Truffle Graphical Toolkit Customization Guide

Version 1.0 (Early Access)
June 15, 1999

Copyright

ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

Truffle Graphical Toolkit Customization Guide

Version 1.0 (DRAFT)
May 25, 1999

Also available in PDF file [175KB]

This customization guide describes how to modify and customize the Truffle Graphical Toolkit. It focuses on programming and architecture issues.

Table of Contents

- I18N Support

A Tour of `Button`

- Creation of a `Button` Component
- Event Delivery to a `Button` Component
- Models, Views & Controllers
- Optimizations

Truffle Graphical Toolkit Customization Guide

Preface

This customization guide describes how to modify and customize the Truffle Graphical Toolkit. It focuses on programming and architecture issues.

Audience

The primary reader is a Java software developer who is responsible for developing a look & feel based on the Truffle graphical toolkit. The reader should be familiar with Java programming in general and AWT terminology in particular.

Additional Reading

The following documents provide important related information:

- The PersonalJava Product Page provides the latest information about PersonalJava technology.
- The PersonalJava Application Environment (PJAE) Specification describes the API relationship between the PJAE and JDK 1.1 software.
- The **Touchable Look & Feel Specification** describes a flexible look & feel design intended for consumer devices based on a touchscreen display.
- The **PersonalJava Porting Guide** describes how to port the PersonalJava application environment (PJAE) to a target RTOS.
- The Java Programming Language Specification (Addison-Wesley, 1996) is the standard reference for the Java programming language.
- The Java Virtual Machine Specification (Addison-Wesley, 1996) is the standard reference for the Java virtual machine.

- The JDK 1.1.x API reference documentation describes the API of the Java class library.

Technical Support

For technical comments or questions, please send e-mail to: `personaljava-comments@java.sun.com`.

Truffle Customization Guide

Introduction

The Truffle graphical toolkit can be used to develop different look & feel designs for a variety of consumer devices. The Truffle graphical toolkit provides designers with a flexible framework that has most of the functionality necessary to implement a custom look & feel design. This allows consumer device designers to concentrate their efforts on the small amount of code that represents the look & feel itself.

To better understand the Truffle graphical toolkit, you need to see it from different perspectives. The most important is its relationship to the `java.awt` API.

The Abstract Window Tookit (AWT) is a toolkit for building graphical user-interfaces (GUIs) for Java software. The AWT divides the task of supplying GUI services between high-level GUI classes and platform-level peer components supplied by the underlying platform. Java software interacts only with the high-level GUI classes while the AWT maps these high-level classes to peer components. This architecture allows Java software with GUIs to run on different target platforms without modification.

The Truffle graphical toolkit provides a flexible implementation of the platform-level peer components required by the AWT. To achieve this goal while meeting the needs of different platforms, the Truffle graphical toolkit includes a framework for delivering alternate look & feel designs for different Java application environments.

As an example of a specific look & feel design, the Truffle graphical toolkit includes the Touchable look & feel design for the PJAE operating on touchscreen-based consumer devices. Touchable can be either modified or replaced to support the product identity or product design needs of specific licensees.

The Truffle graphical toolkit does not add any new APIs for application developers to learn. It provides a set of peer components that Java software can use through the standard `java.awt` API. In addition, the Truffle graphical toolkit includes

a special-purpose window and graphics system for platforms that lack a native window and graphics systems.

Definitions

The following list defines some of the important terms in the Truffle vocabulary. Some of these definitions describe how concepts differ between the Truffle context and the desktop context.

Consumer Device
: For the purposes of this discussion, a consumer device is a network-connectable device with both a small graphics display and a few simple input devices. A touchscreen would represent a combination of these two requirements. The computing resources (memory, CPU, etc.) should be sufficient to support a Java application environment like the PJAE.

PersonalJava Application Environment (PJAE) API
: The PJAE API is derived from the JDK API. For example, version 1.1 of the PJAE API is based on the JDK 1.1.6 API. Some components in the JDK 1.1.6 API are **optional** in the PJAE API. See the PersonalJava Application Environment Specification for more information on how the PJAE API differs from the JDK API.

  In particular, the PersonalJava Application Environment Specification allows an implementation of the PJAE to support a subset of the `java.awt` API.

Abstract Window Toolkit (AWT)
: The AWT is an API for building graphical user-interfaces (GUI) for Java software. It includes high-level GUI classes that are mapped to platform-level peer components through the `java.awt.peer` interface. For example, `java.awt.Button` is a high-level class that Java software can use to produce a generic button.

Peer Set
: Each implementation of a Java application environment must supply a group of user-interface components, called a **peer set**, that support the high-level classes of the AWT. Desktop versions of the JDK usually provide peer set implementations through a small amount of Java wrapper code and a set of interface functions based on a platform-specific GUI library. For example, a JDK implementation for a desktop system like the Solaris

Operating Environment supplies a font peer component that is mapped to the high-level `java.awt.Font` class. The Solaris implementation of the JDK uses the Motif library to supply a peer set implementation. The Truffle graphical toolkit supplies a peer set written entirely in the Java programming language.

