specific. This means that widgets define which styles can be used in them. Also the concrete behavior that results when a style is used in a widget may be different from implementation to implementation. Applications should only rely that a device will implement the semantics defined in eSWT for a style. The semantics of different styles are defined later in this specification inside the Widgets and Advanced Widgets sections.

- GUI control style groups:
  - Widget look and feel styles:
    - CASCADE: a sub-menu style in a menu item.
    - SEPARATOR: a space line on the screen, or a separator line in a menu.
    - CHECK: a check style with two states: checked and unchecked.
    - PUSH: an action style that the user can click.
    - RADIO: an exclusive style with two states: selected and deselected.

    Note: There may be a need for one or two styles which hint to the eSWT implementation whether the widget should be edited in-line (as on a desktop platform) or via a popup. This area needs to be investigated further. See Key-only Traversal Control section for more information on this topic.
  - Functional styles:
    - DROP_DOWN
    - READ_ONLY
    - SINGLE: a single selection style.
    - MULTI: a multiple selection style.
    - FULL_SELECTION: a row selection style (only for Table in Expanded Widget Set)
  - Modality hints (see Section Windows and Containers for details):
    - MODELESS
    - APPLICATION_MODAL
    - SYSTEM_MODAL
  - Orientation styles: HORIZONTAL, VERTICAL, LEFT_TO_RIGHT, RIGHT_TO_LEFT
  - Visual effect styles: TOP, BOTTOM, LEFT, RIGHT, CENTER, WRAP

- Window decoration styles

  High-level window decoration styles:

  - SHELL_TRIM
  - DIALOG_TRIM
  - NO_TRIM: no window decorations

  Low-level window decoration styles:

  - CLOSE
  - TITLE

- Widget decoration styles: BORDER

Some of the styles are mutually exclusive. For example, we want to have a separator with border placed in the center, the style will be: SEPARATOR | BORDER | CENTER but you cannot combine CENTER and LEFT. The eSWT specification will define the behavior when styles are combined in conflicting ways. The final appearance of a widget may vary from device to device. The exact behavior of styles is device-dependent.

1.4.2 Top-level Application Window

In many embedded devices there are no window resize capabilities, and no partially overlapping windows. Often only one window is visible at a time and it takes the whole screen space. Often users cannot resize the top-level window.

An eSWT top-level window consists of a top-level UI container (See Section 1.4.10, ***Windows and Container*** for details) with a title pane and possibly a menu bar and/or a softkey pane in some platforms for mobile devices (see Mobile Extensions for SWT for details).

The title pane contains the icon and name of the application. Both the icon and the name are changeable by applications.

The menu pane contains labels for menus and sub-menus. The position of the menu bar and other panes is eSWT implementation dependent. For example, in many small screen devices the menu bar is normally hidden and made visible only when the user activates the menu.

Applications can work in three different UI modes (defined by styles used for the top-level UI container).

- Normal (TITLE): Implementation makes visible the normal window decorations found in applications.
- Reduced (NO_TRIM): Applications are given a larger working space compared to normal by hiding or reducing the size of decoration panes.
- Full-screen (NO_TRIM): Applications can use the whole screen area. Any window decorations are hidden. Note: Investigation is required to determine how this mode is activated/deactivated by the user, and whether programmatic control resides with the application or app manager.

In addition to the above high-level window decoration styles, there are more low-level styles to control some window decorations directly. The low-level decoration styles and high-level styles must not be used together. The eSWT specification must clearly state what the consequences are in such an event.

1.4.3 Component Characteristics

Applications can define and update the UI characteristics of any widget. UI control features:

- Background color: sets the widget's background color to the color specified by the argument, or to the default system color for the control if the argument is null. Apps should not change the background color indiscriminately because this may violate the native LaF control, ***e.g.*** skinning and themes.
- Size, Location, and Bounds: sets the widget's size and location to the rectangular area specified by the arguments. Bounds contains both size and location.
- Enabled/Disabled: enables the widget if the argument is true, and disables it otherwise. A disabled control is typically not selectable from the user interface and draws with an inactive or "grayed" look.
- Focus: sets the receiver to have the keyboard focus, such that all keyboard events will be delivered to it.
- Font: sets the font that the widget will use to paint textual information to the font specified, or to the default font for that kind of control if the argument is null.
- Foreground color: sets the widget's foreground color to the color specified by the argument, or to the default system color for the control if the argument is null. Apps should not change the foreground color indiscriminately because this may violate the native LaF control, ***e.g.*** skinning and themes.
- ToolTip: sets help text for a widget to be displayed upon some UI activation mechanism. Usually a tip will appear and disappear automatically in a short time. The UI mechanism which displays tips is implementation dependent, and may not be implemented on some platforms. The action does not have to be a mouse over event and the tip does not have to appear as hover help over the widget as on desktop platforms.
- Visible: sets the widget visible or invisible. When the control is the top-level window (aka a Shell), making the shell visible will cause it to be displayed. An application may explicitly make itself visible from background to foreground and gain the focus. See Section 1.6.1, ***Permission Control*** for some constraints about the top-level application window.

1.4.4 Layout Management

A Composite is a class that holds one or more UI components. Composite has a Layout object which is responsible for laying out the contained components. If no layout is set by the application (layout is null) then the components can be sized and positioned inside the composite by their absolute coordinates relative to the composite origin. When layout is null, the sizes and positions of UI components are determined by applications and remain unchanged when the container is resized. Using absolute coordinates the UI becomes easily non-scalable for different devices unless the application recalculates layout.

