Article

A Swing Architecture Overview

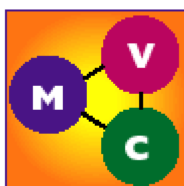
Swing 架构概述

The Inside Story on JFC Component Design
JFC 组件设计探究

By Amy Fowler

作者：Amy Fowler

中文翻译：Azure



Most Swing developers know by now that Swing components have a separable model-and-view design. And many Swing users have run across articles saying that Swing is based on something called a "modified MVC (model-view-controller) architecture."

大多 swing 开发人员现在都知道 swing 的组件采用的是模型-与-视图分离的设计，也有大量的 swing 使用者撰文陈述 swing 是基于"改良的 MVC（模型-视图-控制者）结构"。

But accurate explanations of how Swing components are designed, and how their parts all fit together, have been hard to come by -- until now.

但是在此之前对于 swing 组件是怎么设计的，它们的各部分是如何组合到一块的都描述的不是很准确。

The silence ends with the publication of this article, a major white paper on Swing component design. It provides a comprehensive technical overview of Swing's modified MVC structure and demystifies many other facets of Swing component architecture as well.

直到这篇 swing 组件设计白皮书文章的发表之后才让事情真相大白。本文对 swing 的改良的 MVC 架构做了一个很全面的技术概述，同时也对很多关于 swing 组件架构的其它方面的技术也做了概述。

This document presents a technical overview of the Swing component architecture. In particular, it covers the following areas in detail:

这篇文档对 swing 组件的架构做一个技术概论。详细来说，它具体包括以下几个领域：

- *Design goals*
- *Roots in MVC*
- *Separable model architecture*
- *Pluggable look-and-feel architecture*

- *设计目标*
- *MVC 的来源*
- *可分离模型架构*
- *可拔插感官架构*

Design Goals 设计目标

The overall goal for the Swing project was:

Swing 项目的所有目标如下：

To build a set of extensible GUI components to enable developers to more rapidly develop powerful Java front ends for commercial applications.

为 java 开发人员建立一套可扩展的 GUI 组件，以助其快速开发出强大的面向 java 的商用软件。

To this end, the Swing team established a set of design goals early in the project that drove the resulting architecture. These guidelines mandated that Swing would:

为达到此目的，swing 开发组在项目初期建立了一套设计目标来制定最终的架构。这些指导要求 swing 必须：

1. *Be implemented entirely in Java* to promote cross-platform consistency and easier maintenance.

   完全使用 java 来实现以保证跨平台性和易维护性。

2. *Provide a single API capable of supporting multiple look-and-feels* so that developers and end-users would not *be locked into a single look-and-feel.*

   提供可以支持多感官的单一的 API，这样用户和开发人员将不会被绑定到单一的感官上面。

3. *Enable the power of model-driven programming* without requiring it in the highest-level API.

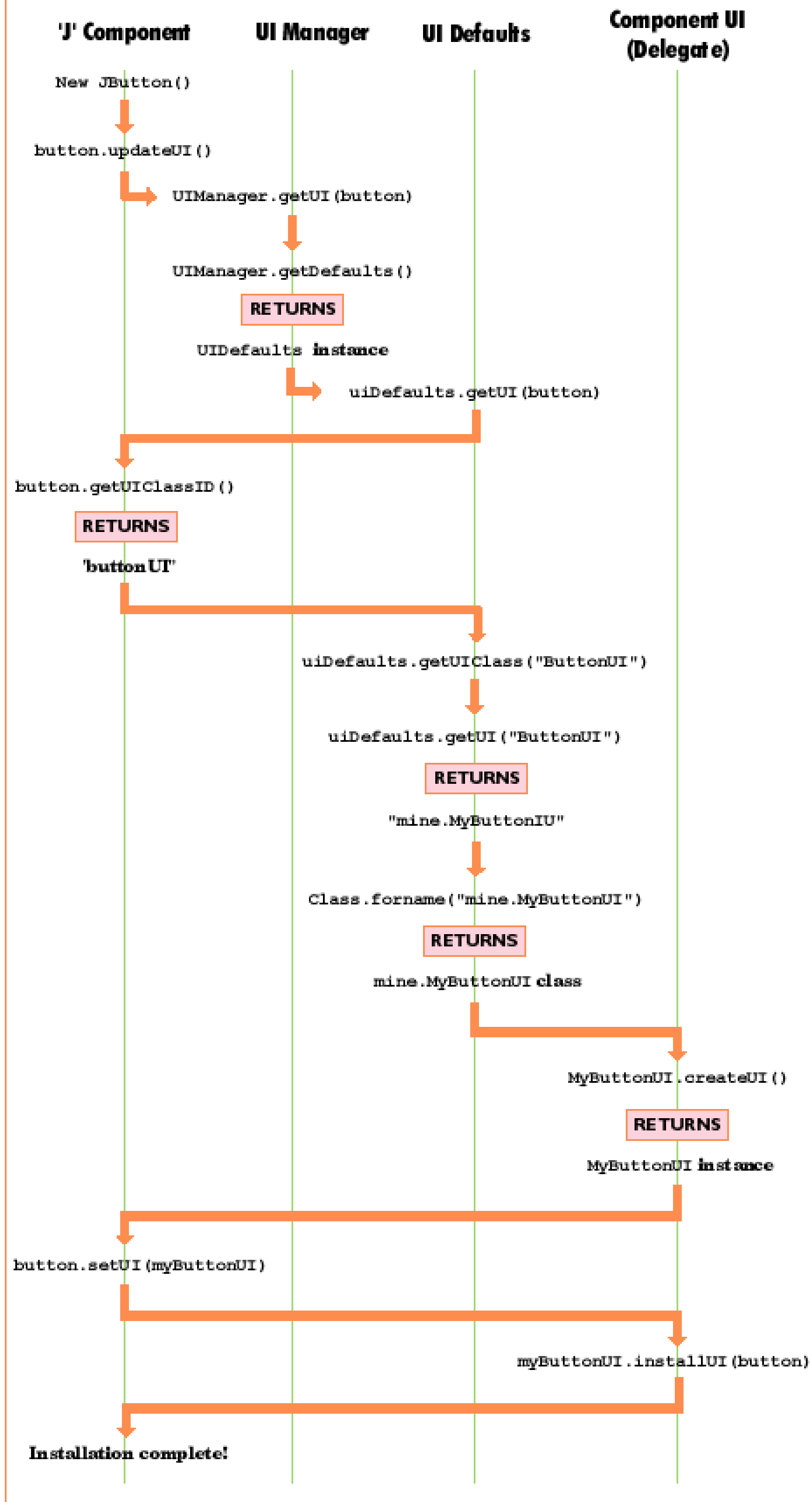
   具有不需要在最上层使用 API 的模型驱动编程能力。

4. *Adhere to JavaBeans design principles* to ensure that components behave well in IDEs and builder tools.

   遵循 JavaBeans 设计规范以保证组件能够获得集成开发工具的良好支持。

5. *Provide compatibility with AWT APIs where there is overlapping, to leverage the AWT knowledge base and ease porting.*

   让功能重叠的 AWT APIs 可以兼容，使得 AWT 为基础知识能够轻松过渡。

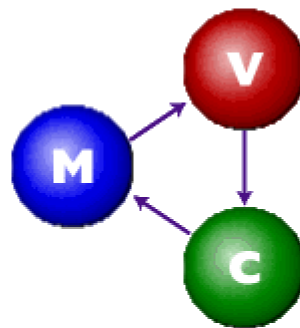

The process of installing a UI delegate
'J' Component
UI Manager
UI Defaults
Component UI (Delegate)
New JButton()
button.updateUI()
UIManager.getUI(button)
UIManager.getDefaults()
RETURNS
UIDefaults instance
uiDefaults.getUI(button)
button.getUIClassID()
RETURNS
'buttonUI'
uiDefaults.getUIClass("ButtonUI")
uiDefaults.getUI("ButtonUI")
RETURNS
"mine.MyButtonIU"
Class.forname("mine.MyButtonUI")
RETURNS
mine.MyButtonUI class
MyButtonUI.createUI()
RETURNS
MyButtonUI instance
button.setUI(myButtonUI)
myButtonUI.installUI(button)
Installation complete!

Roots in MVC　MVC 的来源

Swing architecture is rooted in the *model-view-controller* (*MVC*) design that dates back to SmallTalk. MVC architecture calls for a visual application to be broken up into three separate parts:

Swing 结构的来源要追溯到 SmallTalk 的模型-视图-控制者（MVC）设计。MVC 架构要求可视化的应用程序分为三个独立的部分：

- A *model* that represents the data for the application.
  模型 代表应用软件的数据。
- The *view* that is the visual representation of that data.
  视图 指数据的可视化展现。
- A *controller* that takes user input on the view and translates that to changes in the model.
  控制者 获得视图上用户的输入并把它送到模型里面做相应的改变。



Early on, MVC was a logical choice for Swing because it provided a basis for meeting the first three of our design goals within the bounds of the latter two.