Truffle Graphical Toolkit

The Truffle graphical toolkit provides both a peer set implementation and a framework for supplying alternate look & feel designs for Java application environments. The Truffle graphical toolkit is written in the Java programming language and includes a special-purpose window and graphics system for platforms that lack a native window system and graphics system.

Look & Feel Design

A look & feel design represents the visual appearance and behavior of a GUI component set. The "look" is based on the graphical design characteristics shared within a GUI component set. For example, a GUI component set might have similar color and border decoration schemes. The "feel" is based on the input mechanisms that a GUI component set provides for a user to interact with. For example, a GUI component set might be based on taking advantage of a touchscreen and provide certain kinds of graphical feedback that is appropriate for a consumer. Look & feel designs vary according to the needs of the target user and the underlying platform. For example, the look & feel designs in Java Foundation Classes (JFC) reflect the needs of enterprise users while Touchable is designed for consumers using touchscreen-based consumer devices like screenphones.

Touchable Look & Feel

The Touchable look & feel design is appropriate for touchscreen-based consumer devices like screenphones. These consumer devices require GUI components that are scaled to use finger or stylus-based input mechanisms.

Java Foundation Classes (JFC)

JFC is a set of GUI components that go beyond the basic set of GUI components provided by the AWT. For example, JFC includes a GUI component called `JTree` for displaying complex hierarchical data sets. The GUI components in JFC are intended for use by enterprise applications that benefit from large displays. In addition to providing new GUI components JFC includes three

different "pluggable" look & feel designs: Motif, Microsoft Windows 95/NT and Metal, the cross-platform Java look & feel design. The look & feel designs in JFC only affect the GUI components in JFC and do not affect the GUI components in the AWT.

Goals

The Truffle graphical toolkit is designed to meet the following goals:

- Easily customized look & feel designs
- High portability
- Low memory footprint
- I18N support

Source Code Organization

The Truffle graphical toolkit source code is kept in two main directories in the PJAE source tree:

- `src/reference` contains the reference source code to the Touchable look & feel design. The Touchable source code can be modified to develop new look & feel designs.
- `src/share/classes/sun` contains the source code for OTK (`sun.awt.otk`) and the graphics system and the window system. The source code in `src/share/classes/sun` cannot be modified to develop new look & feel designs.

Truffle Graphical Toolkit Customization Guide

Consumer Devices and Human Interface Design

The amount of computing resources that can be delivered in consumer devices has increased substantially in recent years to the point where these products represent a software platform comparable to desktop systems. This has led to efforts to modify desktop technology for use with consumer devices. In some cases borrowing technology from desktop systems has been successful. But the use of desktop-centric human interface design techniques in consumer devices has not been successful.

Consumer devices are tightly integrated and reflect a careful balance between several competing design criteria. Over time the consumer device market will see a shift where much of the value contained in a given product will be transferred from solid-state into software. As this shift occurs, designers will need a GUI toolkit that is flexible enough to handle a variety of different product design scenarios.

Consumer devices need different human interface designs for several reasons. These devices are smaller and their input devices are simpler. Consumers may not have any computer experience, so human interface metaphors like drag and drop that are borrowed from desktop environments may be inappropriate. Moreover, consumers -- even those with computer experience -- have very different **expectations** when they interact with a consumer device. Psychologically, they expect these products to be simple and predictable and they have a low tolerance for learning how to use them.

The Truffle graphical toolkit represents both an engineering achievement and an advance in human interface design. The engineering effort shows in the design and implementation of the Truffle graphical toolkit itself. The Touchable look & feel is the result of several years of human interface research into consumer behavior. Many lessons learned from developing Touchable will be used to develop additional look & feel designs for different kinds of consumer devices.

Platform Characteristics

Characteristic	Touchable (Screen Phone)	Couchable (Set-Top Box)	Metal (Desktop)
applications	telephony, voice mail, address book, email, web browsing	TV, EPG, web browsing, email	word processing, presentations, web browsing, email, productivity applications, vertical applications
input device	finger, stylus	remote control	mouse
mouse support	none	none (although some remote controls have track balls)	yes
keyboard support	virtual or physical keyboard	virtual or physical keyboard	physical keyboard
viewing distance	1-2 feet	10-15 feet	1-2 feet
display size	6 inches	13 inches to wall-size	13-28 inches
screen resolution	1/4 VGA (320x240) to full VGA (640x480)	broadcast television	640x480 to 1800x1440
screen colors	2, 4 or 8-bit displays; both black & white and color	broadcast television	16-bit or 24-bit
pixel density	~102 dpi	N/A	~72 dpi
multiple screens	no	no	yes

audio input	telephone handset, microphone	none	microphone
audio output	telephone handset, perhaps speaker	TV speakers up to full surround-sound	computer speakers to high-end speakers
data bandwidth	28.8Kbps to ISDN	28.8Kbps to cable modem throughput	28.8Kbps to Ethernet
printer connection	optional	optional	yes

Multiple Look & Feel Designs

Why should the PJAE support multiple look & feel designs, particularly for consumer devices? The short answer is that different products have different needs and the level of integration in consumer devices challenges platform developers to provide unique and elegant solutions to human interface design problems.

