eSWT Requirements and High-Level Architecture

Abstract

There is a need for a standardized UI API fit for embedded devices having fewer resources and smaller screen sizes than a desktop computer. The goal of the eSWT project is to define a version of the Standard Widget Toolkit (SWT) to address this need. eSWT will be defined using a subset of SWT that is made suitable for embedded devices.

This document specifies the requirements for eSWT and outlines a proposed architecture relative to other APIs and toolkits that are used on embedded devices.

Document Information

Author: Mark Patel
Email: mark.patel@motorola.com
Phone: 847-523-0764

Change History

0.9	9/13/04	Date of initial draft
0.9.1	9/21/04	Revisions accepted, Some comments integrated
0.9.2	9/25/04	Incorporated changes from the eSWT kick-off meeting

1 Introduction

There is a need for a standardized UI API fit for embedded devices having fewer resources and smaller screen sizes than a desktop computer. The goal of the eSWT project is to define a version of the Standard Widget Toolkit (SWT) to address this need.

The ultimate goal is for eSWT to be a strict subset of SWT. However, using a strict subset of the current SWT will probably be unable to meet the requirements of eSWT. Therefore, this project will create a modified a subset of SWT, and any changes will be either pushed back into SWT or segregated as part of the Mobile Extensions package.

This document specifies the requirements for eSWT and outlines a proposed architecture relative to other APIs and toolkits that are used on embedded devices.

1.1.1 Scope

The scope of this document is limited to the requirements for the eSWT API and its envisioned architecture relative to other APIs relevant to embedded devices. Architectural details of a specific eSWT implementation are excluded from the scope.

2 High-Level Requirements

The high-level requirements are divided into three main areas. The first area addresses partitioning and the relationships between eSWT and other APIs and standards that are prevalent in the embedded device space. The second area addresses the unique characteristics of embedded device hardware and its implications for eSWT. The third area focuses on functionality that should be provided by an eSWT implementation and exposed via its API.

2.1 Architectural Requirements

2.1.1 Alignment with SWT

As a subset of SWT, eSWT should provide a set of APIs that are closely related to those of SWT. Alignment with SWT will promote developer acceptance and will further the re-use of existing SWT implementations, applications, and educational materials.

However, modifications to the SWT APIs may be necessary to make eSWT suitable for embedded devices. Such changes must either be applicable to SWT and pushed back, or they must be separable classes that will form the basis of one or more optional packages.

2.1.2 Partitioning

To promote flexibility and adaptability to a wider variety of embedded devices, eSWT will be partitioned as two basic components: *Core eSWT* and *Expanded eSWT*. Functionality which is required, but not necessarily appropriate for desktop SWT, shall be placed in a *Mobile Extensions* package.

Mobile Extensions is an optional package that provides user interface elements commonly found on mobile devices. It enables the creation of common applications and is modeled after the user interfaces of typical mobile devices such as phones and PDAs. This package must be either included or excluded in its entirety.

Expanded eSWT is an optional package that provides more sophisticated user interface elements that are commonly found on high-end mobile devices and PDAs. The Expanded eSWT package is intended to be used in addition to the Mobile Extensions package.

Core eSWT provides all remaining functionality that is not part of a specific user interface design or interaction model, including low-level graphics, events, and basic widget infrastructure. This portion of eSWT is device-agnostic and is applicable to a wider range of embedded devices including MP3 players, digital cameras, and automotive applications.

It must be possible to use Core eSWT with vendor-specific widget extensions to provide custom user interfaces (i.e. the use of the Mobile Extensions and Expanded eSWT packages is not required).

The packaging mechanism for each component will not be mandated by the eSWT specification, but it must be possible to use OSGi bundles for this purpose.

2.1.3 Portability

Applications which utilize eSWT shall be binary portable to any platform which implements the eSWT specification and is capable of running any other packages the application may be dependent on. A lower priority goal is that applications using eSWT Core shall also be binary portable to any platform which implements the desktop SWT specification; mobile extensions are not portable to the desktop.

Unlike the somewhat homogenous environment of the desktop, mobile devices vary dramatically in terms of screens and user input facilities. Therefore, eSWT will include APIs where appropriate to facilitate to aid in the portability of applications across multiple devices. Furthermore, eSWT APIs must be generic and must not be modeled around concepts or features that are unique to a particular device or platform.

2.1.4 Complementary APIs

A device that supports eSWT will likely included other Java APIs that provide complimentary functionality. Such functionality does not replicate that of eSWT, but rather provides additional capabilities that are not provided by eSWT and could be incorporated into an eSWT application in a useful way.

eSWT implementations are not required to include complimentary APIs, but their use from within an eSWT application must be possible if they are present. Therefore, eSWT APIs will be structured to permit the use of the following complimentary APIs:

- Mobile Media API (JSR 135)
- Advanced Multimedia Supplements (*JSR 234*). This JSR is a supplement to MMAPI (*JSR 135*), adding advanced audio processing and support for devices such as cameras and radios.
- Mobile 3D Graphics API for J2ME (*JSR 184*)
- Scalable 2D Vector Graphics API for J2ME (*JSR 226*)
- Java Speech API (*JSR 113*)
- Content Handler API *(JSR 211*

2.1.5 Analogous APIs

On a given device, other APIs may be present whose functionality is analogous to that of eSWT (i.e. other UI toolkits that provide similar functionality).