早期，MVC 对 swing 来说是一个合理的选择，因为它基本上能够满足在后来两者的约束下实现我们的前三个设计目标。

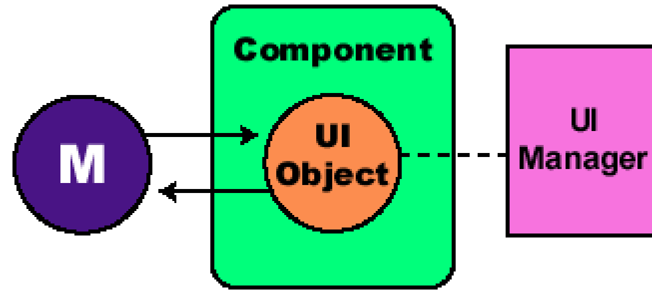
The first Swing prototype followed a traditional MVC separation in which each component had a separate model object and delegated its look-and-feel implementation to separate view and controller objects.

在开始的时候，swing 原型遵循了传统的 MVC 分离式架构，每个组件都有独立的模型对象，并且可以通过代理它的感官实现来分离视图和控制者对象。

The delegate　代理

We quickly discovered that this split didn't work well in practical terms because the view and controller parts of a component required a tight coupling (for example, it was very difficult to write a generic controller that didn't know specifics about the view). So we collapsed these two entities into a single UI (user-interface) object, as shown in this diagram:

我们很快发现这样分割在实际应用中工作的不是很好，因为一个组件的视图和控制者部分需要紧密的耦合在一起（比如，一般很难写出一个不知道视图细节的控制者）。所以我们将这两个实体捏合成一个单一的 UI（用户接口）对象，如图所示：



(The UI delegate object shown in this picture is sometimes called a delegate object, or *UI delegate*. The UI delegate used in Swing is described in more detail in the Pluggable look-and-feel section of this article, under the subheading "The UI delegate".)

（图中的 UI 代理对象有时被称为代理对象，或者 UI 代理，swing 中的 UI 代理在本文的可拔插感官部分有更详细的描述，副标题为“UI 代理”）。

As the diagram illustrates, Swing architecture is loosely based -- but not *strictly* based -- on the traditional MVC design. In the world of Swing, this new quasi-MVC design is sometimes referred to a *separable model architecture*.

如图所示，swing 架构基本上是基于传统的 MVC 架构设计－但不是很严格。在 swing 的世界里，这种类 MVC 架构设计有时归类于模型可分离架构。

Swing's separable model design treats the model part of a component as a separate element, just as the MVC design does. But Swing collapses the view and controller parts of each component into a single UI (user-interface) object.

Swing 的模型可分离设计和 MVC 架构设计一样把一个组件的模型部分当作一个单独的元件。但是 swing 把视图和控制者部分捏合到一块变成一个独立的 UI（用户接口）对象。

To MVC or not to MVC?　　到底是还是不是 MVC 架构？

One noteworthy point is that as an application developer, you should think of a component's view/controller responsibilities as being handled by the generic component class (such as. JButton, JTree, and so on). The component class then delegates the look-and-feel-specific aspects of those responsibilities to the UI object that is provided by the currently installed look-and-feel.

一个值得注意的地方是作为一个应用软件开发人员，你应考虑一般的组件类（如：JButton，JTree 等等）赋予给组件的视图/控制者的责任。

For example, the code that implements double-buffered painting is in Swing's JComponent class (the "mother" of most Swing component classes), while the code that renders a JButton's label is in the button's *UI delegate class*. The preceding diagram illustrates this subtle (and often confusing) point:

举例来说，双缓冲绘制的代码是在 swing 的 JComponent 类（大部分 swing 组件类的“母亲”）中实现的，然而渲染一个 JButton 的 label 的代码是在 button 的 UI 代理类中实现的。在前面的图中说明了这个微妙的（经常容易使人迷惑的）地方。

So Swing does have a strong MVC lineage. But it's also important to reiterate that our MVC architecture serves two distinct purposes:

所以 swing 拥有比较浓的 MVC 血统。但是重申我们的 MVC 架构与传统的 MVC 架构之间具有两种截然不同的设计目的依然很重要。

First, separating the model definition from a component facilitates model-driven programming in Swing.

首先，把模型从一个组件里面分离出来定义方便了 swing 里面的模型驱动编程。

Second, the ability to delegate some of a component's view/controller responsibilities to separate look-and-feel objects provides the basis for Swing's pluggable look-and-feel architecture.
Although these two concepts are linked by the MVC design, they may be treated somewhat orthogonally from the developer's perspective. The remainder of this document will cover each of these mechanisms in greater detail.

第二，为一个组件的视图/控制者代理一些分离感官对象的责任的能力为 swing 的可拔插感官架构提供了基础。

虽然这两个概念因为 MVC 设计而产生联系，从开发人员的角度来看它们可能被认为有点交叉。本文接下来部分将会对所有的这些机制做非常详细的描述。

Separable model architecture 模型可分离架构

It is generally considered good practice to center the architecture of an application around its data rather than around its user interface. To support this paradigm, Swing defines a separate model interface for each component that has a logical *data* or *value* abstraction. This separation provides programs with the option of plugging in their own model implementations for Swing components.

一般认为以应用的数据为架构的中心比以用户接口为中心的做法要更好，为了支持这种范例， swing 为每个拥有数据逻辑或数据抽象的组件都定义了一个单独的模型接口。这种分离使得程序可以在 swing 组件里面选择拔插它们自己实现的模型。

The following table shows the component-to-model mapping for Swing.

下面的表格显示了 swing 里面的组件和模型之间的对应关系：

Component	Model Interface	Model Type
JButton	ButtonModel	GUI
JToggleButton	ButtonModel	GUI/data
JCheckBox	ButtonModel	GUI/data
JRadioButton	ButtonModel	GUI/data
JMenu	ButtonModel	GUI

JMenuItem	ButtonModel	GUI
JCheckBoxMenuItem	ButtonModel	GUI/data
JRadioButtonMenuItem	ButtonModel	GUI/data
JComboBox	ComboBoxModel	data
JProgressBar	BoundedRangeModel	GUI/data
JScrollBar	BoundedRangeModel	GUI/data
JSlider	BoundedRangeModel	GUI/data
JTabbedPane	SingleSelectionModel	GUI
JList	ListModel	data
JList	ListSelectionModel	GUI
JTable	TableModel	data
JTable	TableColumnModel	GUI
JTree	TreeModel	data
JTree	TreeSelectionModel	GUI
JEditorPane	Document	data
JTextPane	Document	data
JTextArea	Document	data
JTextField	Document	data
JPasswordField	Document	data

GUI-state vs. application-data models GUI 状态模型与应用数据模型

The models provided by Swing fall into two general categories: *GUI-state models* and *application-data models*.

Swing 的模型大致分为两种：GUI 状态模型和应用数据模型。

GUI-state models GUI 状态模型

GUI state models are interfaces that define the visual status of a GUI control, such as whether a button is pressed or armed, or which items are selected in a list. GUI-state models typically are relevant only in the context of a graphical user interface (GUI). While it is often useful to develop programs using GUI-state model separation -- particularly if multiple GUI controls are linked to a common state (such as in a shared whiteboard program), or if manipulating one control automatically changes the value of another -- the use of GUI-state models is not required by Swing. It is possible to manipulate the state of a GUI control through top-level methods on the component, without any direct interaction with the model at all. In the preceding table, GUI-state models in Swing are highlighted in green.

GUI 状态模型是定义一个 GUI 控制器的可视状态的接口，比如 button 是否按下或松开，list 里面的 items 是否被选上。GUI 状态模型常常是与图形用户接口（GUI）的上下文有联系的。使用 GUI 状态模型分离对开发程序常常非常有帮助，特别是多个 GUI 控制器与一个公用的状态（比如在一个共享白板程序）有联系的时候，或者如果操纵一个控制器自动改变其它控制器的值时候，swing 不需要使用 GUI 状态模型。通过组件的最顶层的方法可以操纵 GUI 控制器的状态，完全不需要直接与模型进行交互。在前文描述的表格中， swing 的 GUI 状态模型用绿色高亮显示。

Application-data models　　应用数据模型

An *application-data model* is an interface that represents some quantifiable data that has meaning primarily in the context of the application, such as the value of a cell in a table or the items displayed in a list. These data models provide a very powerful programming paradigm for Swing programs that need a clean separation between their application data/logic and their GUI. For truly data-centric Swing components, such as JTree and JTable, interaction with the data model is strongly recommended. Application-data models are highlighted in red in the table presented at the beginning of this section.