A layout control mechanism (aka. a layout manager) can provide some scalability to different device classes at run-time, and ease development of UI layouts at design time for developers; especially when a GUI builder is not present or used.

The Composite class provides the ***computeSize*** function to calculate the sizes, and layout function to position all the UI components inside the composite object. A composite can contain another composite, or many composites. The layout manager in effect for the top composite determines how the sub-composites are sized (or resized), and positioned inside the top composite, and the layout manager in effect for each sub-composite determines how components are sized and positioned inside each sub-composite.

eSWT supports layout classes for the purpose of layout computing and positioning, in additional to absolute positioning:

1. FillLayout: lays out controls in a single row or column, forcing them to be the same size. This layout has no configuration for each child, and no other features like the ability to wrap. This layout is only in the Expanded eSWT package.
2. FormLayout: controls the position and size of the children of a container component by using FormAttachments class to optionally configure the left, top, right and bottom edge of each child.
3. GridLayout: lays out the children of a container in a grid. Each child can have an associated grid control class called GridData class to configure its size and position. This layout is only in the Expanded eSWT package.
4. RowLayout: similar to FillLayout, lays out controls in a single row or column. Each child can use an associated RowData class to define configurable margins and spacing. In addition, the height and width of each control in a RowLayout can be specified. This layout is only in the Expanded eSWT package.

1.4.5 Colors

eSWT provides a way to acquire system default colors (Display. getSystemColor). The color instance should not be disposed explicitly because it was allocated by the system, not the application[1].

System default colors:

- COLOR_WHITE, COLOR_BLACK, COLOR_RED, COLOR_DARK_RED, COLOR_GREEN, COLOR_DARK_GREEN, COLOR_YELLOW, COLOR_DARK_YELLOW, COLOR_BLUE, COLOR_DARK_BLUE, COLOR_MAGENTA, COLOR_DARK_MAGENTA, COLOR_CYAN, COLOR_DARK_CYAN, COLOR_GRAY, COLOR_DARK_GRAY

In addition to the capability of using absolute colors, apps can use logical application colors to utilize a consistent UI color schema of the underlying platform.

System component colors:

- COLOR_WIDGET_BACKGROUND: System color used to paint background areas.
- COLOR_WIDGET_BORDER: System color used to paint border areas.
- COLOR_WIDGET_FOREGROUND: System color used to paint foreground areas.
- COLOR_ WIDGET_HIGHLIGHTED_BACKGROUND: The color for the focus, or focus highlight, when it is drawn as a filled in rectangle. The highlighted background will always contrast with the highlighted foreground.
- COLOR_ WIDGET_HIGHLIGHTED_BORDER: The color for boxes and borders when the object is to be drawn in a highlighted state. The highlighted border color is intended to be used with the background color (not the highlighted background color) and will contrast with it.
- COLOR_ WIDGET_HIGHLIGHTED_FOREGROUND: The color for text characters and simple graphics when they are highlighted. Highlighted foreground is the color to be used to draw the highlighted text and graphics against the highlighted background. The highlighted foreground will always contrast with the highlighted background.

[1] Application misbehavior like disposing a system default font must not cause the halt or crash of the whole system.

Custom defined colors:

- Custom colors may be defined by specifying RGB integer values directly. All custom colors must be disposed manually.

Note: The widget parts colors mentioned above may not be wholly sufficient to create custom widgets which have he same look as natively skinned widgets. Investigation is required to determine if a “skin palette” may be a better method of retrieving native skinning information.

1.4.6 Fonts

eSWT provides one device default font (Display.getSystemFont) for applications to use. There is also need for an overloaded getSystemFont method to return an appropriate font depending on the context. The overloaded method has a parameter defining the logical specifier of a context. A complete list of context specifiers is yet to be defined but shall include at least the following:

- normal system font (same as default font),
- text editor font,
- title font, and
- button font

The default system font will be used if an explicit context font is not provided on a platform. Fonts returned from getSystemFont should not be disposed explicitly because they are allocated by the system, not the application.

Custom fonts may be constructed by providing a device and either name, size and style information or a FontData object which encapsulates this data. All custom fonts must be disposed manually.

Custom font styles:

- BOLD
- ITALIC
- NORMAL.

FontData is used to query what font types are available on the system.

FontMetrics is used to measure a specific font about its ascent, descent, height, leading space between rows, and average character width. Instances of FontMetrics are obtained from the graphics context (GC) using the getFontMetrics() method.

1.4.7 Widgets

The following widgets are provided in the Core eSWT package:

- Button: A Button allows user interaction like pressing and releasing. An icon can be attached to the button with different alignment styles such as “left”, “right”, “top”, and “bottom”.
    - Styles:
        - PUSH: Normal button;
        - CHECK: A check box is a graphical component that can be in either an "on" (true) or "off" (false) state. Clicking on a check box changes its state from "on" to "off" or from "off" to "on", and fires the state change event to its associated event listeners if available, no matter what its previous status is.