Multiple look & feel designs are needed to support both the product design needs of consumer devices and the product identity needs of licensees. If the API for the GUI toolkit (`java.awt`) must remain unchanged so that applications can run on different platforms, and if consumer device manufacturers must develop widely differentiated products, then the PJAE must be able to supply different look & feel designs. The Truffle graphical toolkit provides this flexibility so that licensees can deliver product-specific look & feel designs in a small memory footprint.

The need for simplicity in human interface design does not imply the need for a single look & feel. Successful consumer devices must be simple to use so that consumers can intuitively understand how to use them. Nevertheless, different product designs need look & feel designs that reflect the purpose of the consumer device.

Comparing the Truffle Graphical Toolkit and JFC

It is useful to compare the Truffle Graphical Toolkit with JFC because they share many design goals. Both the Truffle Graphical Toolkit and JFC are written in the Java programming

language and both support multiple look & feel designs. JFC's **pluggable** look & feel architecture allows an application to select a look & feel design at **runtime**. This level of flexibility in JFC may add too much complexity or require too many resources for consumer devices operating in a small amount of memory. In contrast, the Truffle Graphical Toolkit provides a **replaceable** look & feel architecture where a single look & feel design is selected at **startup time**.

Although JFC contains a rich set of GUI components that **extends** GUI functionality beyond what is provided by the AWT, the Truffle Graphical Toolkit provides only core AWT functionality. This has two benefits: it keeps the Truffle Graphical Toolkit implementation compact and it removes the need for application developers to learn a new API to write software for Truffle-based platforms.

The PersonalJava Application Environment Specification places a further constraint on the peer components supplied through the Truffle Graphical Toolkit by defining a minimum subset of the AWT components. These components were selected for their appropriateness for consumer device applications. For example, `java.awt.Frame` is a **modified** GUI component in a PJAE implementation.

The optional and modified peer components are described in the table below.

AWT Peers

Peer	PJAE Support Level		
	Required	**Optional**	**Modified**
`Button`	x		
`Canvas`	x		
`CheckboxMenuItem`		x	
`Checkbox`	x		
`Choice`	x		
`Component`			x
`Container`	x		
`Dialog`			x
`FileDialog`		x	

Font	x		
Frame			x
Label	x		
Lightweight	x		
List	x		
MenuBar		x	
MenuComponent	x		
MenuItem	x		
Menu		x	
Panel	x		
PopupMenu			x
ScrollPane	x		
Scrollbar		x	
TextArea	x		
TextComponent	x		
TextField	x		
Window			x

Touchable Look & Feel

The Touchable look & feel design supports touchscreen-based consumer devices like screenphones. A consumer using a screenphone expects it to behave much like a conventional phone, so that most interaction is driven by pressing a finger on a button to select a choice. The figure below shows a dialog for an address book application. While this application may run without modification on another Java application environment like the JDK, on the PJAE using the Touchable look & feel it has a distinctive consumer-friendly appearance. Buttons are wide and rounded while vertical scrollbars have been replaced with button pairs to simplify finger-based operation.

Touchable Example [Click on image for larger view]

Scalability of Look & Feel Designs

Look & feel designs are usually crafted for a specific resolution. For example, the first version of Touchable is designed for VGA resolution (640x480).

There are two approaches to developing versions of Touchable for other resolutions. The first is to simply scale each graphical element in the look & feel design by a constant scale factor. The drawback to this approach is that it often causes roundoff errors. For example, if a certain graphical element is a single pixel wide, then it may disappear or remain the same size. In any event, the relative sizes of the different graphical elements might shift due to scaling.

The second approach to scaling a look & feel design is to create a derivative look & feel design based on the original. This is not as difficult as creating a new look & feel design because it is based on the graphical design principles of the original look & feel design and because it is limited only to the graphical elements. In a way, it is like designing a screen font for a different resolution based on an existing screen font design.

The Truffle Graphical Toolkit can support implementations that need multiple look & feel designs, for example to support multiple display resolutions. But the Truffle Graphical Toolkit can only support a single look & feel design at runtime. The classes that represent a specific look and feel design are not memory intensive.