应用数据模型是指表示应用上下文中的一些具有重要意义的可以计量的数据的接口， 像一个 table 里面的一个 cell 的值或在 list 里面显示的 items 的值等。这些数据模型为需要在 GUI 与应用数据/逻辑之间有彻底的分离的 swing 程序提供了非常强大的编程范例。对于像 JTree 和 JTable 这样真正以数据为中心的 swing 组件，强烈推荐与数据模型进行交互。应用数据模型在本章节开始部分显示的表格里面使用红色进行了高亮显示。

Of course with some components, the model categorization falls somewhere in between GUI state models and application-data models, depending on the context in which the model is used. This is the case with the `BoundedRangeModel` on `JSlider` or `JProgressBar`. These models are highlighted in purple in the preceding table.

当然，有一些组件在 GUI 状态模型和应用数据模型之间的分类取决于该模型在上下文中的使用。JProgressBar 或者 JSlider 里面的 `BoundedRangeModel` 就是这种情况。这些模型在前面的 table 中用紫色做了高亮显示。

Swing's separable model API makes no specific distinctions between GUI state models and application-data models; however, we have clarified this difference here to give developers a better understanding of when and why they might wish to program with the separable models.

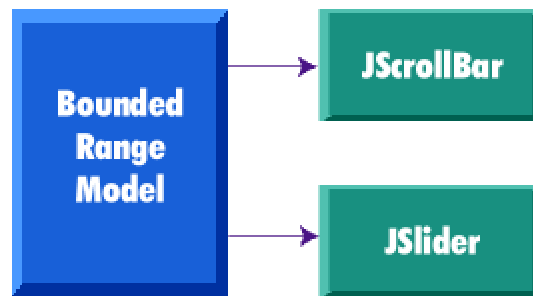
Swing 的可分离模型 API GUI 状态模型和应用数据模型之间没有明显的区别，我们在这里阐明这些不同之处的目的是 帮助开发人员更好的理解为什么和什么时候他们可能会希望使用到可分离的模型编程。

Shared model definitions　　共享模型定义

Referring again to the table at the beginning of this section, notice that model definitions are shared across components in cases where the data abstraction for each component is similar enough to support a single interface without over-genericizing that interface. Common models enable automatic connectability between component types. For example, because both `JSlider` and `JScrollbar` use the `BoundedRangeModel` interface, a single `BoundedRangeModel` instance could be plugged into both a `JScrollbar` and a `JSlider` and their visual state would always remain in sync.

再次谈到本章开始的时候显示的 table，注意在组件之间共享的模型定义在每个组件之间的数据抽象是如此相似，以至于可以支持同一个接口。公共模型具有能自动配合多个组件类型的能力。举例来说， `JSlider` 和 `JScrollbar` 都使用

BoundedRangeModel 接口，一个 BoundedRangeModel 实例同时可以被一个 JScrollbar 和一个 JSlider 使用，它们的可视状态可以一直都保持同步。



The separable-model API　可分离模型 API

Swing components that define models support a *JavaBeans bound property* for the model. For example, JSlider uses the BoundedRangeModel interface for its model definition. Consequently, it includes the following methods:

Swing 组件定义的模型支持 *JavaBeans 值绑定* 模型。比如，JSlider 使用了 BoundedRangeModel 接口作它的模型定义，因此，它包含下面的方法：

```
public BoundedRangeModel getModel()
public void setModel(BoundedRangeModel model)
```

All Swing components have one thing in common: If you don't set your own model, a default is created and installed internally in the component. The naming convention for these default model classes is to prepend the interface name with "Default." For JSlider, a DefaultBoundedRangeModel object is instantiated in its constructor:

所有的 swing 组件都有一样共同的东西： 如果你不设置你自己的模型，在组件内部将会自动创建和安装一个缺省的模型。这些模型类的名称约定为在接口名称的最前面加一个“Default”。如 JSlider，在它的构造函数里面就有一个 DefaultBoundedRangeModel 对象的示例。

```
public JSlider(int orientation, int min,
               int max, int value)
{
    checkOrientation(orientation);
    this.orientation = orientation;
    this.model =
      new DefaultBoundedRangeModel(value, 0, min, max);
    this.model.addChangeListener(changeListener);
    updateUI();
}
```

If a program subsequently calls setModel(), this default model is replaced, as in the following example:

如果程序后来调用了 setModel()方法，这个缺省的模型将会被替换，如例所示：

```
JSlider slider = new JSlider();
BoundedRangeModel myModel =
   new DefaultBoundedRangeModel() {
      public void setValue(int n) {
         System.out.println("SetValue: "+ n);
         super.setValue(n);
      }
   });
slider.setModel(myModel);
```

For more complex models (such as those for JTable and JList), an abstract model implementation is also provided to enable developers to create their own models without starting from scratch. These classes are prepended with "Abstract".

对于更复杂的模型（比如 JTable 和 JList 等的模型），为开发人员提供了一个抽象的模型实现，这样他们不至于一切都从头开始。这些类的前缀为“Abstract”。

For example, JList's model interface is ListModel, which provides both DefaultListModel and AbstractListModel classes to help the developer in building a list model.

比如，JList 的模型接口为 ListModel，该模型接口同时提供了 DefaultListModel 和 AbstractListModel 类来帮助开发人员组建一个 list 模型。

Model change notification 模型改变通知

Models must be able to notify any interested parties (such as views) when their data or value changes. Swing models use the *JavaBeans Event model* for the implementation of this notification. There are two approaches for this notification used in Swing:

当模型的数据或者值改变之后必须能够通知所有相关的部分（如视图）。Swing 模型使用 *JavaBeans Event* 模型来实现这个通知。在 swing 中使用了两个方案来实现这个通知。

Send a *lightweight notification* that the state has "changed" and require the listener to respond by sending a query back to the model to find out *what* has changed. The advantage of this approach is that a single event instance can be used for all notifications from a particular model -- which is highly desirable when the notifications tend to be high in frequency (such as when a JScrollBar is dragged).

当状态发生改变的时候发送一个轻量级通知，这需要事件监听器通过查询模型发生了什么变化来作出响应。这个方案的优点是一个事件实例可以同时被一个模型的所有通知来使用，这在通知被高频率使用的时候特别需要（比如当一个 JScrollBar 被拖动的时候）。

Send a *stateful notification* that describes more precisely *how* the model has changed. This alternative requires a new event instance for each notification. It is desirable when a generic notification doesn't provide the listener with enough

information to determine efficiently what has changed by querying the model (such as when a column of cells change value in a JTable).

发送一个详细描述模型发生了什么改变的有状态的通知。这个方案需要每个通知都有一个新的事件实例。使用这个方案的场合是在一般的通知无法提供足够的信息给事件监听器通过查询模型来有效判断什么发生了变化（比如当一个 JTable 的一列单元的值发生了变化）。

Lightweight notification 轻量级通知

The following models in Swing use the *lightweight notification*, which is based on the ChangeListener/ChangeEvent API:

下面的 swing 模型使用了*轻量级通知*，它们基于 ChangeListener/ChangeEvent API：

Model	Listener	Event
BoundedRangeModel	ChangeListener	ChangeEvent
ButtonModel	ChangeListener	ChangeEvent
SingleSelectionModel	ChangeListener	ChangeEvent

The ChangeListener interface has a single generic method:

这个 ChangeListener 接口有只有一个很普通的方法：

```
public void stateChanged(ChangeEvent e)
```

The only state in a ChangeEvent is the event "source." Because the source is always the same across notifications, a single instance can be used for all notifications from a particular model. Models that use this mechanism support the following methods to add and remove ChangeListeners:

在一个 ChangeEvent 中只有一个唯一的状态即事件的“源”，因为在很多通知中都使用同一个源，在某些特定的模型里一个 event 实例可以被所有的通知使用。模型采用该机制来支持下面的添加和移除 ChangeListeners 的方法。

```
public void addChangeListener(ChangeListener l)
public void removeChangeListener(ChangeListener l)
```

Therefore, to be notified when the value of a JSlider has changed, the code might look like this:

所以，当一个 JSlider 的值发生改变的时候发出通知的代码可能如下：

```
JSlider slider = new JSlider();
BoundedRangeModel model = slider.getModel();
model.addChangeListener(new ChangeListener() {
   public void stateChanged(ChangeEvent e) {
```

```
        // need to query the model
        // to get updated value...
        BoundedRangeModel m =
           (BoundedRangeModel)e.getSource();
        System.out.println("model changed: " +
            m.getValue());
    }
});
```

To provide convenience for programs that don't wish to deal with separate model objects, some Swing component classes also provide the ability to register ChangeListeners directly on the component (so the component can listen for changes on the model internally and then propagates those events to any listeners registered directly on the component). The only difference between these notifications is that for the model case, the event source is the model instance, while for the component case, the source is the component.