        - RADIO: A radio box is a graphical component that can be in either an "on" (true) or "off" (false) state depending on other RadioBoxes in the same radiobox group. In mobile devices, because of small device screen, usually one radiobox group is used within their parental container. Clicking on a radio box gets the focus and fires the state change event to its associated event listeners if available, no matter what its previous status is.
    - User interaction: Action/Ok to push/toggle button
    - Traversal control: Arrow keys to move focus to/from button

- Canvas: Provides a surface for drawing arbitrary graphics.
    - Styles: None
    - User interaction: None or application specific
    - Traversal control: None or application specific

- Combo: A special type of text field that combines a text input and a drop-down or pop-up list. A Combo box allows users to either type in a value or select a predefined value from a list that is displayed when the user asks for it. In some situations, a Combo control can hide the selection list. Users can use navigation keys or buttons associated with the widget to select the text item provided by the combo control.
    - Styles:
        - DROP_DOWN: by default an editable input control with a drop-down list.
        - READ_ONLY: a non-editable input control with a drop-down list. The value can be changed from the list but not freely editable.
    - User interaction: Up/Down to drop down list and change selection within list; Action/Ok to choose selected item
    - Traversal control: Arrow keys

- DateEditor: A special editor widget allows users to input or edit instance of date or time data. The return value is an instance of a Date class.
    - Constraint styles:
        - DATE
        - TIME
        - DATE_TIME
        - DURATION: display a length of time in hours, minutes and seconds.
        - OFFSET: a time in hours, minutes, and seconds which can be subtracted or added to another time value.

- HOURS, MINUTES, SECONDS styles define the fields available in OFFSET, DURATION, TIME and DATE_TIME (time component). The default is that all hours, minutes and seconds fields are visible in editor.

§ Initialization parameters for the editor:

- Date: an instance of date/time in UTC.
- TimeZone: defines the time zone, which is applied on Date object (which is UTC time). The date and time presented in the user interface of DateEditor is a local time in this time zone. Can be null, which defaults to currently active time zone of the device.
- Locale: defines the locale for the date and time formatting. Can be null which defaults to current locale of the device.

§ User Interaction: When selected, widget may provide popup for data selection or editing.

§ Traversal Control: Arrow keys once popup is closed.

- Label: A Label is a string of chars that does not require any input from users. The main purpose of Label is to display information. A Label can contain an icon.. The position of an icon uses the same alignment parameters as Button does.

§ Styles:

- HORIZONTAL
- VERTICAL
- SHADOW (in/out)
- SEPARATOR: A Label control can act as a Separator (Spacer), a blank, non-interactive UI component that has a variable minimal size.

§ User interaction: None

§ Traversal control: None

- List: A List contains an ordered collection of strings. A list issues events when an item is selected (highlighted). The indication of a scrolling list can be implemented as a scroll bar or scrolling indicator, which indicates the relative position of the list item in focus within the list. Usually the scrolling indicator is done and controlled by specific implementation. If an alphanumeric key is entered while the list has focus, the implementation may optionally scroll the list to highlight an item beginning with that character.

Note: Native list widgets on some platforms do not support sorting. However, it is desirable for embedded applications to be able to specify the list be automatically sorted in order to 1) reduce replication amongst applications of common code and 2) enable the sorting to be done natively where it can be done more quickly. Since footprint and speed are not as relevant issues on desktop platforms, desktop implementations may choose to ignore sorting styles in favor of letting the application do all sorting. Since sorting is a rather critical function, we need to determine whether we can specify that sorting styles must be implemented on eSWT implementations but can be treated as hints for desktop implementations.

- Selection Styles:
  - SINGLE: single selection mode
  - MULTI: multiple selection mode
- Sort Styles:
  - SORT_ ASCENDANT
  - SORT_DESCENDANT
- Special styles:
  - QUERY: show a query prompt for filtering list items. As the user adds more characters in the query field, the list is filtered to show fewer items matching the characters entered. This is a style hint and may not be implemented on all platforms.
  - WRAP: wrap the list element text when their lengths are longer than the width of the list. This is a style hint and may not be implemented on all platforms.
- User interaction: Up/Down to move selection within list; Enter/Select to toggle selection to/from chosen state
- Traversal control: Arrow keys

- Menu: A widget contains menu items. A menu can have different semantics in different devices. For instance, a menu can represent the window-style menu bar to hold sub menus, or a pop-up main container. Usually a menu has no direct user interaction associated.
  - Styles:
    - BAR: standard menu item container
    - DROP_DOWN (sub-menu)
    - POP_UP

- MenuItem: A menu item represents the selectable widgets in a menu. In different devices menu items can have different styles and functionalities. Menu items use Menu as their parental container. In some devices, the corresponding concept of menu items is interpreted as commands.
  - Styles:
    - CASCADE

        - CHECK
        - RADIO
        - SEPARATOR
        - PUSH
    - User interaction: Arrow keys to navigate; Select key to select
    - Traversal control: Arrow keys

- MessageBox: a dialog used to inform the user of limited information using a standard style. It can display several standard buttons and return which button was selected by the user. eSWT implementation may utilize softkeys rather than place buttons on a dialog.
    - Message box semantic styles:
        - ICON_NONE
        - ICON_ERROR
        - ICON_INFORMATION
        - ICON_QUESTION
        - ICON_QUERY: a dialog contains prompt text and a data input widget.
        - ICON_WARNING
    - Action styles:
        - OK, OK | CANCEL
        - YES | NO, YES | NO | CANCEL
        - RETRY | CANCEL
        - ABORT | RETRY | IGNORE
    - User interaction: softkeys or select key
    - Traversal control: None.

