11

В этой главе:
- Управление видимостью компонента
- Компоненты навигации
- Создание приложения с фотоальбомом

Управление расположением и видимостью компонентов

При работе с Flex под рукой у разработчика набор прекрасных инструментов, позволяющих создавать интерфейс приложения, скорее напоминающий интерфейс традиционных программ, чем приложений для веб. Одной из самых полезных функций Flex является возможность управления отображением отдельных частей приложения. Ее достоинства очевидны при работе с приложением, состоящим из нескольких разделов, когда необходимо скрыть часть из них, чтобы одновременно могли отображаться только некоторые разделы. С помощью стандартных элементов управления Flex приложение можно поделить на несколько разных видов и даже дать пользователям возможность настройки желаемого отображения определенных разделов.

Управление видимостью компонента

Видимость любого визуального компонента Flex можно регулировать с помощью свойства `visible`, по умолчанию принимающего значение `true`. Это относится не только к визуальным элементам управления, но и к контейнерам. Вы уже знаете, что последние удобно использовать не только для выравнивания и организации расположения элементов приложения; это незаменимые инструменты при создании структуры его различных частей. Поскольку контейнеры являются визуальными компонентами, они наследуют свойства, регулирующие видимость. Это означает, что настройки видимости контейнера распространяются и на его дочерние элементы.

При присваивании значения `false` его свойству `visible` контейнер становится невидимым, однако тем не менее занимает отведенное ему ме-

сто. В качестве примера представим, что в приложении имеются три кнопки, расположенные по горизонтали с помощью контейнера HBox. Если средняя кнопка станет невидимой, крайние кнопки будут находиться на том же расстоянии друг от друга, как будто невидимая кнопка все еще присутствует. Чтобы такого пустого пространства не возникало, можно присвоить значение `false` свойству `includeInLayout`. Обратите внимание на последовательность рисунков 11.1–11.4, которые наглядно демонстрируют эти возможности.

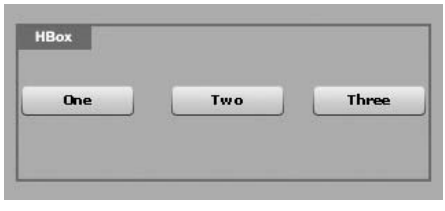

Рис. 11.1. *Три кнопки внутри контейнера HBox, все видимы*

Рис. 11.2. *Три кнопки внутри контейнера HBox; вторая кнопка невидима (visible="false")*

Рис. 11.3. *Три кнопки внутри контейнера HBox; для второй кнопки установлены свойства includeInLayout="false", visible="true" (но третья кнопка ее перекрывает)*

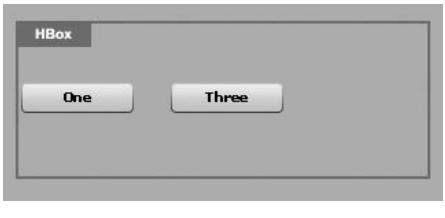

Рис. 11.4. *Три кнопки внутри контейнера HBox; для второй кнопки установлены свойства includeInLayout="false", visible="false" (третья кнопка занимает ее место)*

Компоненты навигации

Свободного пространства для размещения всех необходимых элементов вашего приложения иногда может оказаться недостаточно. Скорее всего вам придется определить, какие из компонентов будут видимы одновременно, или создать возможность выбора между несколькими видами внутри конкретного приложения. Вспомните диалоговое окно настройки предпочтений большинства традиционных программ, окна настройки операционной системы или панель управления. Такого рода диалоговые окна предоставляют широкий набор настроек, которые не обязательно должны быть отображены одновременно. Обычно они сгруппированы по разделам, переход по которым осуществляется, как

правило, с помощью вкладок, что напоминает организацию файлов в картотеке или страницы в записной книжке с разделителями

В стандартный набор компонентов Flex входят компоненты, которые позволяют разработчику легко управлять текущим отображением приложения и создавать различные виды из видимых элементов. Они называются *контейнерами навигации* и предназначены для организации переключения между дочерними элементами. В отличие от контейнеров, предлагающих горизонтальное, вертикальное или тому подобное расположение вложенных элементов, контейнеры навигации располагают дочерние элементы таким образом, что только один из них может быть отображен, при этом остальные элементы становятся невидимыми. Внешне переключение между различными видами чаще всего осуществляется с помощью вкладок.

Мы начнем разговор об элементах навигации с рассмотрения возможностей наиболее часто используемого при построении интерфейса приложения контейнера TabNavigator. Он может включать в себя сколько угодно вложенных элементов, снабжая каждый из них соответствующей вкладкой. В качестве дочерних элементов могут выступать только контейнеры (что справедливо и для остальных контейнеров навигации), поскольку контейнеры предназначены для объединения элементов в единое целое. Каждый вложенный контейнер должен обладать свойством `label`, значение которого будет отображаться на соответствующей ему вкладке. Для наименования вкладок не требуется создавать массив текстовых элементов и т. п., достаточно задать свойство `label` каждому дочернему контейнеру. Рассмотрим следующий код, представляющий собой способ организации переключения между контейнерами Canvas с различным цветом фона при помощи вкладок. Результат выполнения данного кода показан на рис. 11.5.

```
<mx:TabNavigator id="view"
  width="200"
  height="200">

  <mx:Canvas id="redBox"
    label="Red"
    backgroundColor="#FF0000"/>
  <mx:Canvas id="greenBox"
    label="Green"
    backgroundColor="#00FF00"/>
  <mx:Canvas id="blueBox"
    label="Blue"
    backgroundColor="#0000FF"/>

</mx:TabNavigator>
```

Примечание

Помните, что для изменения названия вкладки в TabNavigator необходимо изменить свойство `label` соответствующего контейнера.