Look & Feel Design Template

The process of developing a look & feel design for the Truffle Graphical Toolkit involves the collaboration of human-interface and engineering expertise. The best way to approach this task is to use a template to mock up the design of each GUI component in `java.awt`.

```
API-level
  Button
  Frame
  ...
Internal
  VirtualKeyboard
  ...
```

Describe the visual appearance for each component, and the related appearances for the component in its various states (e.g. selected, unselected). This could include color schemes and scale factors, as is the case in Touchable. This set of descriptions roughly matches the `look` requirements that are implemented by the `View` classes.

Next, create a similar table that describes the user-interaction characteristics, the `feel` for each GUI component. For example, will the user hear an audio feedback when a button is pressed? If so, then add the `Controller` for that purpose.

This second table should provide a starting point for the `Controller` strategy for the look & feel implementation. Since the Truffle Graphical Toolkit already contains a variety of prebuilt `Controllers`, this is often reduced to a matter of selecting from the available `Controllers` in `sun.awt.otk`.

Truffle Graphical Toolkit Customization Guide

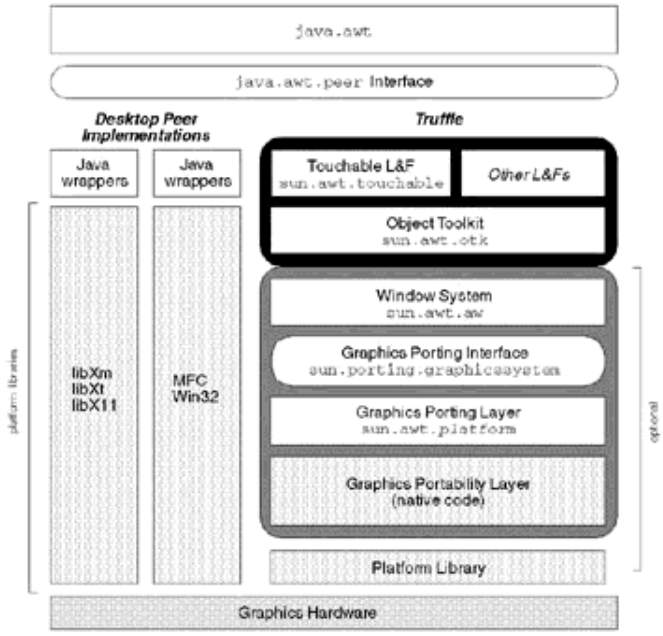
Architecture

The Truffle graphical toolkit was designed to meet several requirements. It supplies an AWT peer set written in the Java programming language. Furthermore, it provides a framework for supplying alternate look & feel designs that are targeted at different kinds of platforms, including consumer devices and desktop systems. This section describes how the Truffle graphical toolkit's architecture supports these goals.

The figure below illustrates the major components of the Truffle graphical toolkit's architecture and compares them with desktop implementations of the JDK. The Truffle graphical toolkit fulfills the goal of providing a cross-platform GUI toolkit by providing a set of peer components and the window and graphics system necessary to support them. The Truffle graphical toolkit plays the same role in a Java application environment like the PJAE that native peer sets perform in desktop implementations of the JDK. In fact, the Truffle graphical toolkit can be used to replace these native peer sets.

It is much easier to understand the Truffle Graphical Toolkit's purpose and design by thinking of the Truffle Graphical Toolkit in its relationship with the AWT rather than seeing it as sitting on top of a window and graphics system. The reason lies in the Truffle Graphical Toolkit's goal of providing **just enough** GUI services to meet the requirements of the AWT. In other words, the Truffle Graphical Toolkit is not a general-purpose GUI toolkit with a large number of widgets like Motif and JFC. It is specifically designed to support the API requirements of the AWT and the human-interface needs of consumer devices.

Truffle Graphical Toolkit Architecture [Click on image for larger view]



Object Toolkit

The Truffle graphical toolkit contains a toolkit-within-a-toolkit named **Object Toolkit** (OTK) that can be used by licensees to construct new look & feel designs. For example, the Touchable look & feel design is represented by a set of classes in `sun.awt.touchable` that use OTK classes to implement AWT peer components that appear and behave in a way that is suitable for touchscreen-based consumer devices like screenphones. Another look & feel design could be an implementation of Metal, the Java look & feel design used in JFC.

The Truffle Graphical Toolkit uses the Model-View-Controller (MVC) architecture to organize the peer set implementation into groups of classes that perform different roles:

Model	data state
View	appearance
Controller	synchronization or behavior

MVC is an example of a design pattern. See Design Patterns: Elements of Reusable Object-Oriented Software. for a more thorough introduction to design patterns.

MVC allows effective use of code sharing that leverages functionality between components and thus reduces code size.

From a licensee's perspective, OTK reduces the software development task for creating a platform-specific look & feel design by focusing effort on the appearance and behavior of the peer components and avoiding the implementation of the underlying support systems.