- ProgressBar: A ProgressBar is to display the progress of an operation.
    - Styles:
        - HORIZONTAL
        - VERTICAL
        - INDETERMINATE
    - User interaction: None
    - Traversal control: None

- ScrollBar: A ScrollBar represents a range of positive numeric values. Typically, scroll bars are used to query the value. To manually change the value, use Slider widget instead.
    - Styles:

    - HORIZONTAL
    - VERTICAL
  - User interaction: Arrow keys
  - Traversal control: Arrow keys

- Slider: A Slider defines a range of numeric values and issues update events to associated components or handlers. Slider can be made up of five areas:
  - an arrow button for decrementing the value
  - a page decrement area for decrementing the value by a larger amount
  - a *thumb* for modifying the value by mouse dragging
  - a page increment area for incrementing the value by a larger amount
  - an arrow button for incrementing the value

  In some device classes, a scrollbar can be represented as a small scroll indicator, which may have no thumb and page indicators. So any changes made to thumb have no effect at all.
  - Styles:
    - HORIZONTAL
    - VERTICAL
  - User interaction: Arrow keys
  - Traversal control: Arrow keys

- Text: A Text widget allows users to input or edit single or multiple lines of text. The implementation is very much device-specific. For instance, a multi-line text input field can be implemented in a full-screen mode in a separate window to enable easier traversal control (simply using arrow keys), or only single line text editing may be allowed, that is, using left-right arrow keys for editing, up/down arrows for focus-in/out.

  For embedded device there is a great need to be able to hint to the implementation the type of content the widget is expected to contain. This allows the implementation to choose the appropriate input mode for devices that have limited input function. For instance, if the content is expected to be only digits, the input mode is selected such that keypad buttons produce only digits. If a person's name is expected, the input mode is selected such that keypad buttons enter alpha characters. If an email address is expected, the input mode may use predictive completion based on past addresses entered. While Shell.setImeInputMode is provided in SWT, this method is insufficient since widgets within a shell may all have different input mode requirements.
  - Input mode styles:
    - TEXT: Normal alphanumeric text entry, may be predictive
    - EMAILADDR: email address
    - NUMERIC
    - DECIMAL

- PHONENUMBER
- URL

Note: “setEchoChar” is allowed only in NUMERIC and TEXT styles.

For TEXT input mode style, the following substyles can be combined:

- Allowed casing modes[2] in the TEXT style editor (users are allowed to use the specified casing modes only):
  - UPPER_CASE
  - LOWER_CASE
  - TEXT_CASE: first character only capitalized
  - TITLE_CASE: first character of each word is automatically capitalized.
  - DEFAULT_CASE: implementation dependent normal casing mode
- For turning off possible non-predictive text input[3]: NON_PREDICTIVE. By default any predictive input facilities should be turned on if available.
- For telling that locale specific input modes should not be available: LATIN_INPUT_ONLY.
- To have overwrite mode rather than insert: AUTO_OVERWRITE
- Alignment styles: CENTER, LEFT, RIGHT, READ_ONLY, WRAP
- Functional styles:
  - MULTI: multi-line
  - SINGLE: single line
- User interaction: Arrow keys to move the caret; select key to commit the input.
- Traversal control: Arrow keys; select key to get/release the focus.

1.4.8 Expanded eSWT Widgets

The following widgets are provided in the Expanded eSWT package.

- FormattedLabel: A subclass of the Label widget which formats its content to a specific format. Note: More investigation is required to determine in which package this widget best fits.
  - Styles:
    - URL: A Label control can act as a hyperlink to launch an external browser in the platform. The label text is used as the URL for the browser.
    - SEPARATOR: A Label control can act as a Separator (Spacer), a blank, non-interactive UI component that has a variable minimal size.

[2] Effective only if current input language has alphabet casing, e.g. in Latin. Cyrillic, Greek and so on.

[3] Predictive text input: any assistive input technology, e.g. automatic word completion based on initial characters entered.

        - PHONE: A Label control can act as a hyperlink to make a call in the platform, if applicable. The label text, e.g. a numerical text, is used as the dialed number.
        - EMAIL: A Label control can act as a hyperlink to the default email client in the platform, if applicable. The label text is used as the recipient email address for the email client.
    - User Interaction: If Label is a hyperlink, may be activated via Enter/Select
    - Traversal Control: If Label is a hyperlink, may be focused via arrow keys

- ListView: A ListView contains a collection of list items. A list item can be a single line of text, an icon, or text with icon. The difference between List and ListView is that ListView supports icons (images), and displays its items in a user selectable layout. The initial layout is programmatically selected from a list of generic layout style hints. The implementation maps a generic style to a platform specific style that best fits. In all layout styles, the exact sizes of icons and text are platform dependent. The user may dynamically switch layout styles in a platform dependent manner. Sorting style shall behave similarly to the List widget.
    - Layout Styles: Icon and Text styles may be or-ed together. The following styles are subject to revision based on investigation of common platform capabilities.
        - SMALL_ICON: shows small size icons
        - MEDIUM_ICON: show medium size icons.
        - LARGE_ICON: show large size icon.
        - SMALL_TEXT: shows small size icons
        - MEDIUM_TEXT: show medium size icons.
        - LARGE_TEXT: show large size icon.
    - Selection Styles:
        - SINGLE: single selection mode
        - MULTI: multiple selection mode
    - Sort Styles:
        - SORT_ ASCENDANT
        - SORT_DESCENDANT
    - User interaction: Up/Down to move selection within list; Select key
    - Traversal control: Arrow keys