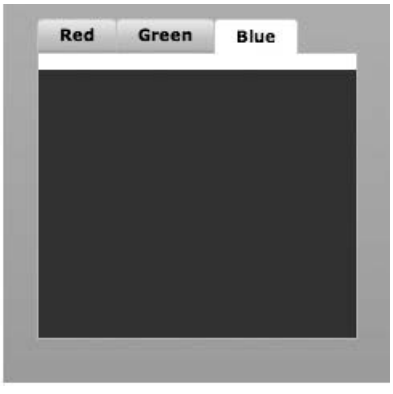

Рис. 11.5. *Три контейнера Canvas внутри TabNavigator; переключение между ними реализовано с помощью вкладок*

Для решения такой задачи используется следующий механизм: три контейнера внутри TabNavigator (в нашем случае Canvas) располагаются друг под другом, и видим лишь тот, что выбран пользователем. При выборе другой вкладки становится видимым соответствующий ей контейнер, скрывая все остальные. Таким образом, TabNavigator (как и другие контейнеры навигации) осуществляют контроль над *видом*, т. е. определенной комбинацией видимых одновременно компонентов.

Примечание

Свойства `title` и `label` элемента Panel не идентичны. Свойство `label` данного элемента (а также других контейнеров) используется для отображения текста на соответствующей ему вкладке в контейнере навигации, а `title` задает текст строки заголовка.

TabNavigator наследует все основные черты контейнера навигации ViewStack. Данный контейнер является основополагающим компонентом для реализации возможности смены видов, но он не обладает какими-либо визуальными средствами для выполнения этой задачи. Поэтому его целесообразно использовать в сочетании с такими элементами навигации, как LinkBar, ToggleButtonBar или TabBar. Если разместить ViewStack и связать его с соответствующим TabBar, выровняв их по вертикали, результат будет примерно таким же, как и при использовании TabNavigator:

```
<mx:VBox>

  <mx:TabBar
    dataProvider="{view}"/>

  <mx:ViewStack
    id="view"
```

```
    width="200"
    height="200">
    <mx:Canvas id="redBox"
      label="Red"
      backgroundColor="#FF0000"/>
    <mx:Canvas id="greenBox"
      label="Green"
      backgroundColor="#00FF00"/>
    <mx:Canvas id="blueBox"
      label="Blue"
      backgroundColor="#0000FF"/>
  </mx:ViewStack>

</mx:VBox>
```

TabBar, как и другие элементы навигации, обладает свойством dataProvider. В нашем случае оно привязано к конкретному компоненту ViewStack. TabBar выводит дочерние контейнеры и создает вкладки с соответствующими заголовками, как это сделал бы TabNavigator.

Зачем вообще затевать построение такой конструкции, если можно с тем же успехом использовать TabNavigator? Поскольку существуют различные элементы навигации, возможность их выбора для использования совместно с компонентом ViewStack в зависимости от требований приложения расширяет возможности реализации. В качестве примера можно привести ситуацию, когда необходимо использовать механизм, подобный вкладкам, но отличающийся от них визуально. Тогда можно использовать ToggleBar вместо TabBar:

```
<mx:VBox>

  <mx:ToggleButonBar
    dataProvider="{view}"/>

  <mx:ViewStack
    id="view"
    width="200"
    height="200">
    <mx:Canvas id="redBox""
      label="Red"
      backgroundColor="#FF0000"/>
    <mx:Canvas id="greenBox"
      label="Green"
      backgroundColor="#00FF00"/>
    <mx:Canvas id="blueBox"
      label="Blue"
      backgroundColor="#0000FF"/>
  </mx:ViewStack>

</mx:VBox>
```

В этом коде также используется свойство dataProvider, но приложение выглядит совсем по-другому. В качестве альтернативы можно также

использовать элемент навигации LinkBar, создающий набор кнопок, напоминающих гиперссылки.

Примечание

Учтите, что для полноценного функционирования элементы навигации не обязательно должны быть привязаны к компоненту ViewStack. Можно подключить такой элемент навигации, как LinkBar, к контейнеру TabNavigator, присвоив его свойству `dataProvider` значение идентификатора (`id`) данного контейнера. При этом у вас появится возможность осуществления навигации различными способами – вы сможете воспользоваться как вкладками TabNavigator, так и кнопками LinkBar, работа которых будет синхронизирована.