Internal Toolkit Components

The Truffle Graphical Toolkit allows implementations to create two types of components: peer components that correspond to the `java.awt.peer` interfaces and internal toolkit components. This second category contains components without external application-level APIs. Instead these internal components are used by the toolkit itself to provide platform-level services (e.g. `Taskbar`) or alternate input techniques (e.g. `VirtualKeyboard`).

Look & Feel Implementation Strategies

There are a number of approaches to take in implementing a look & feel design with the Truffle Graphical Toolkit. The sections below describe the major alternatives.

Configure Touchable

The term *configuration* is used to indicate ways in which Touchable can be changed without many source code changes. Touchable was designed to be configurable in two ways:

- **Scaling Factors.** By default, Touchable supports a display size of 640x480. This can easily be switched by using a scaling factor defined in `DrawUtils`. The PersonalJava emulation environment (PJEE) also has some runtime properties (`sun.graphicssystem.height` and `sun.graphicssystem.width`) that are useful in testing different scaling factors.

- **Color Schemes.** Touchable has a color palette which is based on a subset of the symbolic color names in `java.awt.SystemColor`. Each palette entry identifies colors used for various component elements. For example, user text is displayed with the color defined by the symbolic color `java.awt.SystemColor.TEXT_TEXT`. This symbolic name is mapped to a named color value defined in a palette.

  The default color palette used by Touchable is based on the color orange. Touchable can be configured to use a different predefined color palette (by using the PJEE

runtime property `sun.awt.palette`) or defining a new color palette in `reference/classes/lib/touchable.palettes`.

Modify Touchable

Modifying Touchable is the most common approach to developing a new look & feel design. This involves a moderate amount of source code changes. Here are some possible scenarios:

- **Develop a new feel.** Add or subtract `Controllers` attached to the `View` classes. This modification is easy to do and represents a modified feel.

- **Develop a new look.** The two approaches to take with this task are organized around where the implementation for the painting code resides:

    - Directly extend a `View` class. This approach is easier to understand at first, but it fails to take advantage of the benefits of factoring shared drawing code. Still, some of the more complex components in Touchable use this approach.

    - Share design elements in `DrawUtils`. This approach steamlines the implementation of components by coordinating their design features and reducing their implementation footprint.

Build an OTK-based Toolkit

Developing an entirely new toolkit based on OTK is a significant amount of work. This approach usually indicates that some architectural goal of the new look & feel is not being met by Touchable and it is better to start from scratch. Even so, it is still best to have a solid understanding of the Touchable implementation before starting on a new toolkit.

OTK contains the minimal common subset of toolkit functionality. Therefore, a full Truffle-based toolkit requires substantially more than the functionality that `sun.awt.otk` provides. For example, the `ColorPolicy`, `DrawUtils` and `TouchableToolkit` classes are all outside of `sun.awt.otk`.

Replaceable Look & Feel Designs

The Truffle Graphical Toolkit can support a variety of look &

feel designs. A specific look & feel design is attached to a given Java application environment at startup time. So only a single look & feel design is available for that platform during a user session. This is called a **replaceable** look & feel architecture to distinguish it from JFC's **pluggable** look & feel architecture which allows each application to select a look & feel design at runtime.

Look & Feel Design	Target System
Touchable	touchscreen-based screen phones
Couchable	remote control-based set-top boxes
Metal	network computers & PCs

In addition, licensees can use OTK to build product-specific look & feel designs that support their product identity or product design needs. This can be done by modifying an existing design like Touchable with a different color scheme or by developing a new look & feel design from scratch.

Truffle Graphical Toolkit Software and Different Java Technology-based Application Environments

The Truffle Graphical Toolkit is part of the PJES but it can also be used with other Java application environments like the EmbeddedJava application environment, JavaOS and JDK-based systems.

Window and Graphics System

The Truffle Graphical Toolkit includes a special-purpose window and graphics system written mostly in the Java programming language. The platform-dependent portion of the graphics system is usually based on a platform's graphics library.

I18N Support

The Truffle Graphical Toolkit provides I18N services required by the AWT. In particular, The Truffle Graphical Toolkit includes an input method framework written in the Java programming language and virtual keyboard support. Native input methods can be integrated with this input method framework with an adapter class. Beyond portability, the main advantages of the Truffle Graphical Toolkit input method framework is its integration with both virtual keyboards and lightweight components.

Truffle Graphical Toolkit Customization Guide

A Tour of `Button`

The different GUI component classes like `Button` and `List` that make up the `java.awt` API vary in complexity. To reduce memory footprint and streamline their implementation, the Truffle graphical toolkit shares many features between the various GUI component implementations. It's useful to understand these shared features as a starting point for learning about the architecture of the Truffle graphical toolkit. Therefore this chapter is organized around a fairly simple component class, `Button`, to introduce the basics of the Truffle Graphical Toolkit's architecture. `Button` is the simplest GUI component in `java.awt` that can handle input events. But many other component classes in the Truffle Graphical Toolkit are based on the same architectural features that `Button` uses.

Because the classes in the the Truffle graphical toolkit can at first seem complex, it helps to narrow your focus on a small subset and discover how they interact to perform simple tasks like component object creation and event handling. The sections below describe how the Truffle graphical toolkit operates in some basic scenarios involving `Button`.

Creation of a `Button` Component

Figures 1 and 2 illustrate the call path that occurs during the creation of a `java.awt.Button` component in a Java application.

The steps below describe how the process of creating a `Button` component object uses classes in the `java.awt`, `sun.awt.otk` and `sun.awt.touchable` packages. Each of these packages performs a different role:

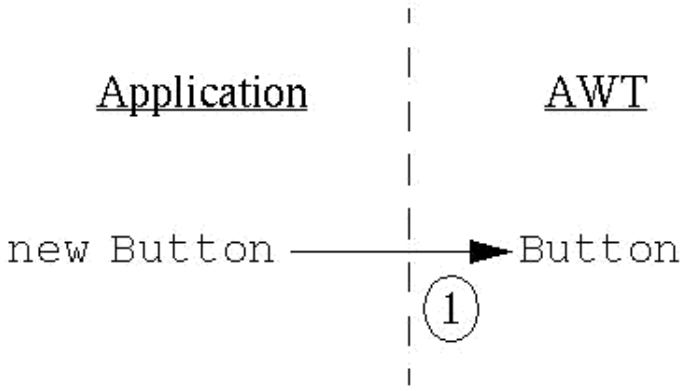
- `java.awt` provides an API for applications.
- `sun.awt.otk` provides a set of reusable classes for implementing look & feel designs. The major building blocks are:
  - `Model` classes represent the data state of components.
  - `View` classes represent the graphical appearance or look of components.
  - `Controller` classes manage the effects of events, in other words their feel.

  Note: While the `sun.awt.otk` source code is available for reference purposes, it cannot be modified for the purposes of building a new look & feel design.
- `sun.awt.touchable` contains a set of classes that represent the Touchable look & feel design. The most important among these are the various `Views`, `TouchableToolkit`, `Palette` and `DrawUtils`.

Note: The Touchable classes can be modified or configured to build new look & feel designs.

Here is an overview of the process that the Java runtime goes through in using the Truffle graphical toolkit to build a `Button` component object:

Figure 1



1. **Create a `Button` object.** First, a Java application class calls the `Button` constructor:

   ```
   Button b = new Button("My Label");
   ```

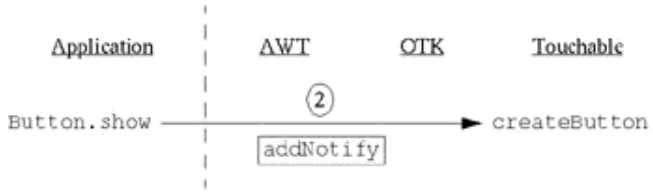
   The `Button` constructor sets the button label and then calls the `Component` superclass constructor which performs the rest of the component initialization.

2. **Create a `Button` peer component.** The processes of creating a `Button` component object and creating a `Button` peer object are separate. The constructors for `Button` and `Component` don't immediately create a `Button` peer component. Instead, they save the `Button` state information in the `Button` object for later use by the `Toolkit`. The actual creation of the `Button` peer component is deferred until the `Button.addNotify` method is called by `Component.setVisible`.

   A component object like `Button` actually creates a number of objects:
   - The Java application keeps a reference to a `Component` object while the object is in use. When the `Component` object is no longer needed, it is made available for garbage collection.
   - The AWT toolkit keeps a corresponding set of classes that implement the `java.awt.peer` interface. These peer classes represent the inner workings of a GUI component. Implementations of `java.awt` for desktop platforms often use native GUI toolkit widgets to implement these peer interfaces. In contrast, the Truffle graphical toolkit provides a Java implementation of these peer classes.
   - The top-level classes that implements the `java.awt.peer` interface are the `Model` classes. Each `Model` class has a `View` and zero or more `Controller` classes

associated with it.

Figure 2 [Click on image for larger view]



Eventually, `addNotify` creates the `Button` peer:

```
ButtonPeer peer = getToolkit().createButton(this);
```

For each `Button` peer object, `TouchableToolkit.createButton` creates a `LabelledComponentModel` object, a `ButtonView` object, and any number of `Controller` objects associated with the `ButtonView` object.

```
protected java.awt.peer.ButtonPeer createButton(java.awt.Button target) {
  LabelledComponentModel model =
    (LabelledComponentModel) makeLabelledComponentModel(target);
  View view = new ButtonView(model, this, getParentView(target),
    Palette.BUTTON_COLORS);
  ControllerManager mgr = (ControllerManager)view;

  mgr.addController(mouseSound);
  mgr.addController(ArmableMouseController.consumingController);

  model.setDefaultView(view);
  ((BasicView)view).acceptFocus();

  Rectangle b = target.getBounds();
  model.setBounds(b.x, b.y, b.width, b.height);

  peerMap.put(target, model);
  model.initComplete();

  return (java.awt.peer.ButtonPeer) model;
}
```

3. **Create a `LabelledComponentModel` object.** `Model` objects store a component's application state data. The most important `Button`-specific state data in `LabelledComponentModel` is `label`; the rest of the state data (e.g. the foreground color) is inherited from `BasicView`.

   ```
   LabelledComponentModel model =
     (LabelledComponentModel) makeLabelledComponentModel(target);
   ```

   The `LabelledComponentModel` class is shared by `ButtonPeer` and `LabelPeer`. The main difference is that `LabelPeer` doesn't have any controllers attached to it while `ButtonPeer` has two and `LabelPeer` does not capture the input focus.

4. **Create a `ButtonView` object.** The `createButton` method then creates a `ButtonView`

object which manages the graphical appearance of the button in both the default (unselected) and triggered (selected) states.

```
View view = new ButtonView(model, this, getParentView(target),
  Palette.BUTTON_COLORS);
```

The actual drawing code for `ButtonView` is in the `DrawUtils.drawButton` method (see `DrawUtils`).

5. **Add `Controller` objects to the `ButtonView` object.** `Controller` objects handle events that have been dispatched to a `View` object:

```
mgr.addController(mouseSound);
mgr.addController(ArmableMouseController.consumingController);
```

`Controllers` implement specific behavior that can be reused in different components. For example, `mouseSound` can be used to provide audio feedback to a number of different components.

A component peer object can have any number of `Controller` objects associated with it. These `Controller` objects form a controller chain, which is described in Event Delivery to a `Button` Component.

Remember that peer component creation is a staged process. The application class creates a component object and eventually the toolkit creates a peer object. The `Model` class implements the `java.awt.peer` interface and has a `View` and zero or more `Controllers`.

`DrawUtils`

Many of the various `View` classes like `ButtonView` keep their drawing code in a single utility class called `DrawUtils`. This allows the drawing code for many of the components to easily share functionality. The most important method used by `ButtonView` is `drawButton`. The `getMinimumSize`, `getPreferredSize` and `boundsChanged` methods in `ComponentView` are also important. This is the starting point for many customizations for a given look & feel design.

`DrawUtils` also contains code for the following:

- Shared painting code for simple components and text.
- Defaults for a look (heights, inter-component spacing, etc.)
- Scaling factors
- Initialization of the default palette.

`Palette`

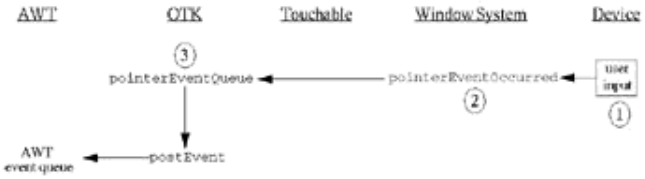
The `Palette` class defines the symbolic names for the palette entries used by Touchable.

Event Delivery to a `Button` Component

Many GUI components in `java.awt` can handle events that allow users to interact with them to change their state. These events are initiated by the window system as raw input events and then forwarded to specific components by `ObjectToolkit`. Figures 3 and 4 illustrate the call path that occurs during the delivery of an event from the `sun.awt.aw` window system through `ObjectToolkit` to a `Button` component.

Here is an overview of the event delivery process for a `Button` component. More complex components like `List` use a similar approach.

Figure 3 [Click on image for larger view]



1. **The user interacts with an input device.** For example, when a user presses a location point on an interactive LCD display, the display device driver and RTOS cooperate to send a raw input data to the Truffle window system.

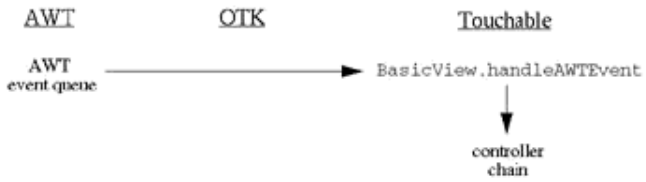
2. **The `sun.awt.aw` window system forwards the event to `ObjectToolkit`.** When `ObjectToolkit` is instantiated, it registers an event handler with the `sun.awt.aw` window system. This provides a callback mechanism that allows the window system to send events to `ObjectToolkit`. The main method for doing this is `WindowSystem.pointerEventOccurred`. At this stage, the event is unformatted and not yet associated with a specific GUI component. The signature for the `pointerEventOccurred` shows that the event only contains time, location and device data:

   ```
   void pointerEventOccurred(long when, int x, int y, int ID, int number);
   ```

3. **`ObjectToolkit` sends the event to a specific `View`.** When a component is instantiated, a `View` object is created along with an associated `Window`. The `View` object stores a reference to the `Window` object in its user data. Since the `Window` object also keeps track of its parent `View` object, this technique allows the two objects to keep track of each other.

   When `ObjectToolkit` receives notification of a raw pointer event from the window system, it first determines the window in which the event occurred and uses the reference to that window to look up its associated `View`. `ObjectToolkit` then translates the event into a `java.awt.MouseEvent` and delivers the event to the associated `View` object by calling the `postEvent` method. This method posts the event to the system event queue which eventually delivers it to the `View` object through the `BasicView.handleAWTEvent` method.

   Figure 4 [Click on image for larger view]



4. **The `View` object handles the event.** When the `BasicView.handleAWTEvent` is called by the system event queue, it starts a `View`-specific event handling mechanism. The first step is to filter the event through the `EventDispatcherPolicy` class (see `Gizmos` and the `EventDispatcherPolicy` Class) which eventually sends the event to its `Controller` chain.

5. **The event is processed by the `Controller` chain.** Since even a simple component like `Button` can have more than one `Controller` associated with it, these `Controllers` form a chain. Events are handled by `Controllers` in the order that the `Controllers` were added to the `View`. Each `Controller` can ignore or handle the event. When handling events, `Controllers` can choose to consume the event completely or pass it on to the next `Controller` in the chain. `Controllers` call methods in the `View` and `Model` classes to implement the expected "feel" of the particular event. The exact methods that can be called are enforced by a shared interface that serves as a contract. In this example, the interface is `Armable` and provides methods to implement the behaviour of clicking a button.

6. **The event is processed by the Java application.** Eventually, the event may be forwarded to an application-level class for event processing via the `Listener` interface.

`Gizmos` and the `EventDispatcherPolicy` Class

Simple components like `Button` have `View` classes that are based on `Window`. But more complex component classes like `List` need a mechanism for directing events to subregions of the component. For example, when a user selects an item in a `List` component, the component needs a dispatch mechanism that allows the appropriate subregion to handle the selection event.

The `Gizmo` and `EventDispatcherPolicy` classes provide a general mechanism for subregion event delivery. The `Gizmo` class is like the `View` class except that instead of having an associated `Window` class it has a subregion of a `Window`. The `EventDispatcherPolicy` class allows fine-grain control of event dispatching from the top-level `View` to the appropriate `Gizmo`.

Simple components like `Button` have no need for `Gizmos`, and their `EventDispatcherPolicy` object forwards events directly to the first `Controller` associated with the `View`.

More complex components are composed of several `Gizmos` that are each capable of event handling. For example, in the case of a `List` component, each item in the list is represented by a `Gizmo`. The `ListView` is associated with a single `Window` but potentially several `Gizmos`, each of which represents a subregion of the component. Components like `List` can extend `EventDispatcherPolicy` to implement complex event dispatching techniques that organize how events are mapped to arbitrarily complex sets of `Gizmos`.

Models, Views & Controllers

The following sections describe some of the major classes that are used to implement `Button` and provide an overview to implementing an alternate look & feel design.

`LabelledComponentModel`

Most of the `Model` implementations are in `sun.awt.otk` and do not need additional modification to support an alternate look & feel design. `LabelledComponentModel` is an excellent example of why this is the case. Because it only needs to record the current state of the component and the mechanism for displaying and interacting that component is independent of how it is recorded, it is unlikely that any additional modifications will be needed for this class. Of course, `LabelledComponentModel` can be extended if necessary.

`ButtonView`

The `View` classes are the most time-consuming part of developing a new look & feel design. Most of the `View`-specific classes are in `sun.awt.touchable` and can be modified or replaced. The simplest approach to take is to concentrate on the drawing code in `DrawUtils.drawButton`. In some more complex cases, it may be necessary to have `View`-specific drawing code in a `View` class.

`ArmOnPress`

There are two ways to approach the issue of `Controllers`:

- Use a different mixture of prebuilt `Controller` classes. In this case, just clone `TouchableToolkit.createButton` and use different `Controller` classes.
- Write a new `Controller` class by extending `SimpleController`. As with `Models`, most of the `Controller` classes are in `sun.awt.otk`.

Optimizations

As a reference implementation, Touchable must serve certain goals that a production implementation does not. The following list contains some ideas for optimizing the Touchable implementation:

- Get rid of extra `Controllers`.
- Reduce object creation.
- Remove some of the generality of Touchable
- Get rid of runtime scaling.
- Share a single `DrawUtils` instance between GUI classes.

- Implement a single shared `Controller` class.