有人不希望和可分离模型对象打交道，为了提供方便，有些 swing 类还提供了直接在组件上注册 ChangeListeners 的能力（这样组件能够在内部监听到模型的变化，然后将那些事件传播给任何直接注册在组件上的监听器），这些通知的唯一不同之处在于对于模型来说，事件源是模型的实例，而对组件来说，事件源是组件。

So we could simplify the preceding example to:

我们可以简单的举个例子：

```
JSlider slider = new JSlider();
slider.addChangeListener(new ChangeListener() {
    public void stateChanged(ChangeEvent e) {
        // the source will be
        // the slider this time..
        JSlider s = (JSlider)e.getSource();
        System.out.println("value changed: " +
           s.getValue());
    }
});
```

Stateful notification　　有状态的通知

Models that support *stateful notification* provide event Listener interfaces and event objects specific to their purpose. The following table shows the breakdown for those models:

模型根据事件监听器接口和事件对象的用途提供有状态的通知，下面的表格显示了这些模型的详细分类：

Model	Listener	Event
ListModel	ListDataListener	ListDataEvent
ListSelectionModel	ListSelectionListener	ListSelectionEvent
ComboBoxModel	ListDataListener	ListDataEvent
TreeModel	TreeModelListener	TreeModelEvent
TreeSelectionModel	TreeSelectionListener	TreeSelectionEvent
TableModel	TableModelListener	TableModelEvent
TableColumnModel	TableColumnModel-Listener	TableColumnModel-Event
Document	DocumentListener	DocumentEvent
Document	UndoableEditListener	UndoableEditEvent

The usage of these APIs is similar to the lightweight notification, except that the listener can query the event object directly to find out what has changed. For example, the following code dynamically tracks the selected item in a JList:

这些 APIs 的使用方法与轻量级通知相似，除了那些可以直接查询事件对象来发现什么发生的变化的事件监听器之外。举例来说，下面的代码动态的跟踪了 JList 中被选上的 item。

```
String items[] = {"One", "Two", "Three");
JList list = new JList(items);
ListSelectionModel sModel = list.getSelectionModel();
sModel.addListSelectionListener
       (new ListSelectionListener() {
   public void valueChanged(ListSelectionEvent e) {
      // get change information directly
      // from the event instance...
      if (!e.getValueIsAdjusting()) {
         System.out.println("selection changed: " +
           e.getFirstIndex());
      }
   }
});
```

Automatic View Updates 自动更新视图

A model does not have any intrinsic knowledge of the view that represents it. (This requirement is critical to enable multiple views on the same model). Instead, a model has only a list of listeners interested in knowing when its state has changed. A Swing component takes responsibility for hooking up the appropriate model listener so that it can appropriately repaint itself as the model changes (if you find that a component is not updating automatically when the model changes, it is a bug!). This is true whether a default internal model is used or whether a program installs its own model implementation.

模型与表现它的视图没有任何内在联系。（具有相关性的做法在多个视图使用同一个模型的时候会受到指责）。模型只有一些对它的状态什么发生变化感兴趣的事件监听器。一个 swing 组件负责钩住与其相关的模型监听器，这样当模型发生变化的时候它可以准确的重绘它本身（如果你发现当模型发生变化时一个组件没有自动更新，这会是一个 bug！）。不管是使用缺省的内部模型还是使用程序自己实现的模型都是这样的。

Ignoring models completely 完全忽视模型

As mentioned previously, most components provide the model-defined API directly in the component class so that the component can be manipulated without interacting with the model at all. This is considered perfectly acceptable programming practice (especially for the GUI-state models). For example, following is JSlider's implementation of getValue(), which internally delegates the method call to its model:

前文提到，大多组件提供了在组件类中直接进行定义模型 API，所以可以不通过与模型交互就可以对组件进行操作。这是非常令人满意的编程范例（特别是 GUI 状态模型）。比如，下面是 JSlider 的 getValue(),方法实现，它会在内部代理调用它的模型的方法。

```
public int getValue() {
    return getModel().getValue();
}
```

And so programs can simply do the following:

于是程序就变的如下这么简单了：

```
JSlider slider = new JSlider();
int value = slider.getValue();
//what's a "model," anyway?
```

Separable model summary 可分离模型总结

So while it's useful to understand how Swing's model design works, it isn't necessary to use the model API for all aspects of Swing programming. You should carefully consider your application's individual needs and determine where the model API will enhance your code without introducing unnecessary complexity.

理解 swing 的模型设计是如何工作的是非常有用的，在 swing 编程的各个场合都使用模型 API 是不必要的。你应该仔细考虑你的应用的具体需要，然后决定在什么地方使用模型 API，不用搞一些不必要的复杂性就可以改善你的代码。

In particular, we recommend the usage of the Application-Data category of models for Swing (models for `JTable`, `JTree`, and the like) because they can greatly enhance the scalability and modularity of your application over the long run.

在特殊情况下，我们推荐在 swing 中使用应用数据类的模型（比如 JTable 和 JTree 等的模型），因为从长远来看它们可以很大的提高你的应用的模块性和可测量性。

Pluggable look-and-feel architecture 可拔插的感官架构

Swing's pluggable look-and-feel architecture allows us to provide a single component API without dictating a particular look-and-feel. The Swing toolkit provides a default set of look-and-feels; however, the API is "open" -- a design that additionally allows developers to create new look-and-feel implementations by either extending an existing look-and-feel or creating one from scratch. Although the pluggable look-and-feel API is extensible, it was intentionally designed at a level below the basic component API in such a way that a developer does not need to understand its intricate details to build Swing GUIs. (But if you *want* to know, read on . . .)

Swing 的可拔插感官架构允许我们可以使用单一的组件 API，而不用指定特定的感官。Swing 的工具包里面提供了一套缺省的感官，API 是“开放”的，这样的设计既可以让开发人员继承已有的感官来创建新的感官也可以从头开始创建一套新的感官。虽然可拔插感官 API 架构具有可扩展性，但它被有意设计在基本组件 API 的下层，这样使得开发人员开发 swing GUIs 的时候不需要明白它的所有细节。（如果你想知道的话，继续往下读。。。）

While we don't expect (or advise) the majority of developers to create new look-and-feel implementations, we realize PL&F is a very powerful feature for a subset of applications that want to create a unique identity. As it turns out, PL&F is also ideally suited for use in building GUIs that are accessible to users with disabilities, such as visually impaired users or users who cannot operate a mouse.

我们不期望（或建议）大多数的开发人员都创建新的感官实现，我们认识到可拔插感官特性为希望创建具有一致性的 系列应用提供了有力的支持。可拔插感官还为创建具有为残障人士提供可达性的 GUIs 提供完美的支持。

In a nutshell, pluggable look-and-feel design simply means that the portion of a component's implementation that deals with the presentation (the look) and event-handling (the feel) is delegated to a separate UI object supplied by the currently installed look-and-feel, which can be changed at runtime.

综上所述，可拔插感官设计简单的来说就是组件的表现（外观）和事件处理（感觉）部分的实现被当前安装的可以在 运行时随时被更换的感官的独立的 UI 对象来代理。

The pluggable look-and-feel API 可拔插感官的 API

The pluggable look-and-feel API includes:

可拔插感官 API 包括：

- Some small hooks in the Swing component classes.

  Swing 组件类的一些小钩子。

- Some top-level API for look-and-feel management.

  一些感官管理的顶层 API。

- A more complex API that actually implements look-and-feels in separate packages.

  在独立的包中真正实现感官的较复杂的 API。

The component hooks 组件钩子

Each Swing component that has look-and-feel-specific behavior defines an abstract class in the `swing.plaf` package to represent its UI delegate. The naming convention for these classes is to take the class name for the component, remove the "J" prefix, and append "UI." For example, `JButton` defines its UI delegate with the *plaf* class `ButtonUI`.

每个具有感官具体行为的 swing 组件在 `swing.plaf` 包中都有一个抽象类表示它的 UI 代理。命名规则是组件的名称除去前缀“J”，加上后缀“UI”。比如， `JButton` 的 *plaf* 类 UI 代理定义为 `ButtonUI`。

The UI delegate is created in the component's constructor and is accessible as a JavaBeans bound property on the component. For example, `JScrollBar` provides the following methods to access its UI delegate:

UI 代理在组件的构造器里面创建，可以像 JavaBeans 属性绑定一样访问。比如， `JScrollBar` 提供了下面的方法来访问它的 UI 代理：

```
public ScrollBarUI getUI()
public void setUI(ScrollBarUI ui)
```

This process of creating a UI delegate and setting it as the "UI" property for a component is essentially the "installation" of a component's look-and-feel.