Существует еще одна причина, по которой вам может потребоваться компонент ViewStack. Иногда лучше контролировать вид непосредственно в коде приложения, ограничив возможности пользователя в этом отношении. К счастью, существует отличный способ контроля за выбранным видом во ViewStack и других контейнерах навигации, поскольку каждый такой контейнер имеет свойство `selectedIndex`, значением которого можно назначить целое число (начиная с нуля), соответствующее необходимому виду. В нашем примере, чтобы контейнер Canvas с зеленым цветом фона стал текущим видом, достаточно использовать следующий код ActionScript: `view.selectedIndex=1`. Если вам больше нравится красный контейнер, используйте код `view.selectedIndex=0`. С помощью привязки свойств `selectedIndex` элемента List и контейнера навигации можно достичь изменения выбранного вида ViewStack или иного контейнера навигации в зависимости от выбранного пункта в списке List. Существует еще один способ определения текущего вида – с помощью свойства `selectedChild`, которому можно присвоить `id` контейнера для отображения. При использовании такого метода нет необходимости задумываться о порядке следования видов. Рассмотрим следующий код, реализующий ситуацию, когда при нажатии на кнопку контейнер Canvas зеленого цвета станет видимым:

```
<mx:ViewStack
  id="view"
  width="200"
  height="200">
  <mx:Canvas id=""redBox"
    label="Red"
    backgroundColor="#FF0000"/>
  <mx:Canvas id="greenBox"
    label="Green"
    backgroundColor="#00FF00"/>
  <mx:Canvas id="blueBox"
    label="Blue"
    backgroundColor="#0000FF"/>
</mx:ViewStack>

<mx:Button
```

```
label="Make Green"
click="view.selectedChild = greenBox"/>
```

Такой метод вам особенно пригодится при создании сложных форм, состоящих из нескольких секций, когда одновременно отображаться может только одна из них. В этом случае процесс заполнения формы пользователем разбивается на несколько частей, что обычно довольно удобно – кому нравятся огромные формы без конца и края? Гораздо лучше разделить процесс ввода данных на несколько частей – это не только выглядит более четко и логично, но и дает пользователю некое ощущение продвижения вперед («Так, я уже заполнил первую, вторую и третью часть... уже почти все!»).

Для организации процесса ввода данных таким образом можно создать специальную кнопку Next (Далее), при нажатии на которую будет установлено соответствующее значение свойства selectedIndex или selectedChild контейнера ViewStack, содержащего части вашей формы. Чуть ниже, во врезке «Кнопка Back и журнал посещений», вы узнаете о том, что с этой целью можно использовать даже стандартные кнопки Back (Назад) и Forward (Вперед), к чему привыкло большинство пользователей.

Примечание

Значение свойства selectedIndex любого контейнера навигации по умолчанию равняется нулю, а свойство selectedChild соответствует первому по порядку дочернему элементу.

Создание приложения с фотоальбомом

А теперь давайте попробуем применить новые компоненты навигации. Вы наверняка сталкивались с различными видами при просмотре фотографий на своем компьютере. Большинство современных программ позволяют располагать просматриваемые изображения различными способами: подобно плиткам, по горизонтали, вертикально и т. д. Сейчас мы попробуем создать нечто подобное, а заодно научимся выводить графическую информацию средствами Flex. Итак, создайте новый проект с названием PhotoGallery. Мы организуем простой фотоальбом с возможностью переключения различных способов отображения фотографий.

Создание нескольких видов

Первым делом перенесите на сцену компонент TabNavigator в режиме Design. Автоматически созданный компонент будет включать в себя единственный дочерний элемент Canvas.

При выделении только что созданного компонента TabNavigator появится всплывающая панель инструментов с зажимом в виде перекрещенных стрелок и значками плюс и минус. Зажим позволяет перемещать элемент по сцене, а кнопки с плюсом и минусом предназначены

Выделение контейнеров навигации в режиме Design

При работе с компонентами навигации в режиме Design иногда бывает непросто определить, выделен ли сам компонент или его дочерние контейнеры. На помощь придет панель Outline или функция Show Surrounding Containers (о чем уже было сказано во врезке «Визуальное представление структуры вашего приложения» в главе 8). Но существует еще один способ выделения необходимого элемента. Чтобы выбрать именно TabNavigator, щелкните по зажиму, расположенному рядом со значками плюс и минус. Кроме того, с помощью зажима можно изменять расположение компонента, перемещая его по сцене. Двойной щелчок мышью по одной из вкладок также выделит родительский элемент TabNavigator.

для добавления и удаления вложенных контейнеров одним щелчком мыши. При нажатии на знак плюс открывается диалоговое окно, в котором можно выбрать тип нового контейнера (например, Canvas, VBox, Panel и т. д.), а также добавить его текстовую метку. Итак, с помощью данного диалогового окна в режиме Design или вручную разместите внутри контейнера TabNavigator два дочерних элемента Canvas с метками List View (Список) и Tile View (Плитка).