- Table: This widget implements a selectable user interface object that displays a list of items like images and strings; and issues notifications when an item or a row of items is selected.
    - Styles:

- SINGLE: Only single cell/row may be selected
- MULTI: Multiple cells/rows may be selected
- CHECK: Adds check box selection behavior
- FULL_SELECTION: Entire row is selected when any cell in row is selected
- HIDE_SELECTION: Selection is cleared when widget loses focus

§ User interaction: Arrow keys; Select key.

§ Traversal control: Arrow keys; Select key.

- Tree: This widget provides a selectable user interface that displays a hierarchy of items and issues notifications when an item in the hierarchy is selected. The item children that may be added to this widget must be of type TreeItem.
  - § Styles:
    - SINGLE: Only a single item may be selected
    - MULTI: Multiple items may be selected
    - CHECK: Adds check box selection behavior
  - § User interaction: Arrow keys; Select key.
  - § Traversal control: Arrow keys; Select key.

- Web browser: a widget which can be given a URL to display. The HTML/JavaScript capabilities of the widget are dependent upon the native web browser which is providing the rendering.
  - § Styles:
    - Border
    - URL_BAR: show URL display/input field
    - TOOL_BAR: show navigation buttons
    - STATUS_BAR: show status bar
  - § User interaction: pointing device required
  - § Traversal control: pointing device required.

1.4.9 *Key-only Traversal Control*

On some devices, the nesting of certain widgets within containing widgets is likely to cause a navigation problem. For instance, a table widget uses the arrow keys to navigate within the table. Normally, pressing the tab key would move focus out of the table widget to some other widget. However, most mobile devices do not have the tab key, thus making leaving the table widget difficult. For this reason, the developer should be careful about combining some widgets within the same shell.

As a core UI API for all device classes, the minimal requirement for focus traversal must be based only on arrow keys plus a select key. Many devices do not have an "intra-widget" navigation key such as the Tab key on a regular keyboard. It should also be understood that many clients of eSWT do not have pointing devices. Direct focusing of certain areas or widgets in the UI is impossible without a pointing device[4]. Because some advanced widgets may "consume" the events from arrow keys for internal navigation it may be difficult or impossible to transfer focus from widget to widget.

This is generally called the "widget traversal problem" in this document. Because of this problem the implementations may implement certain widgets in a way that there are two separate modes (interaction and navigation modes) for interacting with the widget and for navigating in and out of the widget. Initially a widget is in navigation mode so that pressing arrow keys will directly move in and out of the widget but when the widget is focused there is an operation that allows user to enter a widget interaction mode. In this mode, for example, the content of the widget can be edited and arrow keys are used to move focus inside the widget. Implementations must also provide an operation to leave editing mode and resume navigation mode. All of the user visible operations to change the modes and actual mode changes are done in device implementations and are transparent to applications. This is referred to as "*dual mode interaction*". In order for apps to not have to be concerned about editing modes, the preference is for an eSWT implementation to determine which editing mode is appropriate automatically. If during prototyping or implementation it is found that this is difficult to achieve, the design may change to require that applications provide editing mode hints such as FORCE_INLINE_EDITING or FORCE_POPUP_EDITING to achieve appropriate behavior.

It may also be feasible to for applications to query the availability of an intra-widget navigation key (e.g. Tab key in some platforms), and either not nest advanced widgets or provided some other intra-widget navigation mechanism.

The widget traversal problem affects the following widgets:

- ListView
- Text
- DateEditor
- Table
- Tree
- WebBrowser

1.4.10 Windows and Container

A container component is capable of containing other UI components, including the container component itself. This is known as "nested containment". Top-level application frames (aka. Shell) and dialogs are commonly used containers. In small display devices, it is common to layout all widgets inside a single container. However, for large display devices, an application typically uses containers nested within other containers. A container uses a layout manager or layout methods to specify how its children should be arranged.

[4] Pointing device is a general concept that can be implemented in a device using a mouse or with a touch screen and stylus; some devices have so called "virtual cursor" modes in which the cursor is moved with arrow keys but this is not a very practical solution.

- Composite: This base form of SWT container may be used for building customized controls. It and its subclasses can have a layout style set for controls.
- Group: A special Composite contains title text with special border styles.Border styles[5]: SHADOW_ETCHED_IN, SHADOW_ETCHED_OUT, SHADOW_IN, SHADOW_OUT, SHADOW_NONE
- Dialog: is a special container, usually including some decorations like a title and border. Typically a dialog is used to accept input from the user. This is an abstract class in SWT, allowing programmers to produce customized versions.
    - Modality hints:
        - MODELESS: Normal window
        - APPLICATION_MODAL: All windows owned by the same application are disabled until the window is dismissed.
        - SYSTEM_MODAL: Window must be dismissed before resuming any system activity. Not all native controls support such style. MessageBox may often be the only widget to support such model.

    The modes are specified as hints only because they are device and platform dependent. The modes are not always available in all device classes. APPLICATION_MODAL is the only compulsory modal support required for all devices. In addition, as is the case for top-level windows, the application manager for the device on which the instance is visible has ultimate control over the appearance and behavior of the instance, including its modality.

    Special dialogs:

    - ColorDialog (only in expanded widget set)
    - DirectoryDialog (locate folder; rename folder; only in expanded widget set)
    - FileDialog (locate file; rename file; create file, only in expanded widget set)
    - FontDialog (only in expanded widget set)

- Shell: Top level window which is managed by the window manager. Used for application's main window.
    - Styles**:** BORDER, CLOSE[6], SHELL_TRIM, DIALOG_TRIM, NO_TRIM, TITLE, APPLICATION_MODAL, MODELESS

1.4.11 Event and Event Listeners

The API uses the same event and event listener paradigm as SWT, AWT and Swing. Event listeners handle the corresponding events fired by eSWT components in response to user input or underlying platform activities.

Events:

[5] Border styles will take effect only when the BORDER style has been set.

[6] How the close function is provided is determined by implementations. For example, Implementations can add a Close icon to a window to activate the closing event, or add an Exit menu item automatically in some platforms if the Close icon is not used.

- Component events for all relevant status changes to a UI component, such as resized, moved, and enabled/disabled.
- Pointing device events (mouse) for all relevant pointing device actions that may occur on a touch screen device. Mouse events include MouseUp, MouseDown, MouseDoubleClick, *MouseMove, MouseEnter, MouseExit*
- KeyPressed and KeyReleased events for all relevant key press and release actions, including softkeys. The event tells the number of times the key has repeated (while being held down for an extended period). The application may use this information to determine whether to treat the event as a repeat of the previous event or a completely different semantic event. The app must be able to query the repeating key initial delay time and time between additional key repeats in order to calculate how long the key is being held down.
- Repaint events for actions to re-draw all or partial areas.
- Verify events for actions to verify the changes in text input widgets before committing the changes.
- Traversal and focus events for the focus changes of GUI components.
- Application events for application status changes. For example, the change between foreground and background (SWT.Activate and SWT.Deactivate)

Event listeners are corresponding event interfaces. After creating an instance of a class that implements an event listener interface, it can be added to a GUI component using the *add<EventType>Listener* method and removed using the *remove<EventType>Listener* method. When a corresponding event occurs, the appropriate method in the class will be invoked.

The default event listener implementations, known as event adapters, can provide a skeleton for applications.

1.4.12 Input Methods

1.4.12.1 International Character Input

The text input widgets can define the default device input methods (IME). The change of input methods must return a value to indicate the success of the change.

When necessary, eSWT automatically displays the appropriate input method widget or dialog in order to accept user input.

Methods:

- PHONETIC (phonetic input method used for Katakana text entry)
- ROMAN (Japanese roman character input behavior)

1.4.12.2 Pen input

Since eSWT uses native widgets, any pen based character recognition employed by the native platform is automatically inherited by Java UI widgets.

1.4.12.3 Clipboard support

Since eSWT uses native widgets, text editor and combo define common clipboard operations like copying, cutting and pasting plain text. This enables cut and paste between native and Java apps and amongst Java apps (as long as the native platform supports clipboard operations).

However, applications may require more advanced clipboard operations like copying and pasting a data entry in a calendar application. This kind of operations can, for example, be activated from the application menu items like “Copy” and “Paste”. These features require applications to have a direct access to a clipboard object. As well as the support of clipboard, an abstract data object [7] used by the clipboard must be defined accordingly. SWT provides “*org.eclipse.swt.dnd*” package for clipboard and drag and drop functionalities, eSWT can subset the clipboard and relevant data classes (*TransferData*). Not all platforms may provide Clipboard support.

1.4.13 Imaging and Drawing Support

eSWT provides advanced imaging APIs to support broad application usages. As the API utilizes the native graphics context, double-buffering is usually handled natively depending on the platform. Apps can query the hardware double-buffing capability in order to use any software double-buffering techniques.

There is also concern that existing SWT image handling which is done completely in Java is inefficient and needs to be revised to utilize the native capabilities provided on most platforms. This may require additional methods on the Image class.

The following shape drawing is provided:

- drawArc
- drawFocus
- drawLine
- drawOval
- drawPolygon
- drawPolyLine
- drawRectangle
- drawRoundRectangle

The following shape filling is provided:

- fillArc
- fillOval
- fill Polygon
- fill Rectangle
- fill RoundRectangle
- fillGradientRectangle

To clear background in system dependent manner. For example, implementation may show some skin or scheme graphics in the background ..

- drawSystemBackground

Clipping capacity:

- setClipping - limits drawing operations to the rectangle specified in argument

Drawing modes

[7] This feature may be added into Advanced package.

- setLineStyle – sets line drawing style to one of the following formats. SWT.LINE_SOLID, SWT.LINE_DASH, SWT.LINE_DOT, SWT.LINE_DASHDOT or SWT.LINE_DASHDOTDOT.
- setLineWidth – sets line drawing width to specified argument.
- setDrawMode –sets the drawing mode to the specified constant SWT.REPLACE, SWT.XOR, SWT. UNION, SWT.INTERSECT, SWT.EXCLUDE, SWT.COMPLEMENT. This puts the receiver in a drawing mode where the resulting color in the destination is determined by performing the specified operation on the source and destination values.

Text drawing capacity

- drawString – draws the argument text string to the specified location. This method is overloaded to also accept sub-strings and character arrays to enable more efficient operation.
- drawText – Support draw the argument text string with control chars:
  - DRAW_DELIMITER (draw multiple lines)
  - DRAW_TAB (expand tabs)
  - DRAW_TRANSPARENT (transparent background)
- Image and picture operations
  - copyArea – performs copy of image data from one area of a graphics context to another, or to an image object.
  - drawImage – draws an image to the graphics context.
  - ImageLoader – Load/save images from/to file or stream. It is mandatory for implementations to support loading of GIF, JPEG, and PNG formats. Loading of other formats like BMP may be supported on an implementation dependent basis. Saving of images in any format is supported on an implementation dependent basis. An exception is returned if loading or saving in the specified format is not supported.
- ImageData – Device independent representations of image
  - Image – Device dependent image

Table 1—Supported Image Data Types

Image Data Types	File Extensions
PNG (1.0): JTWI (MIDP) mandatory format.	.png
JPEG (supporting JPEG 2000 is unknown; but probably same licensing cost as current JPEG)	.jpg, .jpeg
GIF (87a and 89a): legacy support (backward compatible)	.gif

Shape/region transformation operations:

- transform –Transforms the specified image or region by multiplying each of its data points by a specified 3 x 3 matrix which represents an affine transformation.

1.4.14 Multiple Displays

One device can have multiple screens (inside and outside), or display accessories (local and remote). There are two requirements: display the same content to multiple screens; and display different content to different screens. This support is in the Expanded eSWT package.

1.4.15 Multimodal Notifications

Devices support various aural or tactile effects to notify or alert the end-user. Because these are device specific and often configured by the end-user, determination and invocation of such effects should be centralized in an Application Manager service, rather than delegated to each application.

1.4.16 Device-specific Extensions

Certain devices may provide device unique features. Optional device specific packages may provide access to these features from Java applications. All packages in this section must be treated as optional and device or implementation-specific.

1.4.16.1 Special Device Notification Events

An eSWT application must be notified when a part of the screen (or the full screen) is hidden or becomes invisible intentionally by other device accessories. The application is able to change its state and updates the UI accordingly.

The full-set of relevant event types are defined in each device dependent package.

1.4.16.2 Screen Orientation Change

There are two rotation types:

- LANDSCAPE
- PORTRAIT

Applications can freely change from one state to the other. Not all implementations may honor this call. Applications may query the current orientation. Querying the screen dimensions returns the width and height according to the current orientation. Implementations shall deliver orientation change events when appropriate. Newly created Shells and MessageBoxes display using the current orientation. That is, they do not change the orientation. It must be done by an application or app manager.

1.5 Supported JSRs

1.5.1 Mobile Media API (JSR 135)

1.5.1.1 Introduction

MMAPI is the multimedia API for applications. The API provides the capability of playing back audio (plus speech) and displaying video contents. The minimal requirement for target platforms for this JSR is CLDC range of devices. MMAPI defines a device display-independent mechanism, which currently has documented examples how to integrate it to AWT infrastructure, and LCDUI.

1.5.1.2 Implementation

MMAPI supports the same formats supported by JTWI (*JSR 185*).

There is a small mismatch between SWT and MMAPI because SWT requires widget constructors to supply the parent widget. This means that a dummy composite will be created by the MMAPI implementation to temporarily hold the eSWT control created from the GUIControl in USE_GUI_PRIMITIVE mode. Applications must add the control to a real parent composite by calling *setParent(Composite).* The MMAPI GUIControl.initDisplayMode must return an SWT Control for which calling isReparentable() returns always true. This requires minimal modification to the MMAPI documentation, so that the above is documented in the MMAPI GUIControl. This does not change any semantics of MMAPI specification but only gives rules how the SWT and MMAPI are used together. See also an example in Section 1.9, *eSWT and MMAPI example*.

1.5.2 Advanced Multimedia Supplements (JSR 234)

This JSR defines optional extensions to support other entertainment and multimedia accessories like camera, radio, and advanced audio controls. The JSR runs as a supplement in connection with MMAPI.

The API package for *JSR 234* is not finalized yet.

1.5.3 Scalable 2D Vector Graphics API (JSR 226)

1.5.3.1 Introduction

This JSR defines the API for showing and drawing scalable vector graphics and image files, including W3C Scalable Vector Graphics (SVG) format (SVG Tiny).

1.5.3.2 Implementation

This API defines a device display-independent mechanism, which supports AWT, LCDUI and eSWT. Since the display object class is not specified, any display-related object can be bound by *javax.microedition.m2d.ScalableGc* with *bindTarget* method. Thus, the *JSR 226* packages can run on eSWT.

1.5.4 Mobile 3D Media API (JSR 184)

1.5.4.1 Introduction

The Mobile 3D Graphics API is an optional package. This JSR defines an API for rendering three-dimensional (3D) graphics at interactive frame rates, including a scene graph structure and a corresponding file format for efficient management and deployment of 3D content. The main target platform of this optional API is J2ME/CLDC, used with profiles such as MIDP 1.0 or MIDP 2.0. However, the API can also be implemented on top of J2ME/CDC, or any Java platform in general

1.5.4.2 Implementation

This API defines a device display-independent mechanism. Since the display object class is not specified, any display-related object can be bound by *javax.microedition.m3d.Graphics3D* with *bindTarget* method. Thus, the *JSR 184* packages can run on eSWT.

1.6 Non-API Functionality

The UI API should support development of applications with look-and-feel similar to native applications. This means that the implementation of the UI (LCDUI, AWT, and SWT) has to comply with the native look-and-feel. The most obvious way to implement this requirement is to use the native implementation of the widget when ever possible.

The implementation of the API shall support localization and internationalization. This means that different character sets and writing directions are supported. eSWT automatically launches native Input Method Editors (IMEs) as appropriate for entry of compositional characters. Also, the widgets automatically utilize the proper device/locale settings for things like date formats.

1.6.1 Permission Control

Operational Management is allowed to set security policy regarding the level of function applications can achieve. This means, for example, the use of certain restricted UI styles like Always_On_Top, and controlling the application visibility. A policy might be desired that does not allow apps to bring themselves to the foreground, but requires explicit user action to do so. Thus, apps should only be able to directly adjust z-order of their children if they are in the foreground and should never be able to directly adjust their own (main window) visibility (z-order). This helps with the trusted dialog issue. Particular UI API associated with implementing permission restricted actions verify whether the calling code has the pre-requisite permissions. However, permissions are not settable on a per API basis.

1.6.2 Trusted Dialog

Applications can invoke a trusted dialogue for entering sensitive data. A trusted dialogue is a visual UI component that interacts with the end user in a secured way. No other applications can sniff or interrupt the input and output channels between the dialogue and the end-user. When a trusted dialogue becomes active and visible to the end user, the user can have a “hardware” way to verify the trustworthiness of the dialogue. That is, the implementation must provide an “escape” way to tell the user about the dialog owner and related security information like credentials on demand.

A potential mechanism for this feature is to provide an Application Manager API for displaying a trusted dialog. The App Manager can then request UI display a dialog and simultaneously update a security icon (that is in a screen area not drawable by an app) to show:

- Secure: if requesting app is foreground app
- Insecure: if requesting app is not foreground app

Clicking the icon or pressing appropriate buttons to activate it will request App Manager to display additional credential information.

1.7 Key Codes

Core eSWT provides the same set of keyboard key codes provided in *SWT*. However, mobile devices have a need for key codes to represent special keys that are typical for various embedded devices. Applications for embedded devices may desire to use these key codes without having any other need for Expanded eSWT or Mobile Extensions. Since key codes do not take much space, it makes sense to add these codes to SWT so that they are common across all platforms and inheritable by eSWT. Applications wishing to utilize softkeys may utilize the Mobile Extensions SoftKeyMenu widget to abstract the softkeys and make the application very portable or the application many monitor softkey event key codes directly. In the latter case, applications have to do more work to determine what softkeys are available and how function should be mapped.

Typical functions of the standard control keys:

- Navigation keys: navigation keys are ordinary keys or they can be implemented using different navigation devices like a roller (rocker) in different devices.
  - § Move focus one item in Combo, List, ListView, Table, and Tree.
  - § Move caret one line up/down or one char left/right in Text widget (single or multi-line)
  - § Move to previous/next page in MultiPageDialog.
  - § Adjust the value of scrollbar and slider
- Selection/Action key
  - § Fire the selection event in a focused item for Button (button, checkbox, radio, toggle), List, Table, and Tree.
  - § Select a menu item in menus.

All key codes will be explicitly defined in the eSWT specification.

1.8 New Features for SWT

New Styles:

- List Styles: SORT_ASCENDING, SORT_DESCENDING, QUERY, WRAP
- MessageBox Style: ICON_QUERY
- Colors: COLOR_WIDGET_HIGHLIGHTED_BACKGROUND, COLOR_WIDGET_HIGHLIGHTED_BORDER, COLOR_WIDGET_HIGHLIGHTED_FOREGROUND

New Methods:

- Display.getDefaultFont with parameter giving a logical context specifier
- GC.setDrawMode (New modes: SWT. UNION, SWT.INTERSECT, SWT.EXCLUDE, SWT.COMPLEMENT)
- GC.drawSystemBackground
- GC.queryDoubleBuffering
- ImageData.transform
- Display.queryKeyRepeatInitialDelay, queryKeyRepeatNextDelay[MR1]

New Events:

- Add Mobile oriented key codes to Core eSWT – should align with JSR 209 codes as feasible
- KeyEvent field or method on event for obtaining repeating key count information

New SWT Widgets:

- DateEditor
- ListView
- Web browser - already in some SWT platforms

1.9 eSWT and MMAPI example

```
Display display = new Display();

Shell shell = new Shell(display);

try {
        Player p = Manager.createPlayer("http://abc.mpg");
        p.realize();
        GUIControl guiControl;
        Control swtControl;
        if ((guiControl = (GUIControl) p.getControl("GUIControl")) != null) {
                swtControl =
                        guiControl.initDisplayMode(GUIControl.USE_GUI_PRIMITIVE,
                        “org.eclipse.swt.widget.Control”);
        }
        swtControl.setParent(shell);
} catch (MediaException pe) {
} catch (IOException ioe) {
}
```

End of Document

[MR1]Forgot control.setInputMode methods!