创建和像“UI”属性一样设置一个组件的 UI 代理的过程取决于安装的组件的感官。

Each component also provides a method which creates and sets a UI delegate for the "default" look-and-feel (this method is used by the constructor when doing the installation):

每个组件还提供了一个方法来为缺省的感官创建和设置 UI 代理（这个方法在安装的时候会被构造器调用）

```
public void updateUI()
```

A look-and-feel implementation provides concrete subclasses for each abstract *plaf* UI class. For example, the Windows look-and-feel defines `WindowsButtonUI`, a `WindowsScrollBarUI`, and so on. When a component installs its UI delegate, it must have a way to look up the appropriate concrete class name for the current default look-and-feel dynamically. This operation is performed using a hash table in which the key is defined programmatically by the `getUIClassID()` method in the component. The convention is to use the *plaf* abstract class name for these keys. For example, `JScrollbar` provides:

感官的实现为每个抽象 *plaf* UI 类提供了具体的子类。比如，windows 感官定义了 WindowsButtonUI 和 WindowsScrollBarUI 等。当一个组件安装它的 UI 代理的时候，它必须想办法动态的为当前感官找到相应的具体类名。这个操作具体是通过使用 key 自动定义在组件的 getUIClassID()方法里的 hash 表。

```
public String getUIClassID() {
    return "ScrollBarUI";
}
```

Consequently, the hash table in the Windows look-and-feel will provide an entry that maps "ScrollBarUI" to

"com.sun.java.swing.plaf.windows.WindowsScrollBarUI"

所以，windows 感官中的 hash 表提供了一个值映射“ScrollBarUI”到“com.sun.java.swing.plaf.windows.WindowsScrollBarUI”

Look-and-feel management 感官管理

Swing defines an abstract LookAndFeel class that represents all the information central to a look-and-feel implementation, such as its name, its description, whether it's a native look-and-feel -- and in particular, a hash table (known as the "Defaults Table") for storing default values for various look-and-feel attributes, such as colors and fonts.

Swing 定义了一个 LookAndFeel 抽象类来描述以感官实现为中心的所有信息，比如它的名称，它的描述，是否为本地感官等，在一张 hash 表里面（就是所谓的“缺省值表”）来存储各种各样的感官属性缺省值，如颜色和字体等。

Each look-and-feel implementation defines a subclass of LookAndFeel (for example, swing.plaf.motif.MotifLookAndFeel) to provide Swing with the necessary information to manage the look-and-feel.

每个感官实现都定义了一个 LookAndFeel 子类（例如 swing.plaf.motif.MotifLookAndFeel）来为 swing 对感官的管理提供必要的信息。

The UIManager is the API through which components and programs access look-and-feel information (they should rarely, if ever, talk directly to a LookAndFeel instance). UIManager is responsible for keeping track of which LookAndFeel classes are available, which are installed, and which is currently the default. The UIManager also manages access to the Defaults Table for the current look-and-feel.

组件和程序通过 UIManager API 来访问感官信息（很少，几乎没有直接和 LookAndFeel 实例打交道的）。UIManager 负责跟踪哪些 LookAndFeel 类被提供了，哪些被安装了，哪个是当前缺省使用的。UIManager 还管理对当前的感官的“Default Table”的访问。

The 'default' look and feel 缺省感官

The UIManager also provides methods for getting and setting the current default LookAndFeel:

UIManager 还提供方法来 getting 和 setting 当前的感官：

```
public static LookAndFeel getLookAndFeel()
```

```
public static void
   setLookAndFeel(LookAndFeel newLookAndFeel)

public static void
   setLookAndFeel(String className)
```

As a default look-and-feel, Swing initializes the cross-platform Java look and feel (formerly known as "Metal"). However, if a Swing program wants to set the default Look-and-Feel explicitly, it can do that using the `UIManager.setLookAndFeel()` method. For example, the following code sample will set the default Look-and-Feel to be CDE/Motif:

缺省情况下，Swing 初始化了一套跨平台的 java 感官（以前被称为“Metal”），如果 swing 程序希望显式的设置缺省感官，可以使用 UIManager.setLookAndFeel()方法。例如，下面的代码样例将会设置缺省感官为 CDE/Motif:感官。

```
UIManager.setLookAndFeel(
   "com.sun.java.swing.plaf.motif.MotifLookAndFeel");
```

Sometimes an application may not want to specify a particular look-and-feel, but instead wants to configure a look-and-feel in such a way that it dynamically matches whatever platform it happens to be running on (for instance, the. Windows look-and-feel if it is running on Windows NT, or CDE/Motif if it running on Solaris). Or, perhaps, an application might want to lock down the look-and-feel to the cross-platform Java look and feel.

有时候一个应用可能不希望指定特定的感官，而是希望用这种方式来配置无论它运行在哪个平台上面都动态匹配该平台的感官（比如当它运行在 Windows NT 平台上，它会使用 Windows 感官，而在 Soloris 平台上面使用 CDE/Motif 感官）。或者说，应用软件可能希望使用跨平台的 java 感官。

The `UIManager` provides the following static methods to programmatically obtain the appropriate `LookAndFeel` class names for each of these cases:

针对所有的这些情况，UIManager 提供了下面的静态方法来自动获取合适的 LookAndFeel 类名。

```
public static String
   getSystemLookAndFeelClassName()
public static String
   getCrossPlatformLookAndFeelClassName()
```

So, to ensure that a program always runs in the platform's system look-and-feel, the code might look like this:

所以，为了保证程序总是使用系统平台的感官，代码可能是这样的：

```
UIManager.setLookAndFeel(
   UIManager.getSystemLookAndFeelClassName());
```

Dynamically Changing the Default Look-and-Feel 动态改变缺省感官

When a Swing application programmatically sets the look-and-feel (as described above), the ideal place to do so is *before any Swing components are instantiated*. This is because the UIManager.setLookAndFeel() method makes a particular LookAndFeel the current default by loading and initializing that LookAndFeel instance, but it does *not* automatically cause any existing components to change their look-and-feel.

当 swing 应用程序动态设置感官（像上面所描述的那样），这样做的最好地方是在所有 swing 组件实例化之前。这是因为 UIManager.setLookAndFeel()方法通过加载和初始化特定的 LookAndFeel 实例为当前缺省感官，但是它不会自动使得任何已经存在的组件改变它们的感官。

Remember that components initialize their UI delegate at *construct* time, therefore, if the current default changes after they are constructed, they will *not* automatically update their UIs accordingly. It is up to the program to implement this dynamic switching by traversing the containment hierarchy and updating the components individually. (**NOTE:** Swing provides the SwingUtilities.updateComponentTreeUI() method to assist with this process).

记住组件初始化它们的 UI 代理构造的时间，如果它们构造之后当前缺省感官发生了变化，它们将不会自动更新它们的 UIs，它们必须通过编码实现动态的转换来对组件更新。(注意：swing 提供了 SwingUtilities.updateComponentTreeUI()方法来支持这个处理）。

The look-and-feel of a component can be updated at any time to match the current default by invoking its updateUI() method, which uses the following static method on UIManager to get the appropriate UI delegate:

组件的感官可以在任何时候通过使用它的 updateUI()方法来更新为当前的缺省感官，它使用 UIManager 的如下的静态方法来获得合适的 UI 代理：

```
public static ComponentUI getUI(JComponent c)
```

For example, the implementation of updateUI() for the JScrollBar looks like the following:

比如，ScrollBar 的 updateUI()方法的实现如下：

```
public void updateUI() {
    setUI((ScrollBarUI)UIManager.getUI(this));
}
```

And so if a program needs to change the look-and-feel of a GUI hierarchy after it was instantiated, the code might look like the following:

如果一个程序在它实例化之后需要改变它的 GUI 层次，代码可能如下：

```
// GUI already instantiated, where myframe
// is top-level frame
try {
```

```
    UIManager.setLookAndFeel(
      "com.sun.java.swing.plaf.motif.MotifLookAndFeel");
    myframe.setCursor(
          Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR));
    SwingUtilities.updateComponentTreeUI(myframe);
    myframe.validate();

} catch (UnsupportedLookAndFeelException e) {

} finally {
    myframe.setCursor
          (Cursor.getPredefinedCursor
              (Cursor.DEFAULT_CURSOR));
}
```

Managing look-and-feel data 管理感官数据

The UIManager defines a static class, named UIManager.LookAndFeelInfo, for storing the high-level name (such as. "Metal") and particular class name (such as "*com.sun.java.swing.plaf.MetalLookAndFeel*") for a LookAndFeel. It uses these classes internally to manage the known LookAndFeel objects. This information can be accessed from the UIManager via the following static methods:

UIManager 定义了一个静态类，叫做 UIManager.LookAndFeelInfo，用来保存 LookAndFeel 高级别名称（如“Metal”）和特定的类名（如“*com.sun.java.swing.plaf.MetalLookAndFeel*”）。它内部使用这些类来管理它所知道的对象。这个信息能通过下面的静态方法访问：

```
    public static LookAndFeelInfo[]
       getInstalledLookAndFeels()

    public static void
       setInstalledLookAndFeels(LookAndFeelInfo[] infos)
           throws SecurityException

    public static void
       installLookAndFeel(LookAndFeelInfo info)

    public static void
       installLookAndFeel(String name, String className)
```

These methods can be used to programmatically determine which look-and-feel implementations are available, which is useful when building user interfaces which allow the end-user to dynamically select a look-and-feel.

这些方法可以用来自动获知哪些感官实现被提供，这在创建用户接口的时候非常有用，在用户动态选择一套感官的时 候可以得知哪些是允许被选择的。

The look-and-feel packages　　感官包结构

The UI delegate classes provided in swing.plaf (ButtonUI, ScrollBarUI, and so on) define the precise API that a component can use to interact with the UI delegate instance. (**NOTE**: Interfaces were originally used here, but they were replaced with abstract classes because we felt the API was not mature enough to withstand the concrete casting of an interface.) These *plaf* APIs are the root of all look-and-feel implementations.

swing.plaf 提供的 UI 代理类（ButtonUI， ScrollBarUI 等）定义了用于和 UI 代理实例交互的组件的精确 API。（注意：原来这个地方用的是接口，但是后来被抽象类所代替，因为我们觉得那些 API 接口对于具体转形来说不够成熟）。这些 *plaf* APIs 是所有感官实现的根。

Each look-and-feel implementation provides concrete subclasses of these abstract plaf classes. All such classes defined by a particular look-and-feel implementation are contained in a separate package under the swing.plaf package (for example,. swing.plaf.motif, swing.plaf.metal, and so on). A look-and-feel package contains the following:

每个感官实现提供了这些 plat 类的具体子类实现。所有这些在特定的感官实现中定义的类都放在 swing.plaf 包下面的一个单独的包里面（比如， swing.plaf.motif， swing.plaf.metal 等等）。一个感官包包括下面内容：

- The LookAndFeel subclass (for instance, MetalLookAndFeel).

  LookAndFeel 子类（例如，MetalLookAndFeel）。

- All look-and-feel's UI delegate classes (for example, MetalButtonUI, MetalTreeUI, and the like).

  所有的感官代理类（例如，MetalButtonUI，MetalTreeUI 等）。

- Any look-and-feel utility classes (MetalGraphicsUtils, MetalIconFactory, and so on).

  所有的感官工具类（MetalGraphicsUtils，MetalIconFactory 等）。

- Other resources associated with the look-and-feel, such as image files.

  与感官相关的资源，如图标文件等。

In implementing the various Swing look-and-feels, we soon discovered that there was a lot of commonality among them. We factored out this common code into a base look-and-feel implementation (called "*basic*") which extends the plaf abstract classes and from which the specific look-and-feel implementations (*motif*, *windows*, and so on.) extend. The *basic* look-and-feel package supports building a desktop-level look-and-feel, such as Windows or CDE/Motif. The basic look-and-feel package is just one example of how to build a pluggable look-and-feel; the architecture is flexible enough to accommodate other approaches as well.

在实现各种各样的 swing 感官的时候，我们很快发现它们之间有很多共同的地方。我们把这些共同的代码放到一个基 本的感官实现（叫做“basic”）中，它继承了 plaf 抽象类。所有的具体感官实现（*motif*, *windows* 等）都从它继承。*basic*

感官包支持创建桌面级的感官，比如 Windows 或 CDE/Motif。*basic* 感官包只是一个如何创建可拔插感官的样例，架构足够的灵活，也可以支持其它方案。

The remainder of this document will show how a look-and-feel package works at the generic level, leaving the details on the basic package for a future document.

本文的余下部分将会展示感官包在一般级别下如何工作的， basic 包的细节留在将来的一篇文档中描述。

WARNING: All APIs defined below the `swing.plaf` package are *not frozen* in the 1.0.X version of Swing. We are currently cleaning up those APIs for the version of Swing that will ship with JDK1.2beta4, at which time they will become frozen. So if you are developing your own look-and-feel implementation using the 1.0.1 API, this is likely to affect you.

警告：包下面的所有的 APIs 在 swing1.0.X 版本中都没有被冻结，我们现在清除了 JDK1.2beta4 所带的 swing APIs，从那时起它们被冻结。所以如果你使用 1.0.1 API 来开发你自己的感官实现，这将会对你造成影响。

The LookAndFeel Subclass　　感官子类

The `LookAndFeel` class defines the following abstract methods, which all subclasses must implement:

`LookAndFeel` 类定义下面的抽象方法，所有的子类都必须实现它：

```
public String getName();
public String getID();
public String getDescription();
public boolean isNativeLookAndFeel();
public boolean isSupportedLookAndFeel();
```

The `getName()`, `getID()`, and `getDescription()` methods provide generic information about the look-and-feel.

`getName()`， `getID()`,和 `getDescription()` 方法提供了感官的一般信息：

The `isNativeLookAndFeel()` method returns `true` if the look-and-feel is native to the current platform. For example, `MotifLookAndFeel` returns `true` if it is currently running on the Solaris platform, and returns `false` otherwise.

如果是当前平台的本地感官， `isNativeLookAndFeel()` 方法返回 `true`。例如，如果系统当前运行在 Solaris 平台， `MotifLookAndFeel` 返回 `true`，否则返回 `false`。

The `isSupportedLookAndFeel()` method returns whether or not this look-and-feel is authorized to run on the current platform. For example, `WindowsLookAndFeel` returns `true` *only* if it is running on a Windows 95, Windows 98, or Windows NT machine.

`isSupportedLookAndFeel()` 方法返回该感官是否授权在当前平台上运行。比如， `WindowsLookAndFeel` 只当它运行在 Windows 95，Windows 98 或 Windows NT 机器上的时候才返回 `true`。

A LookAndFeel class also provides methods for initialization and uninitialization:

LookAndFeel 类还提供了初始化和反初始化方法：

```
public void initialize()
public void uninitialize()
```

The initialize() method is invoked by the UIManager when the LookAndFeel is made the "default" using the UIManager.setLookAndFeel() method. uninitialize() is invoked by the UIManager when the LookAndFeel is about to be replaced as the default.

在 LookAndFeel 通过使用 UIManager.setLookAndFeel()方法设置为缺省时 initialize()方法被 UIManager 调用。当 LookAndFeel 作为缺省感官被替换时 uninitialize()方法会被 UIManager 调用。

The Defaults Table　缺省值表

Finally, the LookAndFeel class provides a method to return the look-and-feel's implementation of the Defaults Table:

最后，LookAndFeel 类提供了一个方法返回缺省值表的感官实现：

```
public UIDefaults getDefaults()
```

The Defaults Table is represented by the UIDefaults class, a direct extension of java.util.Hashtable, which adds methods for accessing specific types of information about a look-and-feel. This table must include all the UIClassID-to-classname mapping information, as well as any default values for presentation-related properties (such as color, font, border, and icon) for each UI delegate. For example, following is a sample of what a fragment of getDefaults() might look like for a hypothetical look-and-feel in a package called "mine":

缺省值表通过 UIDefaults 类来表现，UIDefaults 类是 java.util.Hashtable 的一个直接扩展，它增加了访问感官具体类型信息方法。这个表必须包含所有的 UIClassID-to-classname 映射信息和每个 UI 代理所有的与表现相关的属性的缺省值（比如颜色，字体，边框和图标）。比如下面是一个假设名为“mine”感官包中的可能的 getDefaults()代码片断:

```
public UIDefaults getDefaults()  {
   UIDefaults table = new UIDefaults();
   Object[] uiDefaults = {

     "ButtonUI",    "mine.MyButtonUI",
     "CheckBoxUI",  "mine.MyCheckBoxUI",
     "MenuBarUI",   "mine.MyMenuBarUI",
         ...

     "Button.background",
         new ColorUIResource(Color.gray),
```

```
        "Button.foreground",
            new ColorUIResource(Color.black),
        "Button.font",
            new FontUIResource("Dialog", Font.PLAIN, 12),
        "CheckBox.background",
            new ColorUIResource(Color.lightGray),
        "CheckBox.font",
            new FontUIResource("Dialog", Font.BOLD, 12),

            ...
    }
    table.putDefaults(uiDefaults);
    return table;

}
```

When the default look-and-feel is set with UIManager.setLookAndFeel(), the UIManager calls getDefaults() on the new LookAndFeel instance and stores the hash table it returns. Subsequent calls to the UIManager's lookup methods will be applied to this table. For example, after making " mine " the default Look-and-Feel:

当缺省感官通过 UIManager.setLookAndFeel()设置的时候，UIManager 在新的 LookAndFeel 实例中调用 getDefaults()，并保存它返回的哈希表。在后来调用 UIManager 的查找方法时将会应用到这个表，比如，在使得“mine”为缺省感官之后：

```
UIManager.get("ButtonUI") => "mine.MyButtonUI"
```

The UI classes access their default information in the same way. For example, our example ButtonUI class would initialize the JButton's "background" property like this:

UI 类访问它们的缺省信息的方式都一样，比如，我们的例子类 ButtonUI 会初始化 JButton 的“background”属性如下：

```
button.setBackground(
    UIManager.getColor("Button.background");
```

The defaults are organized this way to allow developers to override them. More detail about Swing's Defaults mechanism will be published in a future article.

缺省值通过这种方式组织起来以便开发人员重置它。关于 Swing 缺省值机制的更多细节会在将来的文档中描述。

Distinguishing between UI-set and app-set properties

UI 设置属性与应用设置属性之间的区别

Swing allows applications to set property values (such as color and font) individually on components. So it's critical to make sure that these values don't get clobbered when a look-and-feel sets up its "default" properties for the component.

Swing 允许应用程序在组件上单独设置属性值（如颜色和字体）。所以必须保证这些值不会把感官给组件设置的缺省 值属性弄的一团糟。

This is not an issue the *first time* a UI delegate is installed on a component (at construct time) because all properties will be uninitialized and legally settable by the look-and-feel. The problem occurs when the application sets individual properties *after* component construction and then subsequently sets a *new* look-and-feel (that is, *dynamic* look-and-feel switching). This means that the look-and-feel must be able to distinguish between property values set by the application, and those set by a look-and-feel.

这在 UI 代理第一次安装在组件上的时候（在构造的时候）没有任何问题，因为所有的属性都会被反初始化和被感官合法的设置。问题出现在当应用程序在组件构造之后设置单独的属性，并且随后设置了一个新的感官（即感官动态转换）。这意味着感官必须能够区分哪些是应用程序设置的属性值，哪些是感官设置的属性值。

This issue is handled by marking all values set by the look-and-feel with the `plaf.UIResource` interface. The `plaf` package provides a set of "marked" classes for representing these values, `ColorUIResource`, `FontUIResource`, and `BorderUIResource`. The preceding code example shows the usage of these classes to mark the default property values for the hypothetical `MyButtonUI` class.

这个问题是通过感官使用 plaf.UIResource 接口对所有的这些值做标记来解决的。plaf 包提供了一系列“标记了的”类来表现这些值，ColorUIResource，FontUIResource 和 BorderUIResource。前面的代码例子中展示了这些类为我们假设的 MyButtonUI 类的缺省值做标记的用法。

The UI delegate　UI 代理

The superclass of all UI Delegate classes is `swing.plaf.ComponentUI`. This class contains the primary "machinery" for making the pluggable look-and-feel work. Its methods deal with UI installation and uninstallation, and with delegation of a component's geometry-handling and painting.

所有的 UI 代理类的父类都是 swing.plaf.ComponentUI。这个类包含了使可拔插感官工作的最主要的“机制”。它的方法主要处理 UI 的安装和卸载，代理组件的几何学处理和绘制。

Many of the UI Delegate subclasses also provide additional methods specific to their own required interaction with the component; however, this document focuses primarily on the generic mechanism implemented by `ComponentUI`.

很多 UI 代理子类还根据它们自己和组件的交互需要提供了额外的方法。不过，本文档的焦点主要是 ComponentUI 实现的一般机制。

UI installation and deinstallation　UI 的安装和卸载

First off, the `ComponentUI` class defines these methods methods for UI delegate installation and uninstallation:

马上，ComponentUI 类为 UI 代理定义了这些安装和卸载的方法。

```
public void installUI(JComponent c)
public void uninstallUI(JComponent c)
```

Looking at the implementation of JComponent.setUI() (which is always invoked from the setUI method on JComponent subclasses), we can clearly see how UI delegate installation/de-installation works:

看看 JComponent.setUI()的实现（它总是被 JComponent 子类的 setUI 方法调用），我们可以清晰的看到 UI 代理是如何安装/卸载的：

```
protected void setUI(ComponentUI newUI) {
    if (ui != null) {
        ui.uninstallUI(this);
    }
    ComponentUI oldUI = ui;
    ui = newUI;
    if (ui != null) {
        ui.installUI(this);
    }
    invalidate();
    firePropertyChange("UI", oldUI, newUI);
}
```

UI installation illustrated　　UI 安装插图

This article comes with a giant poster-size chart that illustrates the process installing a UI delegate. It can provide you with a valuable overview of the delegate-installation process.

本文开头有一张巨大的像海报一样大小的图，描述的是安装 UI 代理的过程。他可以提供给你一些关于代理安装过程的有用信息。

The UI delegate's installUI() method is responsible for the following:

UI 代理的 installUI()方法负责下面的事情：

- Set default font, color, border, and opacity properties on the component.

  设置组件的缺省字体，颜色，边框和不透明属性。

- Install an appropriate layout manager on the component.

  在组件上安装相应的布局管理。

- Add any appropriate child subcomponents to the component

在组件上添加相应的子组件。

- Register any required event listeners on the component.

  在组件上注册所有需要的事件监听器。

- Register any look-and-feel-specific keyboard actions (mnemonics, etc.)for the component.

  在组件上注册所有的感官具体键盘事件（助记符等）。

- Register appropriate model listeners to be notified when to repaint.

  注册相应的在重绘的时候会被通知的模型监听器。

- Initialize any appropriate instance data.

  初始化所有相应的实例数据。

For example, the `installUI()` method for an extension of ButtonUI might look like this:

举例来说，一个 ButtonUI 的扩展类的 `installUI()` 方法可能如下：

```
protected MyMouseListener mouseListener;
protected MyChangeListener changeListener;

public void installUI(JComponent c) {
   AbstractButton b = (AbstractButton)c;

   // Install default colors & opacity
   Color bg = c.getBackground();
   if (bg == null || bg instanceof UIResource) {
      c.setBackground(
          UIManager.getColor("Button.background"));
   }
   Color fg = c.getForeground();
   if (fg == null || fg instanceof UIResource) {
      c.setForeground(
          UIManager.getColor("Button.foreground"));
   }
   c.setOpaque(false);

            // Install listeners
   mouseListener = new MyMouseListener();
   c.addMouseListener(mouseListener);
   c.addMouseMotionListener(mouseListener);
   changeListener = new MyChangeListener();
```

```
    b.addChangeListener(changeListener);
}
```

Conventions for initializing component properties

组件属性初始化的约定

Swing defines a number of conventions for initializing component properties at install-time, including the following:

Swing 在安装的期间定义了很多组件属性初始化约定，包括如下：

1. All values used for setting colors, font, and border properties should be obtained from the Defaults table (as described in the subsection on the LookAndFeel subclass).

   所有用于设置颜色，字体和边框属性的值都应该从缺省值表中获取（在 LookAndFeel subclass 小节中有描述）。

2. Color, font and border properties should be set if -- and *only* if -- the application has not already set them.

   颜色，字体和边框属性应该且只应该在应用还没有设置它们之前进行设置。

To facilitate convention No 1, the `UIManager` class provides a number of static methods to extract property values of a particular type (for instance, the static methods `UIManager.getColor()`, `UIManager.getFont()`, and so on).

为了方便第一个约定， UIManager 类提供了很多静态方法来提取具体类型的属性值（比如，静态方法 UIManager.getColor()，UIManager.getFont()等）。

Convention No. 2 is implemented by always checking for either a `null` value or an instance of `UIResource` before setting the property.

第二个约定通过在设置属性之前一直检查是否是空值或 UIResource 实例来实现。

The `ComponentUI`'s `uninstall()` method must carefully undo everything that was done in the `installUI()` method so that the component is left in a pristine state for the next UI delegate. The `uninstall()` method is responsible for:

ComponentUI 的 uninstall()方法必须小心恢复被 installUI()方法做过的所有事情，这样组件就可以为下一个 UI 代理提供纯洁质朴的状态。uninstall()方法负责：

- Clearing the border property if it has been set by installUI().

  如果边框属性已被 installUI()方法设置了，将其清除。

- Remove the layout manager if it had been set by installUI().

  如果布局管理已被 installUI()方法设置了，将其移除。

- Remove any subcomponents added by installUI().

  移除所有 installUI()方法添加的子组件。

- Remove any event/model listeners that were added by installUI().

  移除所有 installUI()方法添加的事件/模型监听器。

- Remove any look-and-feel-specific keyboard actions that were installed by installUI().

  移除所有 installUI()方法添加的感官具体键盘事件。

- Nullify any initialized instance data (to allow GC to clean up).

  使所有初始化的实例数据无效（使 GC 可以回收）。

For example, an `uninstall()` method to undo what we did in the above example installation might look like this:

例如，`uninstall()`方法恢复我们在上面的安装例子中所做的事情可能如下：

```
public void uninstallUI(JComponent c) {
   AbstractButton b = (AbstractButton)c;


   // Uninstall listeners
   c.removeMouseListener(mouseListener);
   c.removeMouseMotionListener(mouseListener);
   mouseListener = null;
   b.removeChangeListener(changeListener);
   changeListener = null;
}
```

Defining geometry 几何学定义

In the AWT (and thus in Swing) a container's LayoutManager will layout the child components according to its defined algorithm; this is known as "validation" of a containment hierarchy. Typically LayoutManagers will query the child components' preferredSize property (and sometimes minimumSize and/or maximumSize as well, depending on the algorithm) in order to determine precisely how to position and size those children.

在 AWT 中（swing 中也一样），容器的布局管理器会根据它定义的算法来布局它的子组件；这被称为包含层次结构“确认”。代表性的来说，布局管理器会查询子组件的 preferredSize 属性（有时是 minimumSize，或者 maximumSize，取决于算法）来精确的判断它的子组件的大小和位置。

Obviously, these geometry properties are something that a look-and-feel usually needs to define for a given component, so ComponentUI provides the following methods for this purpose:

很明显，这些几何属性是某种感官通常需要为所给的组件定义的东西。所以为这个目的 ComponentUI 提供了下面的方法：

```
public Dimension
   getPreferredSize(JComponent c)
public Dimension
   getMinimumSize(JComponent c)
public Dimension
   getMaximumSize(JComponent c)
public boolean
   contains(JComponent c, int x, int y)
```

JComponent's parallel methods (which are invoked by the LayoutManager during validation) then simply delegate to the UI object's geometry methods if the geometry property was not explicitly set by the program. Below is the implementation of JComponent.getPreferredSize() which shows this delegation:

如果几何属性没有显式的被程序设置，JComponent 相应的方法（它们通过确认被 LayoutManager 调用）会简单的代理 UI 对象的几何方法：

```
public Dimension getPreferredSize() {
   if (preferredSize != null) {
      return preferredSize;
   }
   Dimension size = null;
   if (ui != null) {
      size = ui.getPreferredSize(this);
   }
   return (size != null) ? size :
         super.getPreferredSize();
}
```

Even though the bounding box for all components is a Rectangle, it's possible to simulate a non-rectangular component by overriding the implementation of the contains() method from java.awt.Component. (This method is used for the hit-testing of mouse events). But, like the other geometry properties in Swing, the UI delegate defines its own version of the contains() method, which is also delegated to by JComponent.contains():

虽然所有的组件的边框都是矩形，但是通过在实现类中重置 java.awt.Component 的 contains()方法模拟一个非矩形组件也是可行的。（这个方法用来给鼠标事件做点击测试）。但是，和 swing 的其它几何属性一样，UI 代理定义了它自己的 contains()方法版本，它也是被 JComponent.contains()代理。

```
public boolean contains(JComponent c, int x, int y) {
   return (ui != null) ?
      ui.contains(this, x, y) :
            super.contains(x, y);
}
```

So a UI delegate could provide non-rectangular "feel" by defining a particular implementation of contains() (for example, if we wanted our MyButtonUI class to implement a button with rounded corners).

所以 UI 代理可以通过定义特定的 contains()实现提供非矩形的“感觉”（例如，如果我们想要我们的 MyButtonUI 类实现一个圆角的按钮）。

Painting　　绘制

Finally, the UI delegate must paint the component appropriately, hence ComponentUI has the following methods:

最后，UI 代理必须绘制相应的组件，所以 ComponentUI 有下面的方法：

```
public void paint(Graphics g, JComponent c)
public void update(Graphics g, JComponent c)
```

And once again, JComponent.paintComponent() takes care to delegate the painting:

再重复一次，JComponent.paintComponent()负责代理绘制：

```
protected void paintComponent(Graphics g) {
   if (ui != null) {
      Graphics scratchGraphics =
         SwingGraphics.createSwingGraphics(g.create());
      try {
         ui.update(scratchGraphics, this);
      }
      finally {
         scratchGraphics.dispose();
      }
   }
}
```

Similarly to the way in which things are done in AWT, the UI delegate's update() method clears the background (if opaque) and then invokes its paint() method, which is ultimately responsible for rendering the contents of the component.

和 AWT 一样，UI 代理的 update()方法清除背景（如果是透明的），然后调用 paint()方法，它主要负责组件内容的渲染。

Stateless vs. stateful delegates　无状态的代理与有状态的代理

All the methods on ComponentUI take a JComponent object as a parameter. This convention enables a stateless implementation of a UI delegate (because the delegate can always query back to the specified component instance for state information). Stateless UI delegate implementations allow a single UI delegate instance to be used for all instances of that component class, which can significantly reduce the number of objects instantiated.

ComponentUI 所有的方法都是通过参数来获得 JComponent 对象，这个约定使得 UI 代理可以做无状态的实现（因为代理一直可以回过头来查询具体的组件实例来获得状态信息）。无状态 UI 代理实现使得组件所有的实例都可以使用同一个独立的 UI 代理，这可以有效的减少实例化的对象的数量。

This approach works well for many of the simpler GUI components. But for more complex components, we found it not to be a "win" because the inefficiency created by constant state recalculations was worse than creating extra objects (especially since the number of complex GUI components created in a given program tends to be small).

这个方案对于很多简单的 GUI 组件来说工作的很好。但是对于更复杂的组件，我们发现它不能胜任，因为持续的状态判断带来的低效率比创建额外的对象要糟糕（特别是在给定的程序中复杂 GUI 组件的数量比较少的时候）。

The ComponentUI class defines a static method for returning a delegate instance:

ComponentUI 类定义了一个静态方法来返回代理实例：

```
public static ComponentUI
      createUI(JComponent c)
```

It's the implementation of this method that determines whether the delegate is stateless or stateful. That's because the UIManager.getUI() method invoked by the component to create the UI delegate internally invokes this createUI() method on the delegate class to get the instance.

这个方法的实现决定了代理是有状态的还是无状态的。因为组件通过调用 UIManager.getUI()方法来创建 UI 代理对象，它通过内部调用代理类的 createUI()方法来获得实例。

The Swing look-and-feel implementations use both types of delegates. For example, Swing's BasicButtonUI class implements a stateless delegate:

两种类型的代理 swing 感官都有使用。例如，swing 的 BasicButtonUI 类就实现了一个无状态的代理：

```
// Shared UI object
protected static ButtonUI buttonUI;

public static ComponentUI createUI(JComponent c)
   if(buttonUI == null) {
```

```
        buttonUI = new BasicButtonUI();
    }
    return buttonUI;
}
```

While Swing's BasicTabbedPaneUI uses the stateful approach:

然而 swing 的 BasicTabbedPaneUI 使用了有状态的方案：

```
public static ComponentUI createUI(JComponent c)
   return new BasicTabbedPaneUI();
}
```

Pluggable Look-and-Feel summary　可拔插感官总结

The pluggable look-and-feel feature of Swing is both powerful and complex (which you understand if you've gotten this far!). It is designed to be programmed by a small subset of developers who have a particular need to develop a new look-and-feel implementation. In general, application developers only need to understand the capabilities of this mechanism in order to decide *how* they wish to support look-and-feels (such as whether to lock-down the program to a single look-and-feel or support look-and-feel configuration by the user). Swing's UIManager provides the API for applications to manage the look-and-feel at this level.

Swing 的可拔插感官特性复杂（如果你搞懂了就不会）但非常强大。它是针对一小群有开发新感官实现需要的开发人 员的编程而设计的。一般来说，应用开发人员只需要明白这个机制的能力，以便决定他们希望如何支持感官（比如是给 程序提供固定的单一感官还是让用户来配置感官）。Swing 的 UIManager 为应用提供了这个级别的 API 来管理感官。

If you're one of those developers who needs (or wants) to develop a custom look-and-feel, it's critical to understand these underpinnings before you write a single line of code. We're working on providing better documentation to help with this process -- starting with this document, and continuing with others that will follow soon.

如果你是这些需要（或想要）开发自定义感官的开发人员中的一个，不建议在没有理解这些基础之前就开始写代码，我们会提供更好的文档来帮助这个过程－从这篇文档起步，在很快就会来到的其它文档中继续前行。