В дочерние контейнеры добавьте элементы List и TileList в соответствии с их названиями. Они должны занимать все свободное пространство родительских контейнеров, поэтому нужно установить значения их свойств `width` и `height` равными 100%. При этом код должен выглядеть примерно следующим образом:

```
<mx:TabNavigator
  width="200"
  left="10"
  top="10"
  bottom="10" >
  <mx:Canvas
    label="List View"
    width="100%"
    height="100%">
    <mx:List id="photosList"
      width="100%"
      height="100%"/>
  </mx:Canvas>

  <mx:Canvas
    label="Tile View"
    width="100%"
```

```
    height="100%">
    <mx:TileList id="photosTileList"
      width="100%"
      height="100%"
    </mx:TileList>
  </mx:Canvas>

</mx:TabNavigator>
```

Наполнение фотоальбома с помощью XML-данных

В нашем приложении теперь есть TabNavigator, предлагающий пользователю два варианта организации списка изображений (с помощью стандартного элемента List или с помощью TileList). Теперь самое время добавить сами изображения. Поскольку мы имеем дело с фотоальбомом, совершенно очевидно, что нам предстоит работа со списком фотографий. Можно пойти самым легким путем и использовать для этого данные, структурированные посредством XML. Если вам больше по душе другие источники данных и вы хотите воспользоваться ими при создании данного приложения, – все в ваших руках. Единственное, на что стоит обратить внимание, – структура вашего приложения должна повторять вышеописанную, в противном случае следует внести некоторые изменения, например задать другие имена атрибутам.

Нижеследующий XML-код можно поместить в файл с названием *photos.xml*, который будет находиться в папке src с исходным кодом вашего приложения `PhotoGallery` (вместе с основным файлом приложения *PhotoGallery.mxml*). Доступ к данному файлу можно осуществить с помощью компонента HTTPService.

```
<photos>
  <photo
title="Yawning Camel"
thumb="http://www.greenlike.com/photogallery/camel_thumb.jpg"
image="http://www.greenlike.com/photogallery/camel.jpg" />
<photo
title="Crowdy Head Lighthouse"
thumb="http://www.greenlike.com/photogallery/lighthouse_thumb.jpg"
image="http://www.greenlike.com/photogallery/lighthouse.jpg" />
  <photo
    title="Sun Shade"
    thumb="http://www.greenlike.com/photogallery/sunshade_thumb.jpg"
    image="http://www.greenlike.com/photogallery/sunshade.jpg" />
  <photo
    title="Uluru"
    thumb="http://www.greenlike.com/photogallery/uluru_thumb.jpg"
    image="http://www.greenlike.com/photogallery/uluru.jpg" />
  <photo
    title="Devil's Marbles"
    thumb="http://www.greenlike.com/photogallery/marble_thumb.jpg"
    image="http://www.greenlike.com/photogallery/marble.jpg" />
```

```
  <photo
    title="Mother and Child"
    thumb="http://www.greenlike.com/photogallery/mother_thumb.jpg"
    image="http://www.greenlike.com/photogallery/mother.jpg" />
  <photo
    title="Karnak Temple"
    thumb="http://www.greenlike.com/photogallery/temple_thumb.jpg"
    image="http://www.greenlike.com/photogallery/temple.jpg" />
  <photo
    title="Contemplating the Purchase"
    thumb="http://www.greenlike.com/photogallery/purchase_thumb.jpg"
    image="http://www.greenlike.com/photogallery/purchase.jpg" />
</photos>
```

Данный код представляет собой список фотографий, каждая из которых описывается с помощью атрибутов `title`, `thumb` и `image`. Атрибут `title` определяет название фотографии, а `image` содержит URL-адрес полноформатного изображения. Атрибут `thumb` предназначен для ссылки на уменьшенную копию изображения для предварительного просмотра.

Для осуществления доступа к данному файлу используется тег `<mx:HTTPService/>` со следующими свойствами:

```
<mx:HTTPService id="service"
  url="photos.xml"
  resultFormat="e4x" />
```

Не забывайте, что для инициации сервиса по окончании загрузки приложения необходимо установить обработчик события `applicationComplete`, вызывающий метод `send()`.

Теперь у вас имеются в наличии данные, остается лишь подключить их к двум разным спискам. Как вы помните, в предыдущей главе я рассказывал о том, что при использовании списков в виде XML-данных ссылка на них должна быть указана в свойстве `dataProvider` элемента управления List. Поэтому в нашем случае мы осуществим привязку значения данного свойства к `sevice.lastResult.photo`, что указывает прямо на список узлов `<photo/>` в возвращаемом XML-файле. Итак, при осуществлении такой привязки при загрузке приложения будут загружены и изображения.

Присвойте свойству `labelField` первого элемента List (с идентификатором `photosList`) значение `@title`. Поскольку название фотографии представлено XML-атрибутом, доступ к нему осуществляется с помощью E4X-выражения `@title`, а не просто `title`. Это указывает свойству `labelField` на то, что именно должно быть отображено.

Примечание

Вы можете использовать и XML-файл, расположенный на удаленном сервере, указав компоненту HTTPService его URL-адрес, например, *http://greenlike.com/flex/learning/projects/photogallery/photos.xml*.

Для использования в приложении PhotoGallery своих собственных изображений их нужно поместить в папку с исходным кодом и изменить URL-адреса, указанные в XML-документе, на соответствующие расположению ваших изображений. К примеру, если вы хотите использовать изображение whitedog.jpg, находящееся в папке с названием myphotos, измените атрибут image одного из тегов <photo/> на «myphotos/whitedog.jpg».

Как уже было сказано в предыдущей главе, все файлы, находящиеся в папке с исходным кодом, при сборке приложения будут автоматически скопированы в выходную папку (по умолчанию названную bin-debug). Соответственно, папка с вашими изображениями будет также скопирована, и при запуске приложение будет обращаться к ней для доступа к изображениям.

Вывод изображений из внешних источников

Далее нам предстоит собственно вывести изображения в приложении. Для этого создан специальный элемент управления Image, так что выполненить эту задачу не составит никакого труда. Для загрузки изображения в приложение достаточно лишь поместить на сцену данный элемент и указать URL-адрес изображения с помощью свойства source. При изменении значения данного свойства (например, при осуществлении связывания данных) изображение будет обновлено.

Итак, разместим на сцене элемент Image, зададим ему идентификатор image и привяжем значение его свойства source к photosList.selectedItem.@image. Таким образом осуществляется привязка ссылки на источник изображения к URL-адресу (определенному атрибутом image) выбранного в текущий момент узла XML. Теперь при выборе различных пунктов списка List будет загружено соответствующее ему изображение. Запустите приложение – оно должно выглядеть, как показано на рис. 11.6.

Если размеры элемента Image не заданы, они будут определены автоматически в соответствии с величиной показываемого изображения. То есть, если ширина фотографии 300 пикселов, а высота – 100 пикселов, элемент Image примет такие же размеры. В некоторых случаях это вполне соответствует замыслу разработчика, но чаще всего нужно знать размеры данного элемента заранее – это поможет более тщательно продумать общую схему расположения компонентов приложения. При задании размеров элемента Image напрямую отображаемое изображение будет автоматически отмасштабировано, чтобы уместить его в доступном пространстве. Поэтому стоит либо явно определить величину элемента Image, либо, что даже лучше, использовать ограничители для его привязки относительно краев приложения. В последнем случае размеры фотографии будут изменены так, что она займет свободное пространство. В дальнейшем при масштабировании окна приложения пользователем элемент Image будет соответственно менять размер отображаемого изображения.

Рис. 11.6. *Фотоальбом, представленный в виде списка (с помощью элемента List)*

Я также рекомендую воспользоваться свойством `horizontalAlign` для центрирования содержимого элемента Image. Это особенно необходимо при использовании изображений с книжной ориентацией, т. е. когда его высота превышает ширину – при этом центрирование (или его отсутствие) становится особенно заметным.

Отображение процесса загрузки

Загрузка изображения занимает некоторое время, особенно если скорость соединения с интернетом невысока. Иногда это может привести к неправильному восприятию, ведь скорее всего пользователь ожидает, что при выборе изображения из списка оно должно отобразиться мгновенно. При небольшой задержке он может решить, что произошла какая-то ошибка, и сразу выберет другой пункт. Это только усложнит ситуацию, поскольку процесс загрузки начнется заново.

Чтобы механизм работы вашего приложения был понятен пользователю, можно использовать элемент ProgressBar для отображения хода процесса загрузки изображения. Данный элемент можно использовать и в других целях, но в данном случае это просто идеальный инструмент. Вам достаточно лишь присвоить в качестве значения свойства `source` элемента ProgressBar имя (`id`) элемента Image, за которым необходимо следить.

Чтобы довести наше приложение до совершенства, прямо над только что созданным элементом Image разместим элемент ProgressBar с идентификатором `progressbar` и свойством `source`, указывающим на данное изображение. Теперь при выборе определенного пункта списка начнется загрузка изображения, ход которой будет отображен индикатором.

Кэш броузера и загрузка данных

Если вы уже производили загрузку изображения при запуске приложения PhotoGallery или если вы используете локальные изображения, индикатор загрузки сразу покажет значение 100%. Причина состоит в том, что изображение либо находится на локальном компьютере, либо было сохранено в *кэше*. Последнее означает, что изображение было скопировано на ваш компьютер для обеспечения быстроты доступа. Именно поэтому оно загружается мгновенно, и процесс загрузки не отображается.

Кэширование осуществляется броузером так же, как и при просмотре HTML-страниц, когда сохраняются изображения и другие файлы. Файлы SWF также могут быть сохранены в кэше. Это довольно заметно при работе с Flex-приложениями для веб – первая загрузка займет некоторое время и будет отображен индикатор загрузки, но во второй раз приложение будет загружено практически мгновенно.

Возможность кэширования позволяет значительно увеличить скорость загрузки данных. Большинство броузеров позволяют очистить кэш или просто запретить кэширование, что может оказаться полезным при разработке. При попытке запуска приложения PhotoGallery после очистки кэша снова появится индикатор, отображающий ход загрузки.

Очистка кэша приводит и к негативным последствиям, таким как ощутимое снижение скорости при загрузке часто посещаемых страниц, поскольку данные приходится загружать с сервера. При разработке приложений очень удобно пользоваться двумя разными броузерами: одним для выполнения ежедневных задач, другим собственно для разработки – его кэш можно очищать по мере необходимости.

По окончании загрузки изображения индикатор остается на экране. Это не лучшее поведение приложения, ведь изображение уже отображается целиком и в индикаторе загрузки больше нет необходимости. Хотелось бы, чтобы индикатор загрузки появлялся лишь при необходимости, а затем исчезал. Такого эффекта совсем нетрудно добиться: для этого нужно использовать свойство `visible` и несколько событий, характерных для элемента Image.

Для нас представляют интерес прежде всего события `open` и `complete` элемента Image. Первое из них происходит при начале загрузки данных, а второе – при ее окончании. С помощью этих событий можно регулировать видимость элемента ProgressBar.

В теге ProgressBar установите значение false свойства visible: теперь поначалу данный элемент будет невидимым. Затем добавьте обработчики событий open и complete элемента Image для контроля над его отображением. При начале загрузки изображения (событие open) появится индикатор загрузки, а по ее окончании (событие complete) он станет невидимым. Ниже представлен код, реализующий такое поведение:

```
<mx:Image id="image"
  source="{photosList.selectedItem.@image}"
  left="270"
  top="10"
  bottom="10"
  right="10"
  horizontalAlign="center"
  open="progressBar.visible = true"
  complete="progressBar.visible = false"/>

<mx:ProgressBar id="progressBar"
  x="270"
  y="10"
  source="{image}"
  visible="false" />
```

Теперь приложение стало более «чутким» – оно сообщает пользователю о том, что приложение реагирует на нажатия на элементы списка. Немного внимания к пользователю никогда не бывает лишним!

Настройка компонента TileList

Запустите приложение и посмотрите, что получилось. Как видите, вид Tile View (отображение плиткой) еще не готов. Для настройки работы элемента List было достаточно лишь задать ему свойство labelField, в то время как TileList обладает более широкими возможностями, такими как отображение небольших изображений для предварительного просмотра вместо простого названия фотографии. Для реализации этой возможности нам понадобятся itemRenderer в элементе TileList и еще один элемент Image.

Примечание

Использование ActionScript 3.0 API, распространяемого Adobe, предоставляет разработчику возможность подключения любых фотографий, поскольку взаимодействует с Flickr, популярным сайтом для обмена фотографиями. С помощью данного API можно не только искать и показывать фотографии, но и загружать их на компьютер и присваивать им «теги». Для полноценной работы с данным инструментом требуется хорошее знание ActionScript, но если вы хотите вывести ваше приложение PhotoGallery на качественно новый уровень, обязательно загрузите код, расположенный на *http://code.google.com/p/as3flickrlib*.

Значение свойства source этого элемента Image, предназначенного для создания уменьшенной копии изображения, должно соответствовать

атрибуту thumb большого изображения. Также стоит центрировать данный элемент с помощью свойства horizontalAlign, чтобы миниатюры были расположены более аккуратно. (Преимущество центрирования особенно заметно при выборе изображения или наведении на него курсора мыши – в этом случае появляется подсветка, и при выравнивании миниатюры по левому или правому краю она будет выглядеть неравномерно и не так красиво.)

Можно также обеспечить появление всплывающей подсказки с названием изображения при наведении на него курсора мыши. Для этого используется то же свойство, что ранее служило в качестве значения labelField элемента List, но в данном случае текст будет выведен в маленьком всплывающем окошке при наведении на изображения указателя мыши. Код элемента TileList должен выглядеть примерно следующим образом:

```
<mx:TileList id="photosTileList"
  dataProvider="{service.lastResult.photo}"
  width="100%"
  height="100%"
  <mx:itemRenderer>
    <mx:Component>
      <mx:Image
      horizontalAlign="center"
      source="{data.@thumb}"
      toolTip="{data.@title}"
      width="100"
      height="60" />
    </mx:Component>
  </mx:itemRenderer>
</mx:TileList>
```

Обратите внимание, что элементу Image заданы значения ширины – 100 пикселов и длины – 60 пикселов. Благодаря этим настройкам вне зависимости от реального размера картинки, используемой в качестве источника, соответствующие параметры ее ширины и высоты будут подогнаны под эти значения. (Это также увеличит производительность вашего приложения, поскольку размеры изображений известны заранее и нет необходимости их вычислять). Теперь при запуске приложения в Tile View отображается аккуратно расположенный в виде плиток набор миниатюр. Однако при щелчке по какой-либо из них пока ничего не происходит.

Синхронизация работы списков

Установить источник для элемента Image, предназначенного для отображения большого изображения, можно несколькими путями. Во-первых, можно воспользоваться тегом <mx:Binding/> для осуществления привязки элемента Image к нескольким источникам. При этом можно привязать Image к обоим элементам управления списком – так,

чтобы при изменении выбранного пункта как в элементе photosList, так и в photosTileList было загружено соответствующее изображение. Однако это не самый лучший способ решения нашей задачи, поскольку при переключении из одного вида в другой может оказаться, что в каждом из них выделены разные пункты. Например, вы выбираете третью миниатюру в списке TileList, загружается соответствующее полноформатное изображение, а затем вы переключаетесь в List View, в списке которого выделено совершенно другое изображение. Иными словами, механизм работы списков не синхронизирован.

Вместо привязки большого элемента Image к обоим элементам управления списком достаточно оставить привязку только к списку photosList. Для синхронизации работы обоих списков можно установить для каждого из них обработчик события change. Это позволит обновлять List при изменении TileList и наоборот. Теперь при смене выбранного пункта в списке TileList будет выбран соответствующий ему пункт Tile, что в свою очередь приведет к обновлению свойства source элемента Image и отображению требуемого изображения на экране.

Однако у этого метода есть одно ограничение, обусловленное порядком создания дочерних элементов контейнером навигации. Во избежание слишком медленной загрузки приложения не все виды создаются одновременно. К примеру, в нашем случае в приложении имеются две вкладки, но при запуске приложения инициализируется лишь одна из них, а содержимое второй вкладки выстраивается только при обращении пользователя к ней. (При наличии двух вкладок такой принцип может показаться неоправданным и ненужным, но если в приложении, скажем, 10 видов, их одновременная загрузка может занять значительное время.) Из-за наличия такого механизма в нашем приложении выбор пунктов списков не сможет быть синхронизирован до тех пор, пока не будет загружено содержимое второй вкладки (что произойдет только при переключении пользователя на данный вид). Ведь событие change попытается обновить TileList, но ничего не произойдет, поскольку он еще не создан.

Примечание

Можно осуществить двустороннюю привязку свойств selectedIndex списка TileList к аналогичному свойству List и наоборот, но это может привести к проблеме с рекурсией, существенно снижающей производительность программы.

Данную проблему несложно решить, поскольку контейнеры навигации обладают специальным свойством, позволяющим управлять созданием его видов. Оно называется creationPolicy и служит «направляющей» в вопросах создания элементов. Данное свойство принимает одно из четырех значений – all, auto, queued и none. По умолчанию используется auto, при этом создается только первоначальный вид, о чем было сказано выше. Значение all создает все имеющиеся виды одновременно, в то время как queued создает все дочерние контейнеры, а затем

вложенные в них элементы по порядку. Установка значения none полностью отменяет создание каких-либо видов – ее можно использовать в ситуациях, когда инициация создания видов должна осуществляться каким-либо иным способом. В нашем случае идеальным решением проблемы станет задание данному свойству TabNavigator значения all. Теперь оба вида будут созданы и синхронизированы сразу при запуске приложения.

Примечание

Настройка свойства creationPolicy по умолчанию (auto) обеспечивает обычно наилучшую производительность приложения, поскольку в этом случае виды создаются постепенно, по мере необходимости.

Конечный код приложения представлен в нижеследующем листинге, а примерный вид приложения показан на рис. 11.7.

```
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  layout="absolute"
  applicationComplete="service.send()">

  <mx:HTTPService id="service"
    url="photos.xml"
    resultFormat="e4x" />

  <mx:TabNavigator
    width="250"
    left="10"
    top="10"
    bottom="10"
    creationPolicy="all">
    <mx:Canvas
      label="List View"
      width="100%"
      height="100%">
      <mx:List id="photosList"
        dataProvider="{service.lastResult.photo}"
        width="100%"
        height="100%"
        labelField="@title"
        change="photosTileList.selectedIndex=photosList.
                selectedIndex"/>
    </mx:Canvas>

    <mx:Canvas
      label="Tile View"
      width="100%"
      height="100%">
      <mx:TileList id="photosTileList"
        dataProvider="{service.lastResult.photo}"
        width="100%"
        height="100%"
        change="photosList.selectedIndex=photosTileList.
```

```
                selectedIndex" >
        <mx:itemRenderer>
          <mx:Component>
            <mx:Image
              horizontalAlign="center"
              source="{data.@thumb}"
              toolTip="{data.@title}"
              width="100"
              height="60" />
            </mx:Component>
          </mx:itemRenderer>
        </mx:TileList>
      </mx:Canvas>

    </mx:TabNavigator>

    <mx:Image id="image"
      source="{photosList.selectedItem.@image}"
      left="270"
      top="10"
      bottom="10"
      right="10"
      horizontalAlign="center"
      open="progressBar.visible = true"
      complete="progressBar.visible = false"/>

    <mx:ProgressBar id="progressBar"
      x="270"
      y="10"
      source="{image}"
      visible="false" />

</mx:Application>
```

Рис. 11.7. *Приложение с фотоальбомом в виде Tile View*

Кнопка Back и журнал посещений

По умолчанию компонент TabNavigator обладает чрезвычайно мощной функцией, позволяющей использовать кнопку Back (а также и кнопку Forward) вашего броузера. К примеру, при выборе второй вкладки в приложении `PhotoGallery` для возврата к первой вкладке можно нажать на кнопку Back в вашем броузере (данный процесс наглядно показан на рис. 11.8). Дело в том, что компонент «общается» со внутренним механизмом броузера, основанном на JavaScript, регулирующим работу функций кнопок

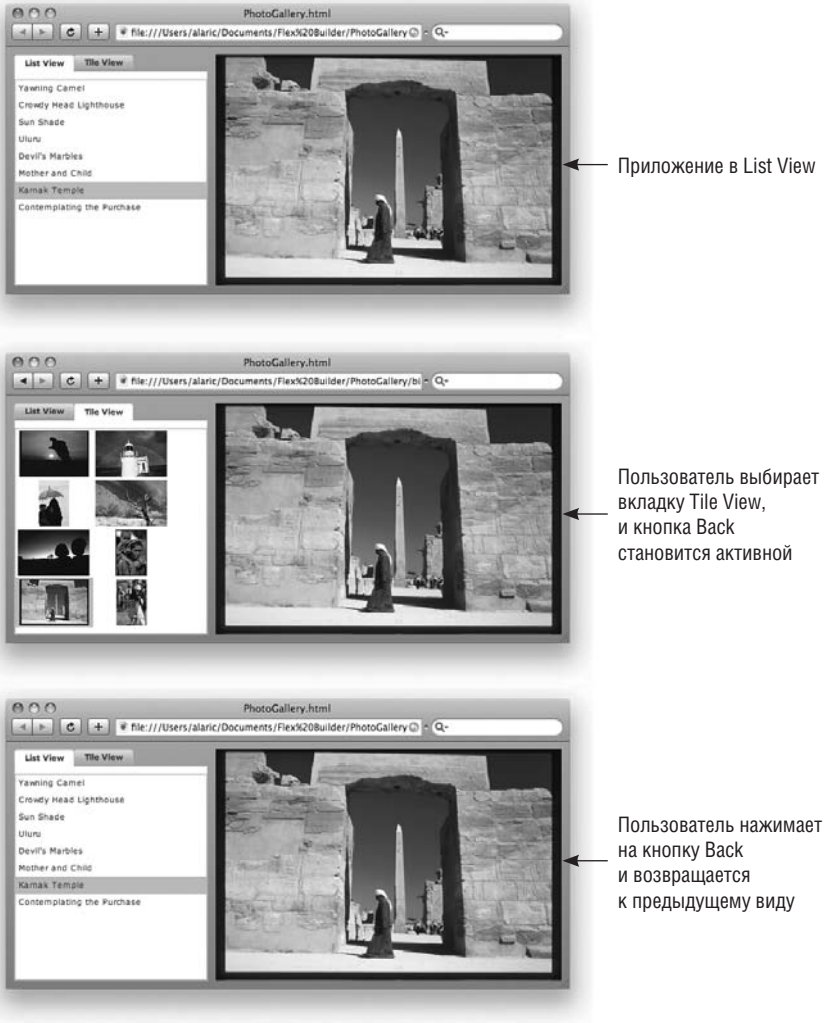

***Рис. 11.8.** Изменение текущего вида с помощью кнопки Back*

для возврата или перехода к следующей странице. Возможно, вы уже обращали внимание на папку history внутри папки bin-debug ваших приложений. В ней содержатся файлы JavaScript и другие файлы, необходимые для управления такого рода переходами.

Это очень полезная функция, поскольку она позволяет пользователю осуществлять привычные для него действия при работе с приложениями (так же, как ранее с веб-страницами). Функция доступна благодаря наличию свойства `historyManagementEnabled` контейнеров навигации. По умолчанию она включена для контейнеров TabNavigator и Accordion, но выключена для ViewStack.

Вдобавок, для управления журналом посещений при работе с другими элементами можно использовать класс `mx.managers.BrowserManager`. С его помощью можно решить и массу других задач. Его использование требует глубоких знаний в области программирования, так что при желании можно ознакомиться с его возможностями в документации Flex.

Итак, мы успешно создали несложный фотоальбом и узнали много нового о возможностях контейнеров навигации. Попробуйте самостоятельно внести какие-то изменения в наше приложение, используя другие виды элементов и контейнеров навигации. К примеру, замените тег `<mx:TabNavigator/>` на `<mx:Accordion/>` и посмотрите, что получится. Контейнер Accordion располагает элементы по вертикали, поэтому его удобно использовать при создании форм, состоящих из нескольких разделов. Такая форма будет выглядеть логично и упорядоченно, что немаловажно для пользователей – в качестве примера представьте себе форму, в которой нужно заполнить адрес для отправки товара, а затем платежную информацию. В этом компоненте элегантно реализовано переключение видов. О том, как добавить свои собственные эффекты к другим контейнерам навигации, вы узнаете в главе 13.

Заключение

В этой главе вы познакомились с методами создания структуры Flex-приложения, позволяющей пользователю переключаться между различными видами. Это позволяет наиболее рационально использовать доступное пространство на экране. Такой метод также весьма удобен для организации различных частей приложения, например при использовании форм или диалоговых окон с широким набором опций. Это также позволяет не перегружать пользователя информацией и избежать появления полос прокрутки, что, как правило, происходит при попытке отобразить все содержимое сразу.

Вы научились использовать новые навигационные компоненты для создания настраиваемых видов приложения с фотоальбомом, создав для выбора фотографии как текстовый список фотографий, так и список миниатюр. Теперь вы умеете загружать изображение как с локального компьютера, так и из сети Интернет, отображая процесс его загрузки. В данной главе мы вновь столкнулись с такими важными понятиями, как представление элемента и загрузка внешних ресурсов, и использовали наши знания при создании нового приложения.

Возможность регулирования видимости компонентов и использования элементов навигации открывают перед разработчиком новые грани Flex. Но существует еще один на удивление простой, но эффективный способ создания гибкого интерфейса приложения, о котором речь пойдет в следующей главе.