Мастера FLASH

Аларик Коул

ИЗУЧАЕМ FLEX™ 3

Руководство по разработке насыщенных интернет-приложений

O'REILLY®

Adobe Developer Library

СИМВОЛ®

Learning Flex 3

Getting up to Speed with Rich Internet Applications

Alaric Cole

O'REILLY®

Изучаем Flex 3

Руководство по разработке насыщенных интернет-приложений

Аларик Коул

Санкт-Петербург — Москва
2009

Аларик Коул

Изучаем Flex 3. Руководство по разработке насыщенных интернет-приложений

Перевод А. Минаевой

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Выпускающий редактор	*П. Щеголев*
Научный редактор	*М. Антипин*
Редактор	*Ю. Бочина*
Корректор	*С. Минин*
Верстка	*Д. Орлова*

Коул А.

Изучаем Flex 3. Руководство по разработке насыщенных интернет-приложений. – Пер. с англ. – СПб.: Символ-Плюс, 2009. – 384 с., ил.

ISBN 978-5-93286-143-1

Книга представляет собой введение в мир Adobe Flex и станет прекрасным пошаговым руководством для всех, кто хочет изучить эту технологию «с нуля». Издание охватывает все аспекты разработки Flex-приложений, начиная со знакомства с основными необходимыми инструментами и заканчивая распространением результатов работы. Кратко, не перегружая читателя излишней информацией, описаны базовые возможности ActionScript и MXML.

Автор сопровождает свой рассказ множеством примеров и предлагает сразу же применять полученные навыки на практике. Это позволяет уже с первых глав книги начать создание собственных насыщенных интернет-приложений, которые по ходу чтения будут приобретать все более сложную структуру и интерактивность. Все упражнения и примеры можно скачать с веб-сайта книги и скопировать код в свои проекты, что поможет избежать досадных опечаток. Это издание – отличный выбор для тех, кто хочет легко и быстро научиться создавать как настольные, так и веб-приложения на основе технологии Flex.

ISBN 978-5-93286-143-1
ISBN 978-0-596-51732-8 (англ)

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 324-5353, www.symbol.ru. Лицензия ЛП N 000054 от 25.12.98.

Налоговая льгота – общероссийский классификатор продукции
ОК 005-93, том 2; 953000 – книги и брошюры.

Подписано в печать 25.02.2009. Формат 70×100¹/₁₆. Печать офсетная.
Объем 24 печ. л. Тираж 1500 экз. Заказ №

Отпечатано с готовых диапозитивов в ГУП «Типография «Наука»
199034, Санкт-Петербург, 9 линия, 12.

Adobe Developer Library (Библиотека Adobe для разработчиков), совместный проект O'Reilly Media Inc. и Adobe Systems, Inc., – авторитетный источник для разработчиков, использующих технологии Adobe. Их всеобъемлющие ресурсы предлагают учебные решения, помогающие создавать передовые интерактивные веб-приложения, которые могут использоваться практически кем угодно на любой платформе.

Adobe Developer Library представляет из первых рук книги высочайшего качества и инновационные ресурсы, посвященные новейшим инструментальным средствам для разработки насыщенных интернет-приложений. Таким образом *Adobe Developer Library* способствует подготовке высококвалифицированных специалистов. Область рассмотрения охватывает ActionScript, программное обеспечение Adobe Flex®, Adobe Flash® и Adobe Acrobat®.

Последние новости о книгах, сетевых ресурсах и прочее ищите на *adobedeveloperlibrary.com*.

Оглавление

Предисловие

Многие разработчики ПО делали первые шаги на профессиональном пути, изучая COBOL, Pascal и другие языки программирования, используемые уже много лет. Мой путь был совершенно иным. Я не изучал информатику в школе, а по образованию я антрополог.

Я не мог подумать, что когда-либо буду зарабатывать на жизнь разработкой программного обеспечения. Но вообще-то мне очень нравилось создавать графику и веб-сайты, и будучи школьником, я подрабатывал именно таким способом. Однажды мне захотелось сделать какую-нибудь необычную анимацию для сайта, и я решил посвятить выходные изучению Flash IDE. Денег на хорошую книгу не было, и мне пришлось потратить кучу времени на чтение прилагаемой документации. Через несколько часов мне наконец удалось заставить свой карандашный рисунок двигаться, что привело меня в дикий восторг. Я буквально погрузился в документацию, узнавая все больше и больше. Так воскресный эксперимент перерос в постоянное увлечение, и я постепенно стал переходить от простейших анимированных эффектов к написанию сценариев.

Мне казалось, что Flash – это круто. Я мог взять текст и добавить к нему эффекты, недоступные при использовании обычных HTML и JavaScript (DHTML). Он также позволял создавать приложения, которые нельзя было создать средствами DHTML, имевшимися в распоряжении у разработчика на тот момент. Например, я мог вставить в свой Flash-ролик информацию с удаленного компьютера, скажем, прогноз погоды, и полностью управлять его видом. Я мог предоставить пользователю возможность посылать электронные письма через Flash-приложение без перезагрузки страницы. Я мог разместить на своем сайте галерею фотографий, поворачивая их под различным углом, добавляя рамки, таким образом, что все это выглядело как настоящий фотоальбом. Я был поражен и зачарован.

ActionScript, язык технологии Flash, со временем развивался, а с ним и мои навыки. Я обнаружил, что чем большую свободу я давал свому воображению, тем сложнее становился написанный мной код. Этап написания одноразовых пробных сценариев был пройден; я всерьез стал изучать программирование. Задуманные проекты превосходили технические возможности Flash, и мне постоянно приходилось придумывать

новые способы оптимизации моего кода. Вскоре я начал искать новые пути реализации таких проектов.

А затем появился Flex. Поначалу он был весьма далек от совершенства, но я понимал, что за ним будущее. Теперь можно было создавать приложения посредством простого XML с его четкой структурой, что напоминало весьма несложное и понятное использование HTML в моей прошлой практике. В этом действительно был смысл: теперь я мог создавать сложные приложения гораздо проще и быстрее, чем когда-либо.

Не я один высоко оценил возможности Flex. С появлением новых версий его начинали использовать все больше разработчиков. Flex превратился в мощное средство разработки программного обеспечения, сочетавшее в себе как традиционные, так и собственные методы. А сам процесс создания приложений остался таким же увлекательным.

Для кого предназначена эта книга

Эта книга рассчитана на всех желающих познакомиться с использованием Flex, включая новичков в мире разработки программного обеспечения. Это означает, что даже если вы никогда не имели дел с Flash IDE, веб-дизайном или программированием, вы можете спокойно приступать к чтению и изучать приведенные примеры. Я даю пояснения по основным понятиям программирования, но все же это не учебник по программированию или разработке программного обеспечения. Моя цель состоит в том, чтобы сделать ваш путь к самой сути технологии Flex как можно более увлекательным. Я тешу себя надеждой, что к концу каждой главы у вас будет появляться куча новых вопросов, а чтение следующей главы – позволять находить ответы на них.

Flex – среда программирования с широкими возможностями, и я не ставлю перед собой цель охватить их полностью в рамках этой книги. Если вам понравится данная технология, есть множество способов оттачивать свое мастерство в дальнейшем. Например, вы можете изучать код, написанный другими, или прочитать книгу о Flex и технике программирования, рассчитанную на более опытных пользователей. В моей книге вы найдете увлекательные и актуальные примеры, с которыми можно экспериментировать самостоятельно и которые послужат необходимым фундаментом для дальнейших шагов в более глубоком изучении предмета.

Структура книги

Данная книга выстроена так, чтобы читать ее от начала и до конца. Информация подается поступательно, так что в каждой следующей главе используются знания, полученные вами в предыдущих главах. Таким образом, я руководствуюсь практическими принципами, которые позволят вам постепенно научиться применять основные понятия

при разработке настоящих приложений. Книгу можно спокойно читать вдали от компьютера, изучая приведенный код, сверяя его с результатами, показанными на подробных иллюстрациях. Затем вам достаточно будет лишь просмотреть главу, чтобы с успехом применить изученный код в своих собственных примерах, каждый раз на практике закрепляя знания о новом понятии. Если какая-либо из рассматриваемых тем вас не интересует, ее можно просто пропустить. Однако обязательно зайдите на справочный сайт *www.greenlike.com/flex/learning*, где можно получить код примеров к каждой главе.

О чем написано в этой книге

Моя цель – создать пошаговое руководство, охватывающее все аспекты разработки Flex-приложений, начиная со знакомства с основными инструментами, необходимыми для изучения базовых возможностей ActionScript и MXML, и заканчивая распространением результатов вашей работы. Мой выбор тем для рассмотрения обусловлен моими представлениями о том, что нужно знать, чтобы начать работать, при этом не перегружая читателя излишней информацией.

Поэтому книга начинается с объяснения самых первых шагов, а именно – настройки вашего компьютера для начала разработки Flex-приложений. Я выбрал наиболее популярное и, как мне кажется, лучшее решение для работы с Flex: Flex Builder. Для разработки Flex-приложений было бы достаточно простого текстового редактора и интерпретатора командной строки, но это не лучший вариант для начинающего разработчика. Визуальный инструментарий, предлагаемый Flex Builder, делает процесс изучения и написания кода во Flex более удобным и увлекательным.

Я предлагаю простые и наглядные примеры, опираясь на которые вы можете сразу же взяться за создание Flex-приложений. Затем мы обсудим основы языков MXML и ActionScript, чтобы у вас появилось представление о них и их взаимодействии во Flex. После этого я рекомендую вам начать создание собственных приложений и развивать их по мере прочтения книги. Мы поговорим о навыках, необходимых для работы с Flex, в том числе о работе с данными и структурировании ваших приложений, и достаточно полно рассмотрим возможности создания динамического интерфейса пользователя и интерактивного взаимодействия. Обладая этими знаниями, вы сможете вдоволь попрактиковаться в создании собственных приложений и «перерасти» крикливые анимации, на которые похожи некоторые Flash-ролики.

Возможности Flex настолько широки, что на страницах данной книги, предназначенной для новичков, я решил не касаться некоторых сложных вопросов. Везде, где мне это представляется полезным, я привожу ссылки на материалы, которые могут быть использованы для дальнейшего изучения.

Справочный веб-сайт

Все упражнения, предлагаемые в книге, можно скачать со справочного веб-сайта, расположенного по адресу *www.greenlike.com/flex/learning*. Хотя многочисленные разделы с кодом примеров приведены на страницах книги, но, возможно, вам будет удобнее скачать Flex-проекты и скопировать соответствующий код в свои примеры. Это поможет избежать досадных опечаток при проработке примеров. Я старался так подобрать примеры, чтобы вы могли опираться на них при изучении каждой главы.

Обозначения, принятые в данной книге

В данной книге приняты следующие обозначения:

Рубленый шрифт

Для обозначения заголовков, пунктов, кнопок меню и модификаторов клавиатуры (как Alt или Command).

Курсив

Для обозначения новых понятий, URL-адресов, адресов электронной почты, названий и расширений файлов, а также пути к файлу.

`Моноширинный шрифт`

Для выделения кода на ActionScript, текста, выводимого при выполнении сценария, тегов XML и HTML, а также содержимого файлов.

`Жирный моноширинный шрифт`

Выделяет команды или иной текст, который должен ввести пользователь.

`Моноширинный курсив`

Для обозначения текста, который следует заменить на определенный пользователем.

Примечание

Обозначает совет, предложение или заметку общего характера.

Внимание

Обозначает предупреждение или предостережение.

Использование примеров кода

Эта книга будет служить подспорьем в вашей работе. В целом вы можете свободно использовать приведенный на ее страницах код в своих приложениях или документации. Если объем используемого вами кода невелик, нет необходимости обращаться к нам за разрешением.

К примеру, включение в вашу программу нескольких строчек кода из данной книги, продажа или иное распространение CD-дисков с записью примеров, взятых из книг издательства O'Рэйли (O'Reilly), или цитирование материалов данной книги при ответе на вопрос не требует специального разрешения. Но при включении в составляемую вами документацию значительного объема кода из данной книги нужно получить наше разрешение.

При использовании наших материалов мы не требуем обязательной ссылки на источник, однако будем вам за нее признательны. Ссылка на источник, как правило, включает в себя название, имя автора, издательство и ISBN. К примеру, «Learning Flex 3» by Alaric Cole. Copyright 2008 O'Reilly Media, Inc., 978-0-596-51732-8.

Если использование вами кода из книги выходит за рамки разрешенного свободного использования, пожалуйста, свяжитесь с нами по адресу *permissions@oreilly.com*.

Нам интересно ваше мнение

Пожалуйста, направляйте ваши вопросы и комментарии по поводу этой книги издателю по следующему адресу:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
(800) 998-9938 (in the United States or Canada)
(707) 829-0515 (international or local)
(707) 829-0104 (fax)

У этой книги есть страница в Интернете, на которой представлены список опечаток, примеры и другие дополнительные сведения. Адрес страницы:

www.oreilly.com/catalog/9780596517328

Адрес электронной почты для комментариев и вопросов технического характера, связанных с этой книгой:

booksquestions@oreilly.com

Для получения информации о наших книгах, конференциях, ресурсных центрах и сети O'Рэйли Нэтворк (O'Reilly Network) посетите наш веб-сайт по адресу:

www.oreilly.com

Благодарности

Как и фильм, хорошая техническая книга – это результат совместного труда многих людей. Будучи автором текста, я бы не смог создать эту книгу в одиночку.

От своего имени хочу выразить благодарность:

Мике Лаакеру (Micah Laaker) – за то, что он подтолкнул меня к написанию этой книги, и за полезные советы по ее содержанию.

Эли Робисон (Eli Robison) – за постоянную поддержку и истинную дружбу.

Шарифу Завайдеху (Sharif Zawaideh) (*http://globalimagesllc.com*) – за потрясающие фотографии, украшающие страницы книги.

Хеппу Маккою (Hepp Maccoy) – за быстрый и содержательный отзыв.

Джастину Келли (Justin Kelly) – за проверку изложенных мной сведений на практике (несмотря на один неудачный опыт).

Майклу Хоху (Michael Hoch) – за человеческое понимание и терпение, когда я приходил на работу совершенно измотанный после бессонных ночей работы над книгой.

Эллен Рабинович (Allen Rabinovich) – за нечто большее, чем работа технического редактора.

Лидии Шембри (Lydia Schembri) – за сильную мотивацию, позитивную энергию и обходительность.

Маме и папе за то, что они лучшие родители.

Робину Томасу (Robyn Thomas), Стиву Вайсу (Steve Weiss), Мишель Фильши (Michele Filshie), Деннис Фитцжеральд (Dennis Fitzgerald), Дэвиду Ван Нессу (David Van Ness) и другим сотрудникам O'Рэйли (O'Reilly) – за постоянную поддерку. Качество этой книги – результат их руководства и упорной работы.

Команду Adobe Flex – за выпуск отличного нового продукта.

В этой главе:

- Что такое Flex?
- Что такое Adobe AIR?
- Для каких целей можно использовать Flex
- Почему именно Flex?
- Сравнение Flex с другими технологиями
- Когда не нужно использовать Flex

1

Первое знакомство

Будущее уже наступило. Просто оно еще неравномерно распределено.

Уильям Гибсон

Добро пожаловать в будущее. Adobe Flex 3 – это совершенно новая технология создания функциональных веб-приложений и традиционных программ (с использованием Adobe AIR). Воздвигнув мост, соединивший выразительность и вседоступность Adobe Flash и широкие возможности традиционной разработки приложений, Flex избирает свой собственный путь.

Я начал работать с Flex с момента его появления и могу с уверенностью сказать, что сейчас настал самый подходящий момент для изучения этой технологии. Flex 3 позволяет создавать привлекательные и мощные приложения быстрее и проще, чем когда-либо. После появления Flex прошло всего несколько лет, но его потенциал растет невиданными темпами. На профессионалов в области Flex-разработок существует стабильный спрос на рынке труда, и т. к. данный продукт превратился в полноценную открытую систему, разработчики поступят весьма мудро, добавив умение работать с ней в свою профессиональную палитру, а новички могут не сомневаться в правильности выбора технологии для изучения.

Что такое Flex?

Flex – это метод простого и быстрого создания насыщенных интернет-приложений (rich Internet applications, RIA). Прежде всего, это среда разработки RIA, воспроизводимых проигрывателем Flash Player. Кроме того, Flex – это новый язык программирования, в основе которого лежит MXML, язык разметки, в свою очередь основанный на расширяемом языке разметки (Extensible Markup Language, XML), использование которого делает создание приложений действительно легким и эффективным. В отличие от разработки приложений для некоторых платформ, требующей создания бинарных файлов, MXML – это обычный текст, который легко читать и модифицировать в простом текстовом редакторе. Соответственно, распространение кода не сложнее распространения обычных текстовых файлов.

Flex – современный гибридный язык программирования

Такая основанная на XML система создания приложений вполне знакома классическим веб-программистам, т. к. в ней используется язык разметки и язык сценариев, напоминающий JavaScript. Веб-разработчики и веб-дизайнеры, привыкшие к использованию гипертекстового языка разметки (HTML) и языка JavaScript, быстро освоят Flex. Несмотря на значительные различия с точки зрения архитектуры, общее сходство технологий позволит им быстро войти в курс дела. Во Flex объединены все достоинства традиционных языков программирования и современные стандарты и методы веб-разработок.

Как выглядят Flex-приложения

Вы наверняка встречали какие-либо Flex-приложения во всемирной паутине, и, возможно, пользовались приложениями Adobe AIR. Может быть, вам попадались только приложения на основе используемой по умолчанию темы оформления Aero – в полупрозрачной голубой цветовой гамме. С большой долей вероятности вы встречали Flex-приложения, совсем не подпадающие под это описание (и вы, скорее всего, и не задумывались над тем, что это именно Flex-приложения). Дело в том, что внешний вид Flex-приложений весьма разнообразен. Flex-приложение не обязательно должно походить на стандартное приложение Windows, Mac или других платформ благодаря возможности простого регулирования его оформления путем подключения различных файлов с темами. Более того, наличие огромного количества бесплатных готовых тем позволяет изменить вид приложения одним щелчком мыши. Вы даже можете дать вашим пользователям возможность изменять вид приложения, подключая к ним свои собственные темы!

Flex – это Flash

Вскоре после первого появления Flex моя подруга, восхищенная выразительными возможностями Flash, стала расспрашивать меня о новой технологии. Взглянув на несколько созданных с помощью Flex приложений, она сказала, что они выглядят сравнительно блекло. «Одни кнопки да панели, – отметила она. – Что в этом интересного?»

Тогда я быстро состряпал пару примеров. Первым из них было небольшое красочное приложение, напоминающее Flash-ролик, вторым – программа с типичным интерфейсом из кнопок и панелей, к которой я быстро добавил несколько эффектов плавного перехода и некоторых других. Увидев результат, она поняла причину моего столь сильного ажиотажа по поводу новой технологии.

Flex-приложения, как и Flash-ролики, являются файлами с расширением *.swf* (обычно произносится как «свиф»). Это очень компактные файлы, для воспроизведения которых на экране (чаще всего в броузере) требуется проигрыватель Flash Player. Это значит, что вы можете создавать полноценные приложения небольшого размера (и, соответственно, быстро скачиваемые), которые, за редким исключением, будут выглядеть и функционировать одинаково на любом компьютере с любой операционной системой.

Что такое Flash Platform?

Flash Platform – общее название платформы разработки, основанной на технологии Flash Player. Даже если вы никогда об этом специально не задумывались, в вашей системе скорее всего установлен Flash Player, широкие возможности которого вам наверняка приходилось неоднократно наблюдать. В противном случае вы принадлежите к малочисленной группе людей, составляющей всего 1% от общего числа пользователей Интернета. Таким образом, на данный момент Flash Player является одной из самых распространенных программ. Разрабатывая приложения под Flash Platform, вы можете быть уверенными, что они будут доступны самой широкой аудитории (рис. 1.1). Практически любая платформа позволяет запускать Flash-приложения; даже при программировании под Windows круг потенциальных пользователей менее широк.

Flex – это Flex SDK

Flex – это совокупность компонентов пользовательского интерфейса и других компонентов для выполнения различных задач при создании приложений. Все это предлагает набор средств разработки программно-

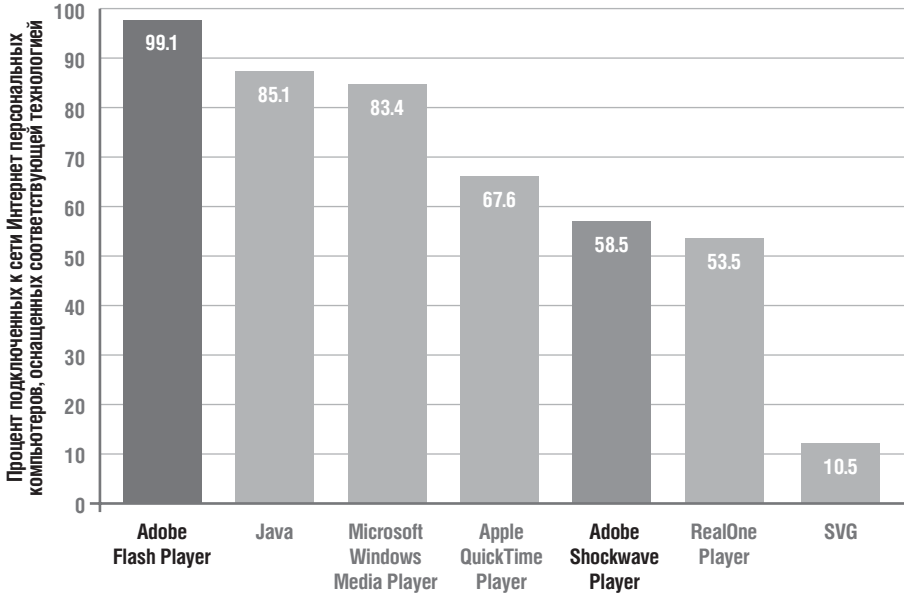

Рис. 1.1. *Доступность проигрывателя Flash Player*

го обеспечения Flex Software Development Kit (SDK). Flex SDK состоит из компилятора, средств создания документации, обширной библиотеки компонентов пользовательского интерфейса и утилит, упрощающих процесс разработки. Теперь для создания кнопки или работы с анимационным объектом вместо последовательного написания строчка за строчкой кода на языке низкого уровня Flex-разработчику достаточно написать `<mx:Button/>` или же перетащить кнопку мышкой в нужное место.

Flex SDK может быть использована с обычным текстовым редактором или с Flex Builder, мощной средой разработки, созданной Adobe. Хотя совсем не обязательно пользоваться именно Flex Builder, однако это могло бы стать существенным подспорьем в работе. Если у вас пока нет конкретных предпочтений, то, выбрав Flex Builder в качестве среды разработки Flex-приложений, вы значительно сократите время на их создание.

Что такое Adobe AIR?

Adobe Integrated Runtime (AIR) — это среда для запуска приложений, позволяющая переносить Flash-ролики и другие веб-приложения на настольные ПК. Неужели, учитывая все достоинства современных веб-приложений, это кому-нибудь может понадобиться? Может, и главным образом потому, что броузеры ограничивают функционирование при-

ложений. В них отсутствует встроенная поддержка перетаскивания объектов с рабочего стола, они налагают некоторые ограничения из соображений безопасности и не позволяют использовать веб-приложения при отсутствии связи с Интернет. Кроме того, многим пользователям нравится, когда приложение находится в доке (Mac OS) или меню Пуск (Windows), откуда его можно быстро запустить, щелкнув по иконке.

AIR дает преимущества и разработчикам. Учитывая масштабы распространения Всемирной паутины, многие разработчики сосредотачивают все свое внимание на таких технологиях, как HTML, JavaScript и Flash, а не на традиционных средствах разработки ПО. Ныне удача улыбается интересующимся технологией Flex – ведь это превосходный способ создания приложений, выполняемых в среде AIR.

Имея в своем распоряжении Adobe AIR, вам не нужно изучать C# или Java для создания автономного приложения: вы можете это сделать, используя имеющиеся знания Flex, Flash или JavaScript; если же вы захотите также создавать веб-приложения, вам не понадобится переучиваться. Причем – и это вполне убедительный довод в пользу использования AIR – вам не нужно разрабатывать программу под конкретную операционную систему, т. к. функционирование приложений AIR платформонезависимо. Иными словами, не нужно выбирать между разработкой для Windows, Mac или Linux – вы пишете программу на избранном вами языке программирования, а благодаря Adobe AIR каждый сможет ею воспользоваться (рис. 1.2).

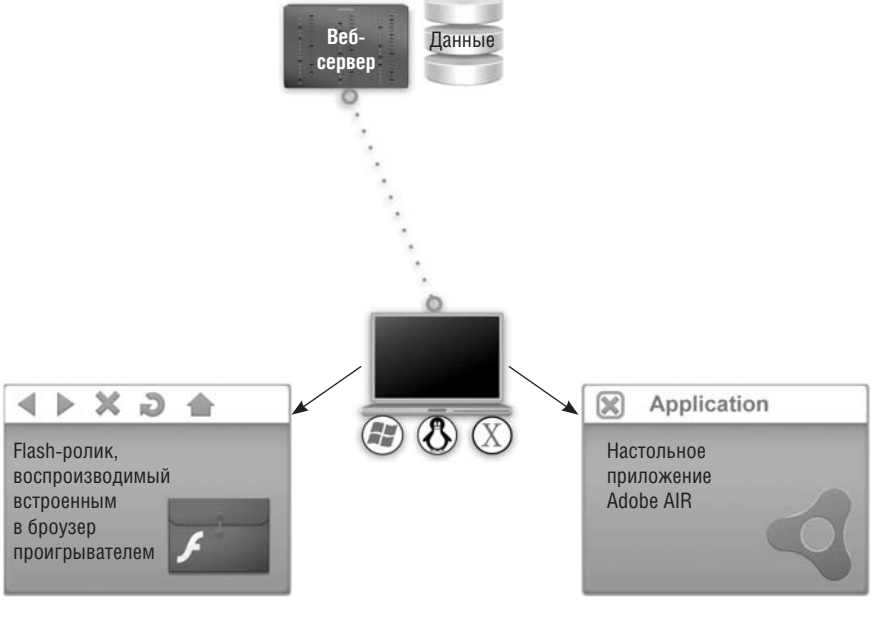

Рис. 1.2. *Использование Flex-приложений*

Для каких целей можно использовать Flex

Flex – это новое слово в сфере разработки насыщенных интернет-приложений. Выражение «насыщенные интернет-приложения» было предложено компанией Macromedia (ныне Adobe) в 2002 году и употреблялось по отношению к тенденции создания все более ярких и экспрессивных веб-приложений. Вначале HTML-документы в сети Интернет были всего лишь документами. Они содержали текст, а позднее также изображения и мультимедиа. Данная парадигма «клиент-сервер» подразумевала, что пользователь, вводя URL-адрес в окне своего броузера, запрашивает определенный документ. Учитывая масштабы распространения Интернета, находчивые умельцы быстро научились создавать серверные приложения и программы, доступ к которым можно было получить через сеть. Вспомните, сколько форм вам пришлось заполнить, указав свое имя, адрес электронной почты и подтвердив введенные данные нажатием на кнопку Отправить. Через несколько мгновений загружалась новая страница, сообщавшая вам, что данные формы успешно отправлены (или о необходимости исправить допущенную при заполнении ошибку). Такова модель «клиент-сервер», в рамках которой «тонкий» клиент (броузер) запрашивает содержимое и отправляет запрос на сервер для обработки. Для создания динамической HTML-страницы сервер должен был сформировать ее и отправить клиенту, который отобразит ее на экране. Все это требует некоторого времени.

С приходом JavaScript появилась возможность немного разгрузить сервер путем переноса части обрабатывающей программы в клиентскую среду. Например, раньше при изменении пользователем параметров товара, предлагаемого в интернет-магазине, все расчеты, такие как стоимость доставки или налог с оборота, производились на сервере. Благодаря сценариям такие изменения стало можно обрабатывать прямо на клиентской машине, затем обновляя содержимое страницы в зависимости от произведенных пользователем действий. В противоположность тонкому клиенту «толстый» клиент для обработки таких сценариев и перезагрузки страницы требует несколько большей мощности от пользовательского компьютера.

Тем не менее некоторым разработчикам, включая меня, этого было недостаточно. Нам хотелось большего: анимации, плавных переходов, яркости и выразительности. Нам хотелось, чтобы данные можно было загружать, не обновляя страницу. До создания Ajax единственным выходом для многих в этой ситуации был Flash.

Вначале среда разработки Flash IDE была средством создания анимации и мультимедийного контента для Интернета. Однако по мере ее развития в нее добавлялось все больше интерактивных элементов, и вскоре на базе новой платформы разработчики стали изобретать способы программирования таких программ, как игры. Пользуясь быстротой, функциональностью и малым объемом приложений, создаваемых во Flash, другие разработчики начали создавать сложные приложения,

позволявшие загружать и отображать данные, создавать каталоги и фотоальбомы. Но так как основным назначением Flash IDE было создание анимации, разработка приложений со сложной схемой взаимодействий часто оказывалась трудоемким и неудобным для классических разработчиков процессом. В 2003 году вышла версия Flash Professional с новыми функциями, такими как компоненты многократного использования и коннекторы данных, позволявшими еще больше ускорить процесс разработки, однако и она оставляла желать лучшего, в особенности с точки зрения разработки приложений уровня предприятий и при совместной работе большого числа разработчиков.

А затем на сцену выходит Flex. С ним разработка ярких функциональных приложений стала простой и удивительно гибкой. Будучи более ориентирована на разработчика, данная модель позволила Java-программистам (и программистам на других языках) сразу начать работать с новой технологией, не задаваясь вопросами вроде «Что такое временная диаграмма?». Более того, успеху нового языка, построенного по принципу языков разметки, способствовало то, что он позволял с легкостью читать и распространять код.

Почему именно Flex?

Возможно, вы задали себе этот вопрос, взяв в руки данную книгу. Почему именно Flex? В чем его преимущества и недостатки? Flex был создан с с целью облегчить процесс разработки функциональных интернет-приложений, но в скором времени он эволюционировал до мощной платформы, пригодной даже для создания традиционного программного обеспечения (с помощью Adobe AIR).

Создание приложений

Flex предназначен для быстрой разработки мощных интерактивных приложений, основанных на технологии Flash Player. В него входят мощные настраиваемые компоненты, значительно ускоряющие и упрощающие конфигурацию приложений. Данные приложения могут работать в среде броузера или AIR, таким образом, Flex – идеальное решение, позволяющее однократно написать код приложения, которое в дальнейшем может быть использовано как в веб-среде, так и в качестве обычного настольного приложения.

Легкость взаимодействия

Приложения Flex обладают высоким уровнем интерактивности благодаря поддержке технологии связывания данных, отличной системе обработки событий и набору превосходных компонентов для обратной связи с пользователем. Добавьте к этому возможность создания красивых эффектов, таких как плавность переходов, и со всей очевидностью станет ясно, что Flex – отличный инструмент разработки.

Что такое веб-приложения

Вы, конечно, знакомы с традиционными настольными приложениями; среди них такие часто используемые программы, как ваш веб-броузер и текстовый процессор. Запуск веб-приложения осуществляется через броузер. Возможно, вы даже пользуетесь как обычной версией какого-либо продукта, так и его веб-аналогом. Чаще всего такой программой является почтовый клиент. Благодаря настольной почтовой программе вы имеете доступ к своим почтовым сообщениям в любое время, даже при отсутствии связи с Интернетом. Преимущество веб-приложения заключается в его доступности повсеместно, где есть Интернет (например, в поездке, когда домашнего компьютера нет под рукой). А теперь представьте, что одна и та же программа работает как через веб-интерфейс, так и в обычном режиме, при этом сохраняя не только единый вид и набор функций, но и единый исходный код. С Flex это вполне реально.

Скорость разработки

Это самый быстрый способ создания привлекательных приложений, скорость которого по сравнению с разработкой во Flash IDE возрастает по принципу снежного кома. Несмотря на то, что все, что делается во Flex, может быть сделано и во Flash IDE, разработка во Flex займет лишь малую часть времени, требующегося Flash-разработчикам. А если вы пользуетесь Flex Builder, скорость разработки возрастет еще в несколько раз.

Скорость для пользователей

Компоненты Flex написаны на языке ActionScript 3.0 – новейшей реализации языка программирования, лежащего в основе Flash Player, который был существенно переработан с целью увеличения производительности, и результат налицо. Безусловно, нужно по возможности оптимизировать свои программы, но теперь вы можете быть уверены, что даже при большом объеме данных и анимации ваше приложение будет нормально функционировать, не поглощая все ресурсы компьютера.

Четкость

Flex – воплощение тайных мечтаний разработчиков. При проектировании во Flex можно следовать традиционным принципам написания кода, касающимся организации кода и создания приложений на основе классов. Следуя практике веб-дизайна, Flex поддерживает разделение содержания и оформления, позволяя управлять внешним видом

приложений извне. Таким образом, вы с легкостью можете изменять «скины» ваших приложений, а также мгновенно добавлять с этой целью любые бесплатно распространяемые темы. Кроме того, среде Flex есть что предложить и приверженцам шаблона проектирования модель-вид-контроллер (Model-View-Controller, MVC). Также совершенно бесплатно доступны различные библиотеки, такие как оболочка Cairngorm, упрощающие внедрение MVC.

Бесплатность

Flex Builder, визуальный редактор Flex де-факто, приобретается за деньги, но оболочка Flex распространяется совершенно бесплатно. Поэтому при желании вы можете редактировать свое приложение в обычном текстовом редакторе и компилировать его в режиме командной строки, не заплатив при этом ни копейки. Эта книга нацелена на новичков, а постижение азов с помощью Flex Builder будет гораздо проще и быстрее, поэтому я часто буду говорить о том, как работать с ним. К счастью, Adobe предлагает полнофункциональную демо-версию сроком на 30 дней, которую можно скачать по адресу *www.adobe.com/products/flex*.

Попробуйте Flex онлайн

Если вы пока не хотите что-либо скачивать, но вам интересно увидеть Flex «в действии», попробуйте Flex Online Compiler, расположенный по адресу *http://try.flex.org*. Вы можете скопировать любой код или выбрать один из приведенных примеров, и результат его обработки немедленно отобразится в вашем броузере.

Открытый исходный код

Flex – продукт с открытым исходным кодом. Это означает, что вы имеете свободный доступ к коду любого компонента, который можно неоднократно использовать самим. Возможность изучать код оболочки Flex сослужит вам как начинающему разработчику хорошую службу. Это также означает, что Flex полностью в ваших руках. Да, именно так – вы можете самостоятельно вносить исправления в код и ваши замечания могут быть учтены при выпуске следующих версий. Кроме того, код компилятора также находится в открытом доступе.

Благодаря открытости Flex сформировалось сообщество разработчиков, занимающихся созданием расширений и исправлением недочетов. Создано огромное количество полезных и качественных компонентов-надстроек к базовому набору функций, так что при необходимости вы обязательно найдете требующийся инструмент для работы.

А если у вас также появится желание улучшить Flex, вашим идеям будут только рады. Будущее Flex в ваших руках.

Быстрота передачи данных

При воспроизведении приложений проигрывателем Flash Player снижаются временные затраты на передачу данных, и в результате вашим пользователям не приходится долго ждать. Flex обеспечивает встроенную поддержку объектов Java и XML для обмена данными, а также поддержку формата Action Message Format (AMF). При использовании серверов, поддерживающих Java и ColdFusion, вы можете гораздо быстрее передавать, обрабатывать и извлекать сжатые двоичные данные в ваше Flex-приложение, чем при работе с обычными приложениями. А для поклонников PHP существует AMFPHP, альтернативный вариант поддержки AMF с открытым исходным кодом для использования с PHP.

Кроме того, работа с данными во Flex невероятно проста. С помощью мастера по работе с данными во Flex Builder вы с легкостью можете подключить базу данных, и, скорее всего, необходимый код будет сгенерирован автоматически. При работе с сервером ColdFusion вы предоставляете базу данных, а Flex Builder автоматически построит необходимые клиентские и серверные компоненты.

Внешняя привлекательность

Возможно, вас вполне устроит внешний вид Flex-приложения, определенный темой по умолчанию, но в общем и целом визуальное оформление приложений ограничено лишь возможностями вашего воображения. Совсем не обязательно делать их похожими на программное обеспечение для определенной операционной системы и вообще на какое-либо уже существующее программное обеспечение. Благодаря легкости манипулирования стилями и темами оформления во Flex вы можете быстро менять вид ваших программ, переработав всего одну строчку кода.

Благодаря широкому и разнообразному набору элементов, предлагаемому в готовом наборе, а также возможности воспользоваться множеством элементов, находящихся в открытом доступе, вы можете создать интерфейс на любой вкус. Flex Charting предлагает богатый выбор инструментов для создания диаграмм и другого рода визуализации данных. Можно строить гистограммы, секторные диаграммы, ежедневные чарт-гистограммы. Все что хотите. Благодаря мощности и выразительности Flash, а также простоте Flex набор компонентов визуализации данных постоянно пополняется сторонними разработками.

Сравнение Flex с другими технологиями

Flex – это гибридная технология, вобравшая в себя лучшие черты современных языков программирования, одновременно оставаясь верной принципам таких стандартов, как XML или каскадные таблицы стилей (CSS). В этом смысле наблюдается как сильное сходство Flex с некоторыми существующими технологиями, так и в равной мере ее отличие от других технологий.

Flash IDE

Как и интегрированная среда разработки Flash (Flash IDE), Flex создает приложения, воспроизводимые Flash Player. Однако единственной общей особенностью создания приложений Flash и Flex является использование одного и того же языка сценариев, в остальном же они очень различны. Flash – это в первую очередь инструмент создания анимации и графический редактор; средства для разработки ПО были добавлены в него позднее. Flex же, напротив, с самого начала предназначался для создания приложений. Тем, кто ранее пользовался Flash, имея дело лишь с самыми простыми сценариями, работа с Flex поначалу покажется более сложной, однако программисты на Java или C будут чувствовать себя как рыба в воде.

Говоря о технической стороне, все, что можно сделать во Flex, можно сделать и с помощью Flash. Средства Flash IDE позволяют разрабаты-

Пара слов о термине «Flash»

Возможно, мне стоило начать этот разговор раньше. В данной книге вы часто встретите термин «Flash», которому нелегко дать однозначное определение. Дело в том, что он может относиться к различным понятиям. Во-первых, Flash IDE, инструмент создания анимации и разработки приложений, с которого все и началось. Во-вторых, это ролики, часто встречающиеся в сети Интернет: анимация, рекламные баннеры, приложения, отображаемые в броузере и являющиеся файлами с расширением .swf. Это также обобщенный термин, обозначающий технологию, основанную на Flash Player, небольшом плагине для броузера, воспроизводящем Flash-приложения на компьютере. В различных контекстах вы можете встретить это слово в самых разных значениях, но в этой книге я постараюсь разграничить эти понятия. Говоря «Flash», я обычно имею в виду технологию или собственно приложение. При упоминании средств разработки программного обеспечения я скажу «Flash IDE» или «инструментальные средства создания Flash».

вать мощные приложения, и собственно именно этим я занимался большую часть своего карьерного пути (а многие занимаются и по сей день). Однако работу следует выполнять предназначенным именно для нее инструментом, и Flex был создан именно для разработки приложений. В нем есть поддержка простой работы с данными, встроенная поддержка стилей и смены темы оформления, функциональные инструменты управления интерактивностью и множество других полезных возможностей. (В скором времени вы увидите, что Flex позволяет легко добавлять в ваши приложения сложную анимацию и другие графические эффекты.) Однако Flex не графическая программа и не средство создания анимации, поэтому, если вы хотите создавать видеоролики или мультфильмы, оптимально будет воспользоваться средой разработки Flash.

Языки C

Хотя Flex и основан на языке, отличном от C++, Objective C и подобных, он оправдает все ожидания разработчика. Flex Builder – интегрированная среда разработки, схожая с Visual Studio и XCode, в которой вы можете проектировать, писать код и запускать приложение. Но одним из самых значимых преимуществ Flex является возможность модификации и даже полного написания кода вне среды разработки в простом текстовом редакторе благодаря использованию языка разметки MXML. В отличие от сценариев код, написанный на языке разметки, легче читать, редактировать и распространять. Так как Flex – объектно-ориентированный язык и программная среда, основанная на использовании классов, разработчики на C++ и других языках программирования быстро войдут в курс дела. Однако проще всего будет разработчикам C# из-за большей унифицированности языков.

Java/Java FX

Flex чем-то похож на Java и платформу Java Swing. По структуре и методике использования язык сценариев ActionScript похож на Java. ActionScript унаследовал концепцию группы классов, и синтакис языков практически идентичен. Самые большие отличия наблюдаются при сравнении с MXML, но вы вскоре увидите, что к ним легко адаптироваться. Так как Flex Builder основан на технологии Eclipse, Java-разработчики с легкостью найдут с ним общий язык.

Java, как и Flex, позволяет разрабатывать приложения, выполняющиеся как в веб-среде, так и в клиентской среде. Однако широкая распространенность и компактность Flash Player по сравнению с Java SDK делает приложения Flex доступными большему количеству пользователей.

HTML/JavaScript/Ajax

Flex был создан в то время, когда Интернет переживал самый пик популярности, и потому его структура охватывает множество особенностей, характерных для разработки веб-приложений. Наиболее значимые – использование тегового языка (MXML) вместе с соответствующим стандартам ECMA[1] языком ActionScript. Благодаря схожести синтаксиса ActionScript с синтаксисом JavaScript использующие JavaScript программисты с лекостью освоят ActionScript. Веб-программисты, знакомые с XML или HTML, быстро поймут принцип действия MXML. Несмотря на то, что при более глубоком рассмотрении отношения MXML с ActionScript совсем не похожи на отношения HTML с JavaScript, но само наличие этой взаимосвязи может стать ключом к пониманию для многих классических веб-разработчиков.

Одни и те же вещи часто можно сделать как с помощью Ajax (асинхронный JavaScript и XML (asynchronous JavaScript and XML), так и Flex; в этом случае разработчики обычно предпочитают использовать уже знакомую им технологию; более того, многие из них весьма категоричны в своих предпочтениях. Если вы пока новичок в этих сферах, у вас есть возможность составить объективное мнение по этому вопросу, но если вы уже достигли уровня специалиста, я могу предположить, что вы признаете некоторые преимущества Flex. То, что вы делаете средствами Ajax, может быть сделано и во Flex, причем созданные веб-приложения не будут в такой степени зависеть от идеологии представления информации в виде страницы. В последнее время библиотеки Ajax существенно расширились, и при наличии подходящего инструментария в Ajax вполне можно работать. И тем не менее, писать и поддерживать код на MXML и ActionScript легче, и при этом приходится меньше заниматься изобретением хитроумных трюков для нормальной работы приложения, чем при работе с Ajax.

Silverlight/XAML

Silverlight – технология, созданная компанией Microsoft. Это сочетание языка разметки, основанного на XML, и языков программирования. В этом смысле понимание технологии Silverlight определенно поможет при изучении Flex. При создании Silverlight была предпринята попытка добиться полной кросс-платформенности. Любое приложение, созданное вами во Flex, будет выглядеть примерно одинаково на любом компьютере, так как оно обрабатывается плагином Flash (Flash Player), установленным на большинстве компьютеров. Плагин Silverlight менее компактен, но его популярность растет. Silverlight предлагает большой набор готовых мощных элементов управления и компо-

[1] ECMA Интернешнл (ECMA International) – основанная в 1961 году ассоциация, деятельность которой посвящена стандартизации информационных и коммуникационных технологий. – *Прим. перев.*

новки содержимого, хотя благодаря открытости исходного кода быстро увеличивается и число компонентов Flex, создаваемых сторонними разработчиками.

OpenLazlo

OpenLazlo – это популярная оболочка с открытым кодом, использующая для создания насыщенных интернет-приложений масштабируемую векторную графику (SVG) и XML/JavaScript. Работающие с ней разработчики быстро разберутся в особенностях MXML и ActionScript, а также откроют для себя много новых функциональных возможностей. Flex – это также продукт с открытым исходным кодом!

Как узнать приложение Flex

С появлением Web 2.0, когда пользовательские интерфейсы с широкими функциональными возможностями стали обычным делом, границы между HTML и Flash стали совершенно размытыми. Еще пару лет назад, увидев необычный переход или красивую анимацию, у вас бы не возникло сомнений, что это Flash. Сегодня с первого взгляда это трудно определить.

Существует один способ проверить, использована ли при создании какой-либо части страницы технология Flex – просто щелкните по ней правой кнопкой мыши (или щелчком мыши при нажатой клавише Control – для операционной системы Mac). Если содержимое оказалось приложением Flex или Flash, на экране появится меню с последним пунктом «About Adobe Flash Player». Это говорит о том, что содержимое отображается с помощью Flash Player, однако это совсем не обязательно Flex-приложение. Таким образом, не существует безошибочного способа определения, т. к. встречаются достаточно сложные приложения, выполненные во Flash IDE. Однако, чем больше у вас будет опыта работы с приложениями Flex и его основными компонентами, тем увереннее вы сможете это определять при первом же взаимодействии с приложением.

Когда не нужно использовать Flex

Flex – отличная технология, и вы увидите, что она способна решать казавшиеся неразрешимыми в прошлом задачи. Однако она не универсальна. Если вы хотите сделать несколько несложных анимаций, не занимаясь при этом написанием кода, использование такого инструмента, как Flash, будет самым подходящим решением. Из-за использования библиотеки компонентов файлы Flex будут иметь боль-

ший размер, чем Flash-приложения или приложения, написанные на чистом ActionScript (однако эту проблему можно решить путем кэширования; я затрону этот вопрос ближе к концу книги). В большинстве случаев некоторое увеличение размера файла вполне окупается уменьшением временных затрат на разработку и преимуществами функциональности оболочки Flex. Но вам, возможно, не захочется, чтобы простые элементы или небольшие приложения с простым набором функций увеличивались в объеме. К счастью, вам совсем не обязательно использовать компоненты Flex (и даже MXML) для создания Flex-приложения – вполне возможно создать проект только на ActionScript и скомпилировать его бесплатным компилятором Flex. А основная интегрированная среда разработки Flex — Flex Builder — прекрасно подходит для выполнения таких проектов.

Оцените Flex

Выбирая Flex для себя или своей компании, посетите веб-страницу *www.oreilly.com/catalog/evaluator1/*, где можно приобрести краткое руководство «Оценка Flex для Ваших проектов» (Flex Early Evaluation: Assessing Flex and Your Project Needs).

Заключение

Я надеюсь, что теперь у вас сложилось более четкое представление о том, что такое Flex. Созданный для разработки приложений, функционирующих на базе Flash Player, он стал ключевым инструментом в мире насыщенных настольных и веб-приложений. Благодаря богатому набору компонентов, ускоряющих процесс работы над приложением, и новому языку разметки, упрощающему написание кода, Flex – отличный выбор для многих разработчиков. Вы узнали о сходствах и различиях между данной технологией и Flash, а также технологиями других языков программирования и о том, для чего следует использовать Flex, а для чего – нет. Если вы хотите изучить именно эту технологию, в последующих главах вы найдете много полезной информации. В следующей главе мы перейдем к изучению азов Flex и Flex Builder.

2

В этой главе:
- Альтернативы Flex Builder
- Знакомство с Flex Builder и Eclipse
- Запуск вашего первого приложения

Настройка вашей системы

Adobe Flex 3 распространяется бесплатно, и в процессе разработки приложений можно использовать любой редактор. Ваш код будет представлять собой файл в простом текстовом формате, распознаваемый любым текстовым редактором, а скомпилировать его будет можно с помощью бесплатного компилятора Flex. Но, несмотря на это, при начальном знакомстве с Flex такой инструмент, как Adobe Flex Builder, просто незаменим. Это главный редактор кода Flex, помогающий не только быстро ознакомиться с его возможностями, но и с легкостью систематизировать и компилировать ваш код. Я начал работать с технологией Flex с момента ее появления и в прошлом пробовал пользоваться разными редакторами, но остановил свой выбор на использовании в повседневной работе Flex Builder. Поэтому при написании этой книги я поставил перед собой цель рассказать о разработке Flex-приложений с использованием Flex Builder.

Flex легко установить, скачав его с сайта *www.adobe.com/products/flex*. Если вы приняли решение пользоваться выбранным вами редактором и компилировать код с помощью командной строки или иным способом, то можете ограничиться скачиванием Flex SDK. Тем не менее, я советую вам установить Flex Builder, так как компания Adobe предлагает демо-версию с неограниченными возможностями сроком на 30 дней. (Правда, на создаваемых диаграммах будет отображаться полупрозрачный логотип, но в остальном эта версия полнофункциональна). При покупке или скачивании Flex Builder перед вами встанет вопрос выбора одного из вариантов, о чем я подробнее расскажу в разделе «Версии Flex».

Альтернативы Flex Builder

Если срок лицензии на Flex Builder истек, если по каким-либо причинам вы не можете его установить или просто твердо решили им не пользоваться, можете работать с любым текстовым редактором. Однако, так как вы будете преимущественно иметь дело с языком, основанном на XML, будет эффективнее использовать текстовый редактор, предназначенный для написания кода на языке разметки, т. е. включающий в себя такие полезные опции, как цветовое выделение синтаксических конструкций. Ниже я приведу несколько примеров популярных программ:

Eclipse

Этот бесплатный редактор с открытым исходным кодом при установленном плагине для работы с XML можно с успехом применять для работы. Более того, Eclipse – не просто редактор, а полноценная интегрированная среда разработки, которую можно настроить и для компиляции ваших приложений. *www.eclipse.org*

Flash Develop (только для Windows)

Это популярная среди программистов на ActionScript интегрированная среда разработки с открытым исходным кодом. Теперь в ней есть также и возможность редактировать и компилировать Flex-приложения. *http://osflash.org/flashdevelop*

TextMate (только для Macintosh)

Отличный текстовый редактор со встроенной поддержкой ActionScript; рекомендуется подключить расширение для работы с Flex, благодаря которому работа с MXML-кодом станет удивительно легкой. *http://macromates.com*

TextPad (только для Windows)

Активно позиционируемый в качестве мощного и легко настраиваемого редактора – неплохой выбор для написания и редактирования MXML-кода вручную.

Таким образом, остановив свой выбор на одном из редакторов, вы решите одну часть задачи – написание и редактирование кода; следующая часть задачи – компиляция. Благодаря открытости Flex новые возможности компиляции приложений появляются буквально каждый день, и вам не составит труда найти подходящий вариант. Предлагаемые сторонними разработчиками решения создаются на основе бесплатного компилятора командной строки, использование которого сводится к написанию буквально нескольких символов в окне терминала или в командной строке.

Примечание

Для получения дополнительной информации об использовании компилятора командной строки Flex ознакомьтесь с документацией Adobe's LiveDocs по адресу *http://livedocs.adobe.com/flex/3/html/compilers_01.html*.

Знакомство с Flex Builder и Eclipse

Установив Flex Builder, ознакомьтесь с его основными функциями. Flex Builder создан на основе популярной интегрированной среды разработки с открытым исходным кодом Eclipse IDE. Она весьма функ-

Еще несколько слов об IDE

IDE (Integrated Development Environment) – **интегрированная среда разработки**, т. е. программа, предназначенная для создания других программ. Иными словами, это единый инструмент программиста, используемый для написания кода, организации проектов, отладки приложений и их запуска. В качестве примеров популярных IDE можно привести также Microsoft Visual Studio и XCode.

Говоря о Flex Builder, я время от времени буду упоминать Eclipse IDE, чаще всего в тех случаях, когда речь пойдет об особенностях, не характерных для Flex Builder, но являющихся неотъемлемой частью Eclipse IDE, положенной в его основу.

Для каких целей используется Eclipse? По умолчанию она предназначена для работы с языком Java, но благодаря архитектуре, основанной на подключении плагинов, может работать и с другими языками программирования. Многие программисты на Java используют Eclipse в качестве основного средства разработки, но она также прекрасно подходит для работы с JavaScript, HTML, C, Python и многими другими языками. Если вам необходимо распространять и сопровождать свой код при работе в команде, в вашем распоряжении система контроля версий исходного кода.

Adobe приняла решение взять за основу своего продукта Eclipse IDE из-за ее популярности среди Java-программистов и благодаря наличию в ней многих полезных и проверенных на практике функций. Таким образом, больше внимания уделялось разработке новых возможностей, а не проектированию IDE в целом. Кроме того, данная среда разработки кроссплатформенна, а значит, нет необходимости создавать отдельные версии для Mac, Windows и Linux.

> Функциональность Eclipse особенно ярко проявляется при разработке сразу на нескольких языках. Многие Flex-разработчики наряду с созданием Flex-приложений могут писать код на HTML, PHP, ColdFusion и открывать все редактируемые файлы в одной программе.
>
> Eclipse написана на языке Java и изначально была создана в рамках одного из проектов IBM. Ее исходный код стал полностью открытым в 2001 году.

циональна, но, вероятно, у нее не самый привлекательный и удобный для пользователя интерфейс, поэтому я постараюсь помочь вам разобраться, что к чему.

Версии Flex

Flex Builder доступен в двух версиях: автономной (или стандартной) и как плагин для Eclipse. Что лучше, спросите вы. Это дело вкуса. Если вы пользуетесь уже настроенной в соответствии со своими предпочтениями Eclipse и намерены заниматься не только разработкой Flex-приложений, будет удобнее скачать и установить только плагин. Однако если вы заинтересованы главным образом в разработке Flex-приложений, будет предпочтительнее установить автономную версию. При этом вам будут доступны такие вещи, как установка иконки Flex в меню «Пуск» и на панели задач, экран-заставка и прочие приятные мелочи. Честно говоря, в интегрированной стандартной версии доступ к необходимым функциям гораздо проще. В своих примерах я буду использовать автономную версию, поэтому, если вы используете расширение для Eclipse, возможны случаи, когда вид экрана немного отличается или пункты меню расположены другим образом. Вы всегда можете отдельно установить стандартную версию, а потом подключить плагин для Eclipse.

Установка Flex Builder

Запустите скачанную программу установки Flex Builder и следуйте появившимся на экране инструкциям. Вам будет задано несколько вопросов о том, где вы будете хранить файлы проектов, и т. д.; если у вас нет на этот счет особых предпочтений, доверьтесь настройкам по умолчанию. По завершении процесса установки запустите Flex Builder. Откроется окно, изображенное на рис. 2.1.

На рисунке отображается стартовая страница Flex, а именно, окно встроенного броузера. На ней вы найдете несколько инструкций и примеров проектов, которые помогут вам получить общее представление о Flex. Конечно, у вас есть эта книга, но ведь всегда приятно сначала самостоятельно освоиться. При выборе ссылки Full Tutorial в области

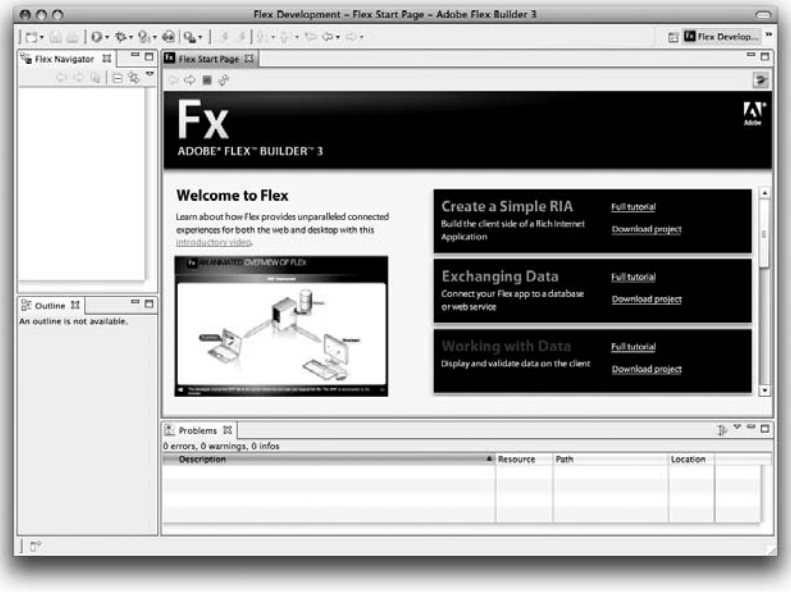

Рис. 2.1. *Стартовая страница Flex*

Create a Simple RIA откроется новое окно броузера, отображающее страницу с полным списком разделов, посвященных знакомству с Flex. Вероятно, вам понравится анимационная презентация возможностей Flex из раздела Get oriented with Flex.

Краткий обзор интерфейса редактора

Открыв только что установленную программу, вы увидите, что интерфейс Flex Builder, как и Eclipse, основан на использовании панелей. Панель Flex Navigator используется для организации файлов и проектов, на панели Outline отображается древовидная структура вашего приложения, а на панели Problems вы увидите сообщения об ошибках и сбоях. К счастью, пока у вас не должно быть никаких сбоев! Мы рассмотрим особенности интерфейса немного позже, а пока давайте займемся чем-то более увлекательным и запустим приложение.

Запуск вашего первого приложения

Имейте в виду, что для Eclipse/Flex Builder все сущее – проект. Это означает, что при разработке Flex-приложений вы всегда будете иметь дело не с одним-единственным файлом, а с целым набором взаимосвязанных файлов, посредством которых описывается работа вашего приложения. Таким образом, для создания нового приложения необходимо создать проект.

Импорт архива

Для создания первого проекта мы импортируем во Flex Builder уже существующий проект, который можно скачать с моего сайта. Зайдите по ссылке *www.greenlike.com/flex/learning/projects/simple.zip* и сохраните файл на рабочий стол или в любое другое удобное для вас место.

Примечание

На справочном сайте, посвященном этой книге, находится исходный код проектов, над которыми вы будете работать по ходу прочтения книги. Адрес страницы *www.greenlike.com/flex/learning*.

Скачав архив в формате .zip, снова откройте Flex Builder и выберите File→Import→Flex Project. При этом откроется диалоговое окно Import Flex Project, изображенное на рис. 2.2, которое позволит импортировать заархивированный проект в рабочее пространство Flex Builder. Выберите опцию Archive file в разделе Import project from, укажите путь к только что скачанному архиву с проектом, нажав на кнопку Browse. Оставьте указанный по умолчанию адрес, куда будет сохранен проект, и нажмите Finish.

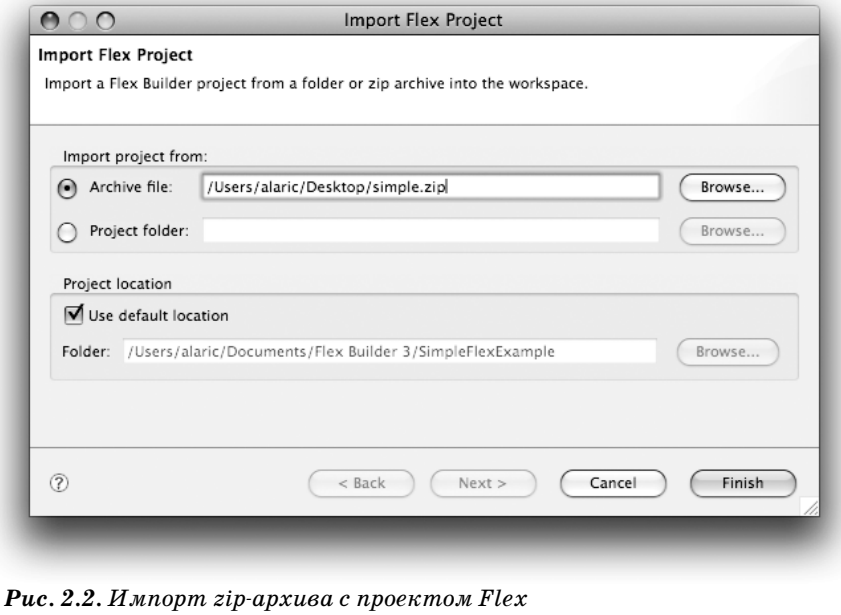

***Рис. 2.2.** Импорт zip-архива с проектом Flex*

Внимание

Если ваш броузер разархивировал скачанный файл или вы сделали это самостоятельно, выбор Archive в окне Import Flex Project не сработает. В этом случае вам придется воспользоваться опцией Import Existing Projects в диалоговом

окне Workspace. Об этой возможности мы поговорим немного позже в разделе «Импорт существующего проекта».

Буквально через несколько мгновений ваш проект отобразится на панели Flex Navigator. Щелкните по стрелке слева от названия проекта. Раскроется структура содержимого каталога, состоящая из нескольких подкаталогов, в одном из которых находится сам файл с приложением. В каком из них, спросите вы? По обыкновению программисты называют папку с исходным кодом «source» или «src». Так что смело открывайте папку src, как показано на рис. 2.3, и вы увидите файл приложения *SimpleFlexExample.mxml*.

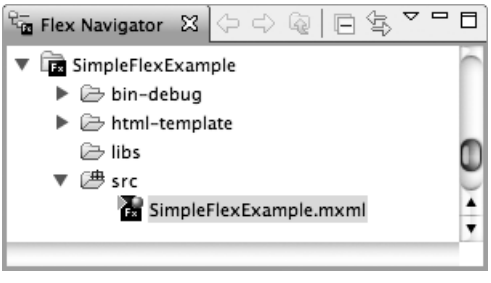

Рис. 2.3. *Ваш первый проект*

После того как вы скачали проект и импортировали его в свое рабочее пространство, самое время взглянуть на исходный код. Двойной щелчок по файлу *SimpleFlexExample.mxml* во Flex Builder откроет его в режиме редактирования Design, при этом отобразится точный вид структуры приложения. Увидеть исходный код можно, переключившись в режим Source. Для этого щелкните по соответствующей вкладке на панели выбора режима редактирования Source/Design, как показано на рис. 2.4. Подробности оставим на потом, а сейчас давайте запустим приложение!

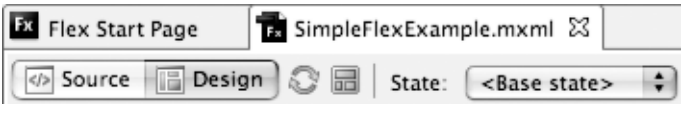

Рис. 2.4. *Переключение между режимами Source и Design*

Маленькая зеленая кнопка: запуск приложения

Теперь файл *SimpleFlexExample.mxml* открыт в Flex Builder, и можно запустить приложение через броузер, выбрав Run→Run SimpleFlexSample через главное меню или щелкнув по кнопке с изображением зеленой

стрелки в панели инструментов, расположенной слева от кнопки с зеленым жучком (используемой для отладки) (рис. 2.5). Через несколько мгновений откроется окно вашего броузера с HTML-страницей, на которой будет запущено ваше первое Flex-приложение.

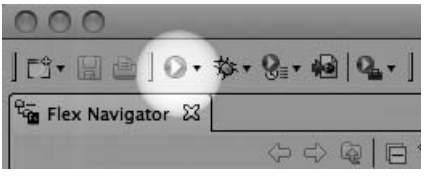

Рис. 2.5. *Кнопка запуска приложения*

Да здравствует проект!

Итак, мы запустили первый импортированный проект, а теперь давайте удалим его. Как, просто так взять и выбросить результат упорного труда?! Нет причин для беспокойства – вы можете вернуть ваш проект в любое время. Дело в том, что управление проектами во Flex Builder подчинено следующим правилам: при удалении проекта он более не показывается в рабочем пространстве, но это не означает, что исходный код был также удален: соответственно, он хранится на вашем жестком диске, но не отображается во Flex Builder. Следуйте моим указанием, и вы поймете, в чем тут дело.

Прежде всего выберите проект во Flex Navigator (основную папку SimpleFlexExample, а не отдельный файл или подпапку) Затем выберите Edit→ Delete в меню. При этом откроется диалоговое окно, в котором требуется выбрать, нужно ли удалить содержимое проекта или нет (рис. 2.6). Удостоверьтесь, что выбран вариант Do not delete contents (который должен быть значением по умолчанию), чтобы оставить файлы проекта на своем компьютере.

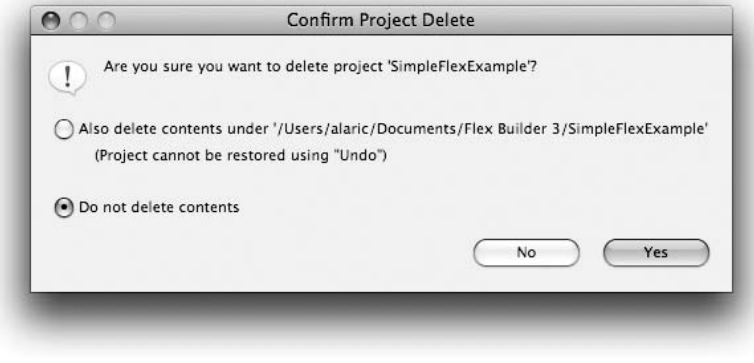

Рис. 2.6. *Удаление проекта*

Примечание

Чаще всего я буду говорить о способе выполнения различных команд через строку меню, но большую часть часто используемых команд можно выполнить и через контекстное меню во Flex Builder. К примеру, для удаления проекта можно щелкнуть правой кнопкой мыши (или щелчок с нажатой клавишей Control на Mac OS X) по содержащей его папке и выбрать пункт Delete в открывшемся контекстном меню.

А теперь проверьте папку с проектом на вашем жестком диске (обычно это каталог Flex Builder 3 в папке Мои документы или другой каталог, куда вы его поместили) – все файлы проекта должны там остаться, и структура подкаталогов должна соответствовать отображаемой ранее на панели Flex Navigator.

В чем же смысл такой операции? Она дает вам представление об особенностях проекта в Flex Builder. Вы не можете работать с лежащим где-либо на вашем жестком диске Flex-кодом во Flex Builder, если не создадите соответствующий проект. Если вы откроете в проводнике папку src и дважды щелкните мышью по лежащему в ней файлу *SimpleFlexExample.mxml*, он будет открыт текстовым редактором; но, возможно, ваш компьютер не сможет определить, как обработать этот файл. Если вы предполагали, что он откроется во Flex Builder, то, к сожалению, ошиблись. Если вы хотите использовать этот файл, он должен быть частью проекта – тогда Flex Builder будет знать, что с ним делать. Чтобы сделать это прямо сейчас, снова осуществим импорт проекта.

Ваше рабочее пространство

Говоря о Flex Builder, я часто буду упоминать **рабочее пространство**. Рабочее пространство – это папка в вашей системе, которую вы выбрали для хранения проектов (в ней также содержатся настройки ваших предпочтений). На практике у вас может быть более одного рабочего пространства, но при установке Flex вы определяете используемое в качестве основного, и по умолчанию оно находится в папке с вашими документами и называется Flex Builder 3. При желании можно создать другое рабочее пространство, выбрав File→Switch Workspace→Other.

Импорт существующего проекта

Не хочу показаться занудным, но при работе с Flex вам придется бесчисленное количество раз выполнять импорт проектов, поэтому будет лучше сразу привыкнуть к этому процессу. Вы, возможно, думаете, что уже знаете, как импортировать проект, но это не совсем так. Ранее

вы импортировали zip-архив с файлами, которые теперь лежат в разархивированном виде. Для импортирования этого набора файлов придется немного изменить порядок действий.

Как и в прошлый раз, выберите команду File→Import→Flex Project, но теперь выберите опцию Project folder в разделе Import project from диалогового окна (рис. 2.7). Это позволит импортировать не собранные в архив файлы для создания проекта.

Нажмите Browse и укажите путь к выбранной в прошлый раз папке SimpleFlexExample, в которой находится сам проект и вспомогательные файлы. Нажмите Finish для завершения процесса импорта.

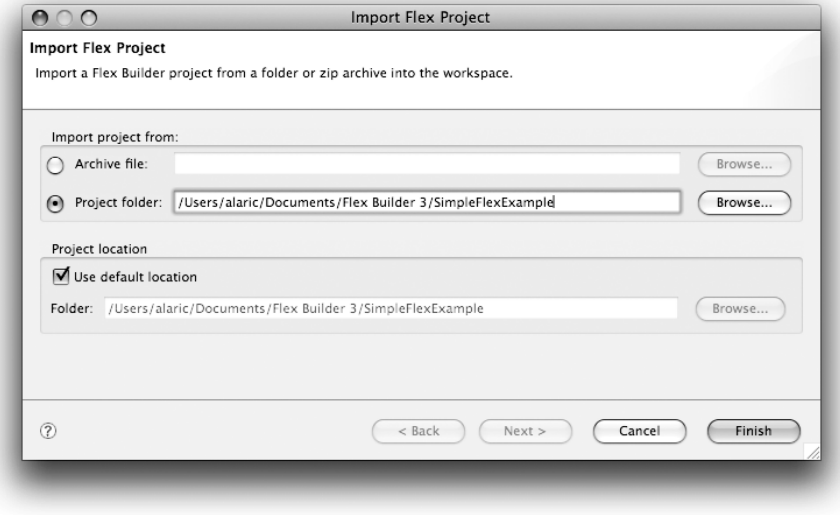

Рис. 2.7. *Импорт проекта из папки*

С чистого листа: создание нового проекта Flex

Безусловно, при разработке во Flex чаще всего вам придется создавать новые проекты. Поэтому сейчас я покажу вам, как это делается, а в дальнейшем вы будете работать с созданным проектом для получения практических навыков.

Примечание

Вы, наверное, обратили внимание на флажок Copy projects into workspace. Его полезно установить, если ваш проект лежит, например, на рабочем столе или в каком-то другом месте, куда вы его поместили, и вы хотите одновременно импортировать его в Flex Builder и поместить его в ваше рабочее пространство. Довольно удобно хранить все проекты в одном месте: это поможет избежать нечаянного удаления нужных файлов (к примеру, при очистке рабочего стола).

Чтобы создать новый проект, выберите меню File и далее New→Flex Project. При этом откроется диалоговое окно, в котором требуется задать настройки вашего проекта. Название проекта укажите в поле Project name. В данном случае это будет `HelloWorld`.

В появившемся диалоговом окне вы увидите три раздела: Project location, Application type и Server technology. Project location изменяет расположение файлов и папок проекта на вашем компьютере. По умолчанию указан путь к папке вашего рабочего пространства, но вы можете указать свой собственный, сняв флажок напротив Use default location.

Внимание

В названии проекта не должно быть пробелов и других необычных символов – имя может состоять только из букв, цифр, знака $ и символа подчеркивания. Впрочем, об этом можно не беспокоиться – система уведомлений Eclipse работает отлично, и вам не удастся сделать ничего «противозаконного».

Во втором разделе Application type вам предстоит сделать выбор между разработкой веб-приложения или настольного приложения, запускаемого с помощью Adobe AIR. В данном примере мы хотим, чтобы приложение запускалось через броузер, поэтому оставьте настройку web application, установленную по умолчанию.

Примечание

Любой проект располагается в определенной папке, название которой обычно совпадает с названием самого проекта. Однако это не обязательное условие: вы спокойно можете поместить проект HelloWorld в папку Hello World Project и даже хранить его в какой-нибудь папке прямо на рабочем столе. Лично мне нравится простота и организованность, поэтому, как правило, я оставляю настройки по умолчанию.

В третьем разделе Server Technologies нужно указать серверные технологии, которые вы намерены использовать. Эту полезную опцию необходимо использовать в том случае, если вы будете подключать к вашему приложению базу данных, файл XML или иные данные, находящиеся на удаленном сервере. В нашем случае просто оставьте значение None, т. к. мы не собираемся использовать какие-либо данные, расположенные на удаленном компьютере. В дальнейшем, если у вас появится необходимость установить связь между вашим приложением и сервером ColdFusion, PHP или LiveCycle Data Services, эта настройка вам пригодится. В главе 10 я подробнее остановлюсь на различных сервисах работы с данными и способах подключения к ним. Вы, наверное, обратите внимание на расположенные в нижней части окна кнопки Next, Cancel и Finish. Пока мы будем работать с простыми примерами, поэтому просто выберите Finish, оставив остальные настройки без изменений.

Adobe AIR

В последнее время Adobe AIR получает немало положительных отзывов, и это не случайно. Благодаря AIR появляется возможность превратить веб-приложения в полноценные настольные приложения со всеми характерными для них функциями, включая наличие иконки в меню Пуск и на панели задач, поддержку возможности перетаскивания объектов при помощи мыши и т. д. Flex – не только мощная платформа, позволяющая создавать программы, воспроизводимые в среде броузера, но и отличное решение для разработки настольных приложений, исполняемых в среде AIR. С Flex Builder изменить тип приложения – веб-приложение или настольное приложение – не сложнее, чем щелкнуть по выключателю. Я еще буду упоминать о возможностях AIR в этой книге.

Примечание

Настройки проекта можно изменить в любое время, выбрав пункт Project→Properties в меню Flex Builder. Рассмотренное только что диалоговое окно предназначено помочь вам в самом начале при создании проекта.

Структура Flex-проекта

Теперь, когда вы освоили различные способы создания проектов, самое время взглянуть на структуру каталога и разобраться, для чего же нужны все эти папки. (Могу поспорить, вы первым делом заглянете в папку bin-debug.) Традиционно в каждый Flex-проект входят папки с названиями bin-debug, html-template, lib и src. В табл. 2.1 подробно описано, для чего предназначена каждая из папок, а на рис. 2.8 представлена их структура, отображаемая на панели Flex Navigator.

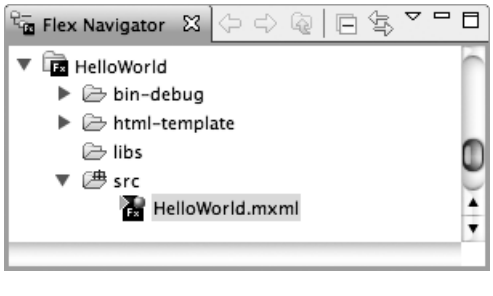

Рис. 2.8. *Новый проект*

Таблица 2.1. Структура папки с проектом

Имя папки	Назначение	Описание
bin-debug	Скомпилированный код (англ. binary debug)	Содержит скомпилированный код (файл SWF). При разработке для веб содержит также HTML-контейнер и вспомогательные файлы. При разработке AIR-приложений – файл описания приложения (Application Descriptor File).
html-template	шаблон html-страницы (англ. html-template)	Содержит HTML-шаблон, на основе которого генерируется файл HTML-контейнера (только для веб-приложений).
libs	библиотеки (англ. libraries)	Содержит файлы дополнительных библиотек.
src	исходный код (англ. source)	Содержит исходный код в виде файлов с расширением .mxtml или .as, а также файл описания приложения (Application Descriptor File) (только для AIR-приложений).

Примечание

Названия папок могут быть и другими. Вы вполне можете переименовать папку src в папку source или вовсе хранить основной MXML-файл в корне каталога. Такие изменения можно сделать, нажав кнопку Next при создании нового приложения в диалоговом окне New Project или позже через меню Properties (которое доступно при щелчке правой кнопкой мыши по проекту). Изменить названия папок можно и в разделе Flex Build Path, изменив названия полей Main Source Folder и Output folder.

Заключение

Итак, в этой главе мы рассмотрели основные вопросы, необходимые для начала работы, и теперь можно приступать к разработке. Вы узнали об использовании проектов в Flex Builder, научились открывать и запускать приложения и имеете представление о структуре типичного Flex-приложения. Теперь вы во всеоружии и можете начать работу над собственным проектом!

В этой главе:

- Чистая доска: ваш рабочий холст
- Добавление компонентов в приложение
- Перемещение компонентов
- Основные компоненты Flex
- Изменение свойств компонентов в режиме Design

3

Работа в режиме Design

Режим Design позволяет редактировать приложение по принципу WYSIWYG[1]. Лучше всего начинать работу именно в этом режиме, даже если вы опытный программист или дизайнер. В вашем распоряжении прекрасный набор визуальных компонентов, которые легко перетаскивать мышью в нужное место при построении интерфейса приложения. Все свойства этих компонентов можно изменять в визуальном режиме, а наиболее часто используемые из свойств доступны наиболее просто. В этой главе вы познакомитесь с режимом Design и начнете создание своего приложения.

Чистая доска: ваш рабочий холст

Вернемся к проекту HelloWorld и откроем основной файл приложения (с расширением *.mxml*) в режиме Design. Перед вами появится пустой экран, заполненный фоном голубоватого оттенка, что сразу указывает на использование стиля по умолчанию для всех приложений Flex. При работе в режиме Design в левой части окна под панелью Flex Navigator и рядом с панелью Outline должна отображаться панель Components. Если вы ее не видите, ее можно вывести на экран через меню Window. Обратите внимание, что при переключении в режим Source данная панель отображаться не будет, т. к. в этом режиме вы пишете MXML-код для создания компонентов.

---

[1] WYSIWYG (англ. what-you-see-is-what-you-get) – режим точного отображения, буквально «что видишь на экране, то и получишь при печати». – *Прим. перев.*

Для вывода на экран панели Components выберите соответствующий пункт в меню Window – он находится в самом начале списка, т. к. это одна из наиболее часто используемых панелей во Flex Builder. Эта панель содержит перечень всех компонентов пользовательского интерфейса, из которых можно построить любой интерфейс.

Добавление компонентов в приложение

Для добавления компонентов в ваше приложение можно просто перетащить их мышью с панели Components в любое место на холсте, как показано на рис. 3.1. Расположенные на нем компоненты можно перемещать и задавать их свойства в режиме визуального редактирования. Попробуем добавить в приложение кнопку. На панели Components этот элемент находится в самом верху раскрывающегося списка, озаглавленного Controls.

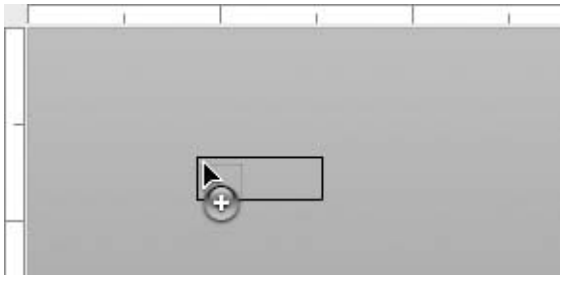

Рис. 3.1. *Перемещение компонента на сцену*

Холст ≈ сцена ≈ приложение

Термины *холст*, *сцена* и *контейнер приложения* употребляются в данной книге в качестве синонимов. Голубой фон, отображаемый при редактировании приложения в режиме Design, – не что иное, как визуальный контейнер, служащий основой всех Flex-приложений. Если вам привычнее термины из сферы дизайна, это холст, т. е. чистая «доска», на которой вы располагаете визуальные элементы. Обратите внимание – Canvas (холст) (с большой буквы) – это отдельный вид контейнера, в отличие от более общего понятия canvas (холст). Я также буду называть холст более старым термином, употребляемым во Flash, – сценой в противоположность объекту Stage (сцена), обозначающему класс, на котором базируются все Flash и Flex-приложения. Термин «сцена» было бы уместнее употреблять по отношению к созданию анимационных и видеороликов во Flash IDE, но что поделаешь, привычка – вторая натура.

Перемещение компонентов

Находящийся на холсте компонент (в нашем случае это кнопка) можно перемещать по всей его области. Вы, вероятно, заметите, что при приближении элемента к краю сцены появляются вспомогательные линии, позволяющие выравнивать его положение. В сфере дизайна эти линии называют *направляющими*, и вы наверняка встречали их, работая в Photoshop или других графических редакторах. Это визуальный инструмент, осуществляющий привязку, т. е. выравнивание компонента относительно краев холста или по отношению к окружающим его компонентам. При перемещении компонента к краю сцены привязка обычно обеспечивает отступ от него в размере 10 пикселов. При приближении к другим компонентам также могут появиться направляющие, помогающие сохранить между элементами расстояние в 10 или 20 пикселов. Благодаря направляющим вы можете легко согласовывать расположение компонента относительно других составляющих интерфейса, даже если он находится от них на существенном расстоянии. Этот метод полезно использовать, если вы хотите быстро и аккуратно расположить свои компоненты, чтобы приложение выглядело симметрично и привлекательно.

Примечание

Если при перемещении компонентов вы не хотите пользоваться методом привязки, удерживайте нажатой клавишу Alt (Option для Mac). Это отменит появление направляющих, и вы сможете располагать элементы с точностью до пиксела. Чтобы полностью выключить функцию привязки, снимите галочку напротив пункта Enable Snapping в меню Design.

Основные компоненты Flex

Компонент – это фрагмент кода, многократно используемый в качестве инструмента создания приложения. Предполагается, что компоненты становятся элементами готовой системы, и скорее всего вы будете использовать множество из них в своих приложениях. Одним из преимуществ Flex является входящий в его состав богатый набор компонентов. Но несмотря на то, что использование компонентов экономит массу времени и не требует много специальных знаний, на более глубокое освоение техники их применения и конфигурации потребуется некоторое время, которое не будет потрачено зря – ведь это ваши первые шаги на пути к вершинам Flex-мастерства. К счастью, для начала работы будет вполне достаточно общих сведений, а овладение более сложными приемами придет вместе с опытом.

Ниже я приведу список наиболее распространенных компонентов.

Элементы управления

Элементы управления – основные визуальные компоненты интерфейса пользователя, такие как кнопки или текст. Такое название объясняется тем, что они предназначаются для *управления* приложением при работе пользователя. Здесь можно провести аналогию с телевизионным пультом управления, позволяющим переключать каналы или регулировать звук путем нажатия на кнопки.

Скорее всего, вы часто будете использовать следующие элементы управления:

Button

Здесь все ясно: этот элемент выглядит и функционирует как настоящая кнопка. Ее можно использовать и в качестве переключателя благодаря свойству `toggle`, а ее состояние регулируется свойством `selected`.

CheckBox

Напоминает кнопку, но выполняет функцию переключателя.

ComboBox

Компонент, состоящий из списка и кнопки, представляет собой компактный выпадающий или всплывающий список из нескольких пунктов. Свойства `selectedItem` (объект) или `selectedIndex` (целое число) позволяют получить или установить выбраннный объект (или его индекс).

Image

Этот элемент позволяет подключать внешние ресурсы. Возможные форматы – GIF, JPEG, PNG, а также SWF-файлы. Ссылка на изображение должна быть указана в свойстве `source`.

Label

Самое подходящее решение для создания простой однострочной метки.

List

Данный элемент отображает список, состоящий из нескольких пунктов. Если для отображения всех пунктов не хватает места, появляются полосы прокрутки.

ProgressBar

Этот элемент хорошо подходит для отображения хода таких процесов, как загрузка файла. Свяжите его с элементом Image, указав в качестве значения параметра `source` элемента *ProgressBar* путь к изображению, и вы увидите ход загрузки изображения.

RadioButton

Данный элемент, напоминающий CheckBox, обычно используется в группе переключателей, причем установка одного из них сбрасывает выбор любого другого. Проще всего перетащить на сцену RadioButtonGroup, при этом откроется диалоговое окно, с помощью которого можно создать группу переключателей в визуальном режиме.

Text

Этот элемент используется для размещения фрагмента текста без полос прокрутки. Размер элемента будет соответствующим образом изменен, чтобы вместить весь текст.

TextArea

Используйте этот элемент, если объем текста может превысить доступное пространство. При необходимости появятся полосы прокрутки.

TextInput

Предназначается для ввода строки текста. Свойство `text` дает возможность получить или задать значение отображаемой строки (это касается также всех текстовых компонентов, включая Text, TextArea и RichTextEditor).

Контейнеры размещения

Контейнеры размещения, или просто *контейнеры,* – визуальные компоненты, контролирующие выравнивание элементов приложения или их взаимное расположение. Иногда они отображаются визуально (например, строка заголовка компонента Panel), но чаще всего они лишь изменяют размещение элементов (или даже других контейнеров) определенным образом, оставаясь при этом невидимыми.

Application

О нем вы можете вообще не вспоминать. Этот контейнер по умолчанию служит базой любого Flex-приложения. Он обладает совершенно магическими свойствами, а именно, способностью стать основой для чего угодно. По умолчанию компоненты интерфейса можно располагать в любом месте внутри контейнера, но вы можете также выравнивать их по вертикали или по горизонтали. Этот контейнер обладает и другими функциями, такими как поддержка индикатора загрузки приложения (который вы наверняка не раз наблюдали при запуске Flex-приложения в своем броузере) и наличие специальных опций для настройки вашего приложения.

Canvas

Контейнер без определенной схемы выравнивания. Для размещения элемента в конкретном месте необходимо задать значения его координат по осям x и y.

Form

Этот контейнер позволяет создать подобие HTML-формы. При использовании совместно с компонентами FormItem (о которых речь пойдет ниже), располагает их по вертикали, при этом поля формы выравниваются по левому краю.

FormItem

Этот контейнер предназначен для таких элементов, как TextInput. При установке значения свойства label контейнера последнему присваивается метка. Несколько контейнеров FormItem, помещенных в контейнер Form, выравниваются наподобие привычных HTML-форм.

HBox

Выравнивает элементы по горизонтали.

Panel

Этот компонент напоминает окно со строкой заголовка. При добавлении Panel в режиме Design по умолчанию размещение внутренних элементов будет «абсолютным», т. е. определенным заданными координатами по оси x и y (как и при использовании Canvas). Но выравнивание можно также изменить на горизонтальное или вертикальное, что означает возможность полного управления расположением дочерних элементов.

VBox

Выравнивает элементы по вертикали.

Навигаторы

Навигаторы – комбинированный тип визуальных компонентов, представляющий собой нечто вроде сочетания контейнера размещения и элемента управления. Они предназначены для создания набора контейнеров, из которого видим в каждый момент только один. Благодаря наличию таких компонентов значительно упрощается процесс создания модульных интерфейсов.

Accordion

Похож на описанный ниже Tab Navigator, за тем исключением, что он располагает контейнеры вертикально и добавляет анима-

ционный эффект при выборе одного из разделов. Обычно используется совместно с группой контейнеров Form для разделения большой формы, требующей прокрутки, на несколько секций.

Tab Navigator

Используется с набором таких контейнеров, как HBox или Panel, обеспечивая одновременную видимость только одного из них. При этом создаются закладки, похожие на папки с файлами, с метками, соответствующими значению свойства label вложенных контейнеров. Видимым становится только контейнер, соответствующий выбранной пользователем закладке, остальные контейнеры при этом скрыты.

Существует также ряд компонентов, называемых невизуальными, которые выполняют определенные функции, но не отображаются визуально. Примером такого, так сказать, безликого компонента может послужить элемент, создающий подключение к удаленным данным. Такие компоненты нельзя добавлять и модифицировать в режиме Design – их код должен быть написан вручную. Написание кода в режиме Source мы обсудим в следующей главе, а в главе 8 впервые попробуем использовать невизуальный компонент.

Flex Component Explorer

Существует замечательное приложение, служащее путеводителем по миру основных компонентов Flex и раскрывающее все особенности их использования. Оно называется Flex Component Explorer и доступно по адресу *http://examples.adobe.com/flex3/componentexplorer/explorer.html*.

Изменение свойств компонентов в режиме Design

Такие свойства компонентов, как метка или размеры, можно изменять прямо в режиме Design. Сейчас вы можете попробовать поменять свойство label вашей кнопки, дважды щелкнув по ней мышью. Появится окошко для ввода текста, позволяющее это сделать. Потянув за появляющиеся у краев элемента стрелки, можно с легкостью варьировать размер кнопки. Что касается голубых пунктирных линий, окружающих компонент, они предназначены для создания взаимосвязанного расположения компонентов, о котором мы немного поговорим в главе 8.

Все под рукой: панель Properties

Перед тем как начать изменять свойства компонента, нам предстоит изучить возможности еще одной панели, называемой Flex Properties. Она должна отображаться в правой части экрана, если вы не переместили или не закрыли ее. В противном случае для вывода панели на экран выберите меню Window→Flex Properties. Обратите внимание, что при работе в режиме Source данная панель отсутствует, ведь она вам не понадобится при написании кода вручную, правда? Но в режиме Design это ваш верный помощник.

Панель Properties изменяет свой вид при выборе различных элементов сцены. Например, выделите кнопку, щелкнув по ней мышью, и вы увидите, что заголовок панели изменится на `mx:button`, а на самой панели отобразится набор изменяемых свойств для данной кнопки.

Standard view (Стандартный вид)

По умолчанию панель Properties отображается в режиме Standard view, при этом для каждого выбранного элемента представлен список основных присущих ему свойств (рис. 3.2). Это особенно удобно для изменения общих свойств компонента, таких как label, id, width и height, и совершенно незаменимо при смене таких параметров оформления компонента, как шрифт, начертание (жирный, курсив, подчеркивание и т. д.) и его прозрачность.

Category view (Вид по категориям)

Список свойств на панели Properties может быть упорядочен и по категориям (рис. 3.3). При этом отображается древовидная структура всех свойств компонента, отчасти напоминающая Standard view, где они собраны в такие группы, как Common, Layout и Style. Но в отличие от него, в Category view доступны все возможные свойства элемента. Этот вид удобен для быстрого доступа к конкретному свойству компонента.

Alphabetical View (Сортировка по алфавиту)

Третий и последний вариант отображения свойств на панели Properties – Alphabetical view. Как и в Category view, в нем представлены все без исключения свойства компонента. Однако в отличие от рассмотренных способов отображения, они не подчиняются какой-либо классификации. Это простой одноуровневый список, в котором свойства представлены в алфавитном порядке. Такой вид удобно использовать, если вы знаете имя свойства целиком или по крайней мере несколько начальных букв. Также это дает возможность обзора всех имеющихся свойств данного компонента.

Попробуйте поместить несколько компонентов на холст и поэкспериментировать с их расположением. Поменяйте некоторые свойства, например `label`, `width` и `color`, – это поможет привыкнуть к работе с панелью.

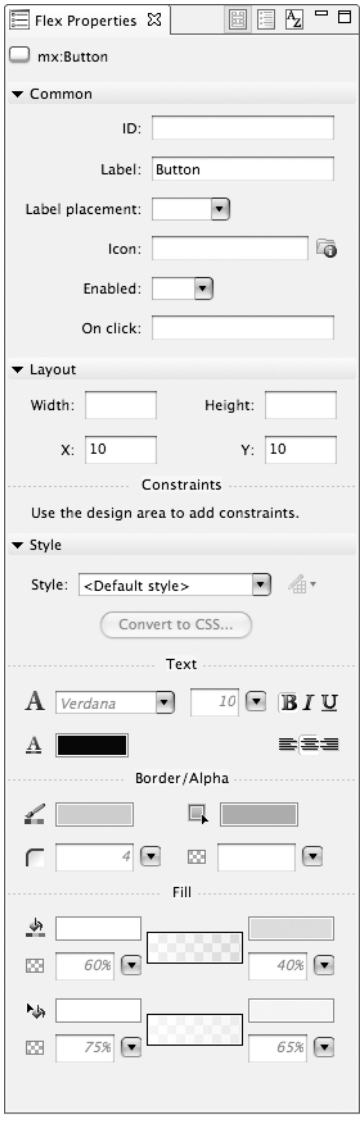

Рис. 3.2. *Панель Flex Properties (Standard view)*

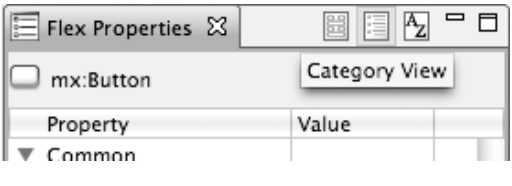

Рис. 3.3. *Панель Flex Properties (Category view)*

Примечание

Во Flex Builder для Windows есть прекрасный способ быстрого доступа к определенному свойству в режиме отображения `Alphabetical View`. Введите первые несколько букв названия свойства, и список будет автоматически прокручен до первого совпадения. Такой способ подходит и при работе с другими панелями Flex Builder, где есть длинные списки. К сожалению, данная функция недоступна в версии для Mac.

Общие свойства компонентов

Основой всех визуальных компонентов (редактируемых в режиме Design) является базовый компонент UIComponent. Он обладает свойствами, регулирующими видимость, размеры, реакцию при взаимодействии с пользователем при помощи мыши и клавиатуры, фокусировку и другие полезные параметры. Кнопки, поля для ввода текста и контейнеры являются расширенными вариациями этого основного компонента, т. е. к унаследованной от него функциональности добавляются характерные только для данного конкретного компонента функции. Поэтому они обладают также всеми свойствами основного компонента. Таким образом, если вы познакомились со свойством базового компонента, то это знание можно с успехом применять к другим компонентам.

В качестве примера возьмем свойство `width` элемента `Button`. Несложно догадаться, что изменение этого свойства влечет за собой соответствующее изменение ширины кнопки. Данным свойством обладают и другие визуальные компоненты, поэтому, если вы понимаете принцип его использования в случае с кнопкой, будет легко применить эти знания и при работе с другими компонентами.

Ниже представлен список наиболее часто используемых свойств основных компонентов с примерами использования.

`id`

`id` (сокращенно от identifier (англ. идентификатор) – очень важное свойство, присваивающее имя определенному компоненту. Если в вашем приложении есть две кнопки, то назвав их button1 и button2, вы будете без труда различать их в вашем коде. Еще лучше дать элементам описательные имена, связанные с выполняемой ими функцией в приложении (например, submitButton (кнопка отправки) и refreshButton (кнопка обновления). При необходимости Flex определит значение данного свойства автоматически, но гораздо эффективнее сделать это самостоятельно.

```
<mx:Button id="submitButton"/>
```

`x`

Числовое свойство, определяющее смещение элемента вправо по отношению к родительскому элементу. То есть при установке значе-

Что такое API? Или SDK?

API (Application Programming Interface, программный интерфейс приложения) – это обобщенный термин, означающий набор функций, предоставляемый для использования программистам. Вы могли слышать его, например, при упоминании такого веб-сервиса, как Yahoo! Search API, позволяющего запрашивать поисковую базу Yahoo! Термин API применяется, вообще говоря, по отношению к любым методам, предоставляющим доступ к функциям программы или языка программирования. Поэтому, когда речь идет о Flex, этот термин будет часто употребляться при разговоре об использовании компонентов.

Вы также наверняка слышали термин SDK (Software Development Kit, набор инструментов разработки программного обеспечения). Как правило, он обозначает набор компонентов или инструментов, помогающих разработчику при создании приложения для определенной платформы (или даже делающих разработку возможной в принципе). При загрузке Flex Builder вместе со средой разработки вы скачиваете также Flex SDK, т. е. библиотеку компонентов, инструмент для компиляции и т. д., благодаря чему разработка Flex-приложений становится возможной.

Разберем занятное предложение: «Я создал Flex API для Yahoo! Search API, который включен в Yahoo! Search SDK». Это означает: «Я создал более удобный метод подключения к результатам поиска Yahoo! Search через Flex-приложение, которое стало составной частью семейства инструментов для использования Yahoo! Search». Вы можете скачать Flex API в составе Yahoo! Search SDK (в котором также представлены API для других языков, таких как Java и ColdFusion) с *http://developer.yahoo.com/download*.

ния данного свойства 20 пикселов для элемента, расположенного внутри контейнера, будет сделан соответствующий отступ от левого края этого контейнера.

```
<mx:Button x="20"/>
```

y

Числовое свойство, действие которого аналогично свойству x с тем лишь отличием, что оно регулирует расположение элемента по вертикали (отступ отсчитывается от верхнего края родительского элемента). Более подробно о свойствах x и y можно прочитать в главе 4 (врезка «X и Y: Несколько слов о системе координат»).

```
<mx:Button y="10"/>
```

`visible`

Свойство, управляющее видимостью элемента на сцене. При присваивании свойству значения `false` элемент становится невидимым, однако будет занимать некоторую область пространства. К примеру, если в контейнере Hbox расположены четыре кнопки, выравненные по горизонтали, и я сделаю вторую из них невидимой, соседние кнопки останутся на своих местах, а на месте невидимой кпопки останется пустое пространство (рис. 3.4).

```
<mx:HBox>
  <mx:Button label="One"/>
  <mx:Button label="Two" visible="false"/>
  <mx:Button label="Three"/>
  <mx:Button label="Four"/>
</mx:HBox>
```

Рис. 3.4. *Заметьте, кнопка Two невидима*

`includeInLayout`

Установите значение этого свойства `false`, и контейнер не будет отображать элемент, причем при этом не будет возникать пустого пространства. Это свойство удобно использовать, если вы хотите сделать какой-либо элемент невидимым, при этом не меняя расположение элементов (рис. 3.5)

```
<mx:HBox>
  <mx:Button label="One"/>
  <mx:Button label="Two" visible="false" includeInLayout="false"/>
  <mx:Button label="Three"/>
  <mx:Button label="Four"/>
</mx:HBox>
```

Рис. 3.5. *Кнопка Two все же существует, хотя и невидима и не занимает места*

`toolTip`

Благодаря этому свойству при наведении курсора мыши на элемент на экране появляется небольшая всплывающая подсказка, указывающая на его назначение (рис. 3.6). Это свойство легко использовать, и им обладают все компоненты пользовательского интерфейса Flex.

```
<mx:Button toolTip="Click Me!"/>
```

Рис. 3.6. *Свойство toolTip на примере кнопки*

`label`

Этим свойством обладает большинство компонентов. Очевидно, что для кнопки – это отображаемый на ней текст, похожим образом дело обстоит и с компонентом CheckBox. Контейнерам также присуще это свойство. К примеру, значение данного свойства контейнера FormItem, входящего в состав Form, будет использоваться для описания (метки) дочернего элемента, такого как TextInput (рис. 3.7). При использовании с другими контейнерами имеет смысл включить их в навигатор. То есть при задании данного свойства для компонента внутри TabNavigator оно будет отображаться в названии соответствующей ему закладки.

```
<mx:CheckBox label="I have read the terms and conditions"/>
<mx:Button label="Submit"/>
```

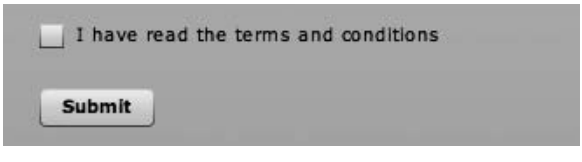

Рис. 3.7. *Свойство label флажка и кнопки*

Примечание

Обратите внимание, что существует элемент управления Label, предназначенный для вставки текстовых строк в ваше приложение. Для изменения отображаемого текста используется свойство `text`, а не `label`.

`text`

Используется для изменения отображаемого текста в таких компонентах, как TextInput, TextArea и Label (рис. 3.8).

```
<mx:Label text="Enter Your Name:"/>
<mx:TextInput text="Alaric"/>
```

Рис. 3.8. *Оба компонента используют свойство text*

`alpha`

`alpha` – сокращенно от alpha channel (альфа-канал). Значением может быть число от 0 до 1, определяющее степень прозрачности элемента. При установке значения меньше единицы фон, на котором расположен элемент, будет просвечивать сквозь него. Значение расчитывается в процентах, т. е. .9 означает 90-процентную непрозрачность, .45 – 45-процентную и т. д. На рис. 3.9 приведен пример элемента со значением свойства `alpha` .5, при этом сквозь полупрозрачный элемент проступает цветной фон.

```
<mx:TextArea alpha="0.5" text="The alpha of this TextArea control is set
to .5"/>
```

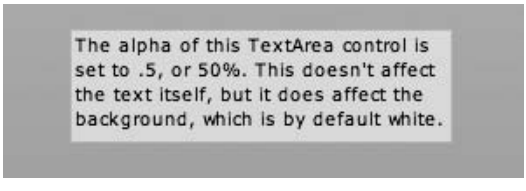

Рис. 3.9. *Значение свойства alpha элемента TextArea – 50%*

`enabled`

Значение `false` данного свойства обеспечивает блокирование компонента, что, как правило, проявляется в его отображении в серых тонах и отсутствии какой-либо реакции при наведении на него курсора мыши и т. п. (рис. 3.10). Особенно эффективно использование данного свойства для кнопки, которая должна оставаться неактивной до определенного момента, но его также можно использовать с контейнерами и их дочерними элементами. Можно установить значение `false` для всего компонента Panel, при этом все его элементы также будут заблокированы.

```
<mx:Panel title="Enabled Panel" enabled="true">
  <mx:ColorPicker/>
  <mx:NumericStepper/>
  <mx:Button label="Button"/>
  <mx:CheckBox label="Checkbox"/>
  <mx:DateField/>
  <mx:ComboBox/>
</mx:Panel>
<mx:Panel title="Disabled Panel" enabled="false">
  <mx:ColorPicker/>
  <mx:NumericStepper/>
  <mx:Button label="Button"/>
  <mx:CheckBox label="Checkbox"/>
  <mx:DateField/>
  <mx:ComboBox/>
</mx:Panel>
```

Рис. 3.10. *Две одинаковые панели Panel, одна активна, другая заблокирована*

`source`

Позволяет указать ссылку на внешний источник файла в таких элементах, как Image и подобных. Совместное использование элементов Image и ProgressBar позволяет отобразить ход процесса загрузки изображения. Для Image свойство `source` определяет путь к источнику изображения, для ProgressBar – указывает на объект