The use of analogous APIs from within an eSWT application is not supported, but shall not be explicitly prevented. That is, platform implementations may enable this behavior, but it is not encouraged and will most likely break binary compatibility with other platforms.

The presence of eSWT must not inhibit the availability of analogous APIs to other applications, nor may it reduce their capabilities or efficiency.

In the interests of code re-use, it is highly desirable to implement the functionality of other UI toolkits using eSWT. Due to the wide variety of widget sets, event mechanisms, etc., re-use across such APIs may be limited to low-level graphics functionality. Therefore, eSWT should provide low-level graphics functionality that is roughly equivalent to that of Java 2D and MIDP 2.0 in the following areas:

- Drawing primitives
- Text rendering
- Image decoding, rendering, and scaling
- Alpha blending

2.2 Device Requirements

Though embedded device hardware has evolved significantly, it still exhibits fundamental limitations due to size, weight, power, and cost constraints. Therefore, the following requirements must be met in order for eSWT to be considered suitable for embedded devices.

2.2.1 Screen Characteristics

Embedded device screens vary greatly in terms of their size, resolution, color depth, and orientation. In addition, some devices also employ multiple screens.

eSWT must allow an application to access multiple screens on a device if available. All eSWT API shall be applicable to any display, but their operation will be limited to the capabilities of the display. The app must be able to query such limitations.

eSWT must allow an application to request portrait (height >= width) or landscape (width > height) orientations for each accessible screen, although implementations are not required to honor these requests if different orientations are not supported. The application must be able to query the current orientation and will be notified of any orientation changes.

eSWT must allow an application to determine the color depth and dimensions of each available screen given its current orientation.

To-Do: Clarify with better use cases and requirements

2.2.2 User Input

Embedded devices support a variety of mechanisms for user input, though most use either key input, touch screen/pen input, or a combination of both. eSWT must allow the following input types:

- Key input modes to allow for representing characters not directly accessible via individual keys
- Control over predictive text input mechanisms
- Control over different FEP modes for international languages in order to handle different writing systems, etc.
- Control over the labeling of *softkeys* which are context sensitive – softkeys are common on mobile devices.

- The use of key 'press & hold' and multiple simultaneous key press is also prevalent on embedded devices to enable more complex functionality with a limited set of keys. eSWT must enable an application to detect such events and respond accordingly. The ability to poll individual key state is an additional goal.

The user must be able to traverse widgets (change focus) using a possibly limited set of focus traversal keys, i.e. only four arrow keys. If possible, applications should not have to be aware of the traversal limitations imposed by any particular hardware and must remain portable to platforms with or without secondary traversal keys, i.e. tab and back-tab.

2.2.3 Code Size

The size of an eSWT implementation will be dependent on a multitude of factors and will be largely dependent on the functionality of the underlying native toolkit. However, for [reference implementation native toolkit TBD], the target package sizes are:

- Core eSWT– 300k
- Expanded eSWT – 200k
- Mobile Extensions – 200k

(These are placeholder estimates subject to discussion)

2.2.4 Memory Usage

Memory is a scarce resource on embedded devices, and therefore eSWT APIs must be structured to permit efficient memory usage. Memory usage varies greatly depending on the size and design of the application, so it is not meaningful to define specific usage targets.

Images tend to consume a large portion of the memory used by most applications, and given the trend towards high-resolution screens and cameras, eSWT must permit efficient implementations for image decoding, rendering, and scaling.

- When implemented as a wrapper on top of a native toolkit, the eSWT APIs must never require native image data to be automatically duplicated in a Java array unless explicitly requested by the application.

2.2.5 Processing Power

Due to limited processing power, APIs must be structured to permit efficient implementations and usage by applications. The following conditions must be avoided wherever feasible:

- Creation of garbage objects (the API should promote the use of primitive types and the modification and re-use of previously-instantiated objects)
- Redundant pixel buffer copying and/or pixel format conversion
- Use of Java to manipulate large arrays such as images

2.2.6 Power Management

To achieve acceptable battery life on an embedded device, eSWT must be designed to support the use of power management strategies. For example, if eSWT provides animated images, it must be possible for applications to start and stop animation based on the current power state.

2.2.7 Robustness

Unlike a desktop environment, embedded devices are consumer products that are expected to have a higher level of robustness. Due to resource constraints, devices may use a single instance of a Virtual Machine to support multiple applications; therefore, undesirable behavior within a single application may result in failures in other areas of the device's user interface.

In a desktop setting, each application uses its own instance of the Virtual Machine, and in the event of a problem, the rest of the desktop UI remains usable and the application can be terminated without impacting the other applications. However, with a single Virtual Machine, a poorly written or malicious app can potentially lock the entire user interface or impact other applications.

To meet the overall robustness expectations for a mobile device, eSWT must be designed such that:

- It is always possible to terminate an application
- A single application cannot impact the operation of other applications, even if it is blocking the event thread
- There are no lingering side effects (e.g. memory usage) from a terminated application, regardless of how it was terminated

These requirements are similar to those currently being addressed by the Mobile Experts Group (MEG). The eSWT mechanisms used to meet robustness requirements should be modeled after those used by MEG.

3 Functional Requirements

The eSWT API and any compliant implementation must provide the following functionality. Unless otherwise stated, this functionality is provided in eSWT Core.

3.1 Image Decoding and Rendering

eSWT will provide the same image decoding and rendering functionality as SWT with the addition of the following:

- Image loading of the following formats: PNG (PNG is the most capable format as it can encode alpha data), GIF (including animated images), and JPEG
- API support for SVGTiny (implementation support is optional)
- Alpha blending (already supported in SWT)
- Affine transforms defined using a 3x3 matrix
- Access to meta-data contained within an image file such as the timestamp, text memo, location data, etc.
- Image operations should be comparable to native code in terms of performance and memory usage

3.2 Graphics Primitives

eSWT will provide the same graphics primitives as SWT with the addition of the following:

- Ability to set the alpha level of the foreground or background drawing color of a Graphics Context – Higher end visual effects are becoming more expected.
- Should support multiple Porter-Duff blending modes

- Reduce garbage object creation – memory conservation.

3.3 Text Rendering

eSWT will provide the same text rendering capabilities as SWT with the following modifications:

- Text-related functions in GC will be overloaded to work with substrings and character arrays in addition to entire String objects.

These changes will allow portions of strings or character arrays to be rendered without the overhead of creating garbage substring objects. Such capabilities were incorporated in MIDP to improve the overall efficiency of text rendering APIs.

3.4 Key Events

eSWT will provide the same events as SWT with the addition of the following capabilities:

- Ability to detect the number of continuous key repeat events
- Ability to query the press & hold delay period for the device
- Ability to poll the current state of each key - for gaming.

3.5 Text Entry

eSWT will provide the same text entry capabilities as SWT.

(need to investigate bi-di requirements and appropriate level of integration and engine exposure)

3.6 Core Widgets

Core eSWT will provide the basic widget infrastructure of SWT with the following additions:

- Transparent widgets will be supported (i.e. widgets may have a clear background, thereby allowing the widget behind them to be seen)
- The overall alpha level of each widget will be settable

Core eSWT includes a small set of basic, fundamental SWT widgets which are sufficiently abstract that their APIs and behavior are applicable to a wide range of user interaction models and device types. These widgets must be in the Core eSWT package:

- Widget
- Control
- Composite
- Canvas
- Shell
- Button
- Label
- List
- Text
- ProgressBar

Core eSWT provides the following layout functionality:

- A layout that enables composite space to be divided between two widgets such that each receives a certain percentage of the overall space.

The Composite class will also support an attribute for animated transitions. This attribute will indicate the type of animated transition to be used when showing or hiding the Shell. Implementations may interpret this attribute based on the availability and style of animated transitions on the device. The attribute value will convey meaning or intent rather than a specific animation effect.

It still needs to be determined if Composite is the appropriate class for this attribute; a better understanding of the expected usage patterns will be needed.

eSWT will provide support for modal and non-modal dialogs. To permit the use of eSWT as part of a complete device UI, it will also include support for system modal dialogs (as needed to implement security features such as PIN unlock codes).

As with SWT, the creation of custom widget classes will be supported. To enable the creation of custom widgets that match the look and feel of the device, a mechanism will be provided for querying skinning attributes. Such attributes may not be exhaustive, but will provide basic information regarding the appropriate colors, fonts, and styles that will allow custom widget to integrate well with the rest of the UI.

3.7 Compatability Widgets

Compatability widgets are widgets that are not necessarily required in the Core eSWT package but may also be useful for low capability devices. They could also go in the Expanded eSWT package. As a minimum, this set of widgets should align with those defined in MIDP.

- Combo
- ListView (not in SWT)
- Menu
- Date Editor (not in SWT)
- MessageBox
- Slider
- Media Picker Dialog

3.8 Advanced Widgets

The Expanded eSWT package will provide widgets with advanced capabilities. These capabilities are generally suited toward high-end devices:

- Table
- Tree
- Browser Widget
- Edit rich text (RTF, MMS, etc.) – must be able to function efficiently on limited input devices.

3.9 I18N and L10N

The API must allow applications to adapt to various languages and regions without requiring source code changes. In applications that are prepared to work on different languages and locales textual elements such as messages and widget labels should not be hard-coded into applications.

Instead, they are stored outside the source code and retrieved dynamically. A standardized API may provide access methods to localized data, for example, in bundles or local file systems if applicable, to enable the same binary to run in different regions and countries. Such an API is outside the scope of UI API. However, the UI API must be designed to enable this API to function properly.

3.10 Accessibility / multimodality:

Speech interfaces are becoming more realistic in near future hardware. The UI API shall be designed to allow voice driven commands.

The UI API must also not preclude text to speech reading for widgets containing text.

End of Document