## Связь с архитектурой MVC

В общем, визуальный компонент определяется тремя отдельными составляющими.

- Внешний вид компонента при воспроизведении на экране.
- Взаимодействие с пользователем.
- Сведения о состоянии компонента.

Независимо от того, какая именно архитектура используется для реализации компонента, она должна неявно включать в себя эти три его составляющие. В течение многих лет свою исключительную эффективность доказала архитектура MVC — "модель–представление–контроллер".

Успех архитектуры MVC объясняется тем, что каждая часть этой архитектуры соответствует отдельной составляющей компонента. Согласно терминологии MVC, *модель* соответствует сведениям о состоянии компонента. Так, для флажка модель содержит поле, которое показывает, установлен ли флажок. *Представление* определяет порядок отображения компонента на экране, включая любые составляющие представления, зависящие от текущего состояния модели. *Контроллер* определяет порядок реагирования компонента на действия пользователя. Так, если пользователь щелкает на флажке, контроллер реагирует на это действие, изменяя модель, чтобы отразить выбор пользователя (установку или сброс флажка). В итоге представление обновляется. Разделяя компонент на модель, представление и контроллер, можно изменять конкретную реализацию любой из этих составляющих архитектуры MVC, не затрагивая остальные. Например, различные реализации представлений могут воспроизводить один и тот же компонент разными способами, но это не будет оказывать никакого влияния на модель или контроллер.

Хотя архитектура MVC и положенные в ее основу принципы выглядят вполне благоразумно, высокая степень разделения представления и контроллера не дает никаких преимуществ компонентам библиотеки Swing. Поэтому в библиотеке Swing применяется видоизмененный вариант архитектуры MVC, где представление и контроллер объединены в один логический объект, называемый *представителем пользовательского интерфейса*. В связи с этим подход, применяемый в библиотеке Swing, называется архитектурой *"модель–представитель"*, или архитектурой *"разделяемая модель"*. Таким образом, в архитектуре компонентов библиотеки Swing не используется классическая реализация архитектуры MVC, несмотря на то, что первая основывается на последней.

Подключаемый стиль оформления стал возможным в библиотеке Swing благодаря архитектуре "модель–представитель". Представление и контроллер отделены от модели, поэтому стиль оформления можно изменять, не оказывая влияния на то, как компонент применяется в программе. И наоборот, модель можно настроить, не оказывая влияния на то, как компонент отображается на экране или реагирует на действия пользователя.

Для поддержания архитектуры "модель–представитель" большинство компонентов библиотеки Swing содержит два объекта. Один из них представляет модель, а другой — представитель пользовательского интерфейса. Модели определяются в интерфейсах. Например, модель кнопки определяется в интерфейсе `ButtonModel`. А пред-

ставители пользовательского интерфейса являются классами, наследующими от класса `ComponentUI`. Например, представителем пользовательского интерфейса кнопки является класс `ButtonUI`. Как правило, прикладные программы не взаимодействуют непосредственно с представителями пользовательского интерфейса.

# Компоненты и контейнеры

ГПИ, создаваемый средствами Swing, состоит из двух основных элементов: *компонентов* и *контейнеров*. Но это, по существу, концептуальное разделение, поскольку все контейнеры также являются компонентами. Отличие этих двух элементов заключается в их назначении: компонент является независимым визуальным элементом управления вроде кнопки или ползунка, а контейнер содержит группу компонентов. Таким образом, контейнер является особым типом компонента и предназначен для хранения других компонентов. Более того, компонент должен находиться в контейнере, чтобы его можно было отобразить. Так, во всех ГПИ, создаваемых средствами Swing, имеется как минимум один контейнер. А поскольку контейнеры являются компонентами, то один контейнер может содержать другие контейнеры. Благодаря этому в библиотеке Swing можно определить *иерархию вложенности*, на вершине которой должен находиться *контейнер верхнего уровня*. А теперь рассмотрим подробнее компоненты и контейнеры.

## Компоненты

В общем, компоненты библиотеки Swing происходят от класса `JComponent`. (Исключением из этого правила являются четыре контейнера верхнего уровня, о которых речь пойдет в следующем разделе.) В классе `JComponent` предоставляются функциональные возможности, общие для всех компонентов. Так, в классе `JComponent` поддерживается подключаемый стиль оформления. Класс `JComponent` наследует классы `Container` и `Component` из библиотеки AWT. Следовательно, компонент библиотеки Swing построен на основе компонента библиотеки AWT и совместим с ним.

Все компоненты Swing представлены классами, определенными в пакете `javax.swing`. Ниже поименно перечислены классы компонентов Swing (включая компоненты, используемые в качестве контейнеров).

| | | | |
|---|---|---|---|
| **JApplet** | **JButton** | **JCheckBox** | **JCheckBoxMenuItem** |
| **JColorChooser** | **JComboBox** | **JComponent** | **JDesktopPane** |
| **JDialog** | **JEditorPane** | **JFileChooser** | **JFormattedTextField** |
| **JFrame** | **JInternalFrame** | **JLabel** | **JLayer** |
| **JLayeredPane** | **JList** | **JMenu** | **JMenuBar** |
| **JMenuItem** | **JOptionPane** | **JPanel** | **JPasswordField** |
| **JPopupMenu** | **JProgressBar** | **JRadioButton** | **JRadioButtonMenuItem** |
| **JRootPane** | **JScrollBar** | **JScrollPane** | **JSeparator** |
| **JSlider** | **JSpinner** | **JSplitPane** | **JTabbedPane** |
| **JTable** | **JTextArea** | **JTextField** | **JTextPane** |
| **JTogglebutton** | **JToolBar** | **JToolTip** | **JTree** |
| **JViewport** | **JWindow** | | |

Обратите внимание на то, что все классы компонентов начинаются с буквы J. Например, класс для создания метки называется `JLabel`, класс для создания кнопки — `JButton`, а класс для создания ползунка — `JScrollBar`.

## Контейнеры

В библиотеке Swing определены два типа контейнеров. К первому типу относятся контейнеры верхнего уровня, представленные классами `JFrame`, `JApplet`, `JWindow` и `JDialog`. Классы этих контейнеров не наследуют от класса `JComponent`, но они наследуют от классов `Component` и `Container` из библиотеки AWT. В отличие от остальных компонентов Swing, которые являются легковесными, компоненты верхнего уровня являются тяжеловесными. Поэтому в библиотеке Swing контейнеры являются особым случаем компонентов.

Судя по названию, контейнер верхнего уровня должен находиться на вершине иерархии контейнеров. Контейнер верхнего уровня не содержится ни в одном из других контейнеров. Более того, каждая иерархия вложенности должна начинаться с контейнера верхнего уровня. Таким контейнером в прикладных программах чаще всего является класс `JFrame`, а в аплетах —класс `JApplet`.

Ко второму типу контейнеров, поддерживаемых в библиотеке Swing, относятся легковесные контейнеры. Они наследуют от класса `JComponent`. Примером легковесного контейнера служит класс `JPanel`, который представляет контейнер общего назначения. Легковесные контейнеры нередко применяются для организации и управления группами связанных вместе компонентов, поскольку легковесный контейнер может находиться в другом контейнере. Следовательно, легковесные контейнеры вроде класса `JPanel` можно применять для создания подгрупп связанных вместе элементов управления, содержащихся во внешнем контейнере.

## Панели контейнеров верхнего уровня

Каждый контейнер верхнего уровня определяет ряд *панелей*. На вершине иерархии панелей находится корневая панель в виде экземпляра класса `JRootPane`. Класс `JRootPane` представляет легковесный контейнер, предназначенный для управления остальными панелями. Он также помогает управлять дополнительной, хотя и не обязательной строкой меню. Панели, составляющие корневую панель, называются *прозрачной панелью*, *панелью содержимого* и *многослойной панелью* соответственно.

Прозрачная панель является панелью верхнего уровня. Она находится над всеми панелями и покрывает их полностью. По умолчанию эта панель представлена прозрачным экземпляром класса `JPanel`. Прозрачная панель позволяет управлять событиями от мыши, оказывающими влияние на весь контейнер в целом, а не на отдельный элемент управления, или, например, рисовать поверх любого другого компонента. Как правило, обращаться к прозрачной панели непосредственно не требуется, но если она все же понадобится, то ее нетрудно обнаружить там, где она обычно находится.

Многослойная панель представлена экземпляром класса `JLayeredPane`. Она позволяет задать определенную глубину размещения компонентов. Глубина определяет степень перекрытия компонентов. (В связи с этим многослойные панели

позволяют задавать упорядоченность компонентов по координате Z, хотя это требуется не всегда.) На многослойной панели находится панель содержимого и дополнительно, хотя и не обязательно, —строка меню.

Несмотря на то что прозрачная и многослойная панели являются неотъемлемыми частями контейнера верхнего уровня и служат для разных целей, бо́льшая часть их возможностей скрыта от пользователей. Прикладная программа чаще всего будет обращаться к панели содержимого, поскольку именно на ней обычно располагаются визуальные компоненты. Иными словами, когда компонент (например, кнопка) вводится в контейнер верхнего уровня, он оказывается на панели содержимого. По умолчанию панель содержимого представлена непрозрачным экземпляром класса `JPanel`.

## Пакеты библиотеки Swing

Библиотека Swing — это довольно крупная подсистема, в которой задействовано большое количество пакетов. Ниже перечислены пакеты, определенные в библиотеке Swing на момент написания данной книги.

| | | |
|---|---|---|
| `javax.swing` | `javax.swing.plaf.basic` | `javax.swing.text` |
| `javax.swing.border` | `javax.swing.plaf.metal` | `javax.swing.text.html` |
| `javax.swing.colorchooser` | `javax.swing.plaf.multi` | `javax.swing.text.html.parser` |
| `javax.swing.event` | `javax.swing.plaf.nimbus` | `javax.swing.text.rtf` |
| `javax.swing.filechooser` | `javax.swing.plaf.synth` | `javax.swing.tree` |
| `javax.swing.plaf` | `javax.swing.table` | `javax.swing.undo` |

Самым главным среди них является пакет `javax.swing`. Его следует импортировать в любую прикладную программу, пользующуюся библиотекой Swing. В этом пакете содержатся классы, реализующие базовые компоненты Swing, в том числе кнопки, метки и флажки.

## Простое Swing-приложение

Swing-приложения отличаются от консольных программ и AWT-приложений, демонстрировавшихся ранее в данной книге. Например, в них используются компоненты и иерархии контейнеров не из библиотеки AWT. Кроме того, Swing-приложения предъявляют особые требования, связанные с многопоточной обработкой. Уяснить структуру Swing-приложения лучше всего на конкретном примере. Имеются две разновидности программ на Java, в которых обычно применяется библиотека Swing: настольные приложения и аплеты. В этом разделе будет рассмотрен пример создания Swing-приложения. А создание Swing-аплета обсуждается далее этой главе.

Несмотря на всю краткость рассматриваемого здесь примера программы, он наглядно демонстрирует один из способов разработки Swing-приложения, а также основные средства библиотеки Swing. В данном примере используются два ком-

понента Swing: JFrame и JLabel. Класс JFrame представляет контейнер верхнего уровня, который обычно применяется в Swing-приложениях, а класс JLabel — компонент Swing, создающий метку для отображения информации. Метка является самым простым компонентом Swing, поскольку это пассивный компонент. Это означает, что метка не реагирует на действия пользователя. Она служит лишь для отображения выводимых данных. В данном примере контейнер типа JFrame служит для хранения метки в виде экземпляра класса JLabel. Метка отображает короткое текстовое сообщение.

```
// Пример простого Swing-приложения

import javax.swing.*;

class SwingDemo {

    SwingDemo() {

        // создать новый контейнер типа JFrame
        JFrame jfrm = new JFrame("A Simple Swing Application");
        // Простое Swing-приложение

        // задать исходные размеры фрейма
        jfrm.setSize(275, 100);

        // завершить работу, если пользователь закрывает приложение
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // создать метку с текстом сообщения
        JLabel jlab = new JLabel("Swing means powerful GUIs.");
        // Swing — это мощные ГПИ

        // ввести метку на панели содержимого
        jfrm.add(jlab);

        // отобразить фрейм
        jfrm.setVisible(true);
      }

      public static void main(String args[]) {
        // создать фрейм в потоке диспетчеризации событий
        SwingUtilities.invokeLater(new Runnable() {
          public void run() {
            new SwingDemo();
          }
      });
    }
}
```

Swing-приложения компилируются и выполняются таким же образом, как и остальные приложения Java. Поэтому, чтобы скомпилировать данное Swing-приложение, нужно ввести в командной строке следующую команду:

```
javac SwingDemo.java
```

А для того чтобы запустить Swing-приложение на выполнение, нужно ввести в командной строке такую команду:

```
java SwingDemo
```

Когда Swing-приложение начнет выполняться, в нем будет создано окно, показанное на рис. 31.1.

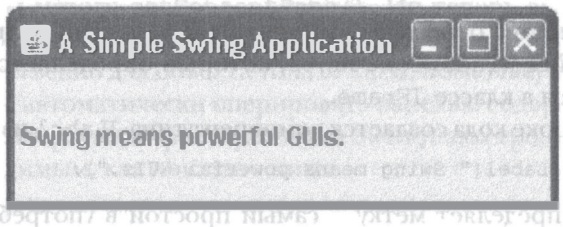


**Рис. 31.1.** Окно, создаваемое приложением `SwingDemo`

Пример приложения `SwingDemo` демонстрирует ряд основных понятий библиотеки Swing, поэтому рассмотрим этот пример построчно. Данное Swing-приложение начинается с импорта пакета `javax.swing`. Как упоминалось ранее, этот пакет содержит компоненты и модели, определяемые в библиотеке Swing. Так, в пакете `javax.swing` определяются классы, реализующие метки, кнопки, текстовые элементы управления и меню. Поэтому этот пакет обычно включается во все программы, пользующиеся библиотекой Swing.

Затем объявляются класс `SwingDemo` и его конструктор, в котором выполняется большинство действий данной программы. Сначала в нем создается экземпляр класса `JFrame`, как показано ниже.

```
JFrame jfrm = new JFrame("A Simple Swing Application");
```

В методе `setSize()`, наследуемом классом `JFrame` от класса `Component` из библиотеки AWT, задаются размеры окна в пикселях. Ниже приведена общая форма этого метода. В данном примере задается *ширина* окна **275** пикселей, а *высота* — **100** пикселей.

```
void setSize(int ширина, int высота)
```

Когда закрывается окно верхнего уровня (например, после того, как пользователь щелкнет на кнопке закрытия), по умолчанию окно удаляется с экрана, но работа приложения не прекращается. И хотя такое стандартное поведение иногда оказывается полезным, для большинства приложений оно не подходит. Чаще всего при закрытии окна верхнего уровня требуется завершить работу всего приложения. Это можно сделать двумя способами. Самый простой из них состоит в том, чтобы вызвать метод `setDefaultCloseOperation()`, что и делается в данном приложении следующим образом:

```
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

В результате вызова этого метода приложение полностью завершает свою работу при закрытии окна. Общая форма метода `setDefaultCloseOperation()` выглядит следующим образом:

```
void setDefaultCloseOperation(int что)
```

Значение константы, передаваемое в качестве параметра *что* определяет, что именно происходит при закрытии окна. Помимо значения константы `JFrame.EXIT_ON_CLOSE`, имеются также следующие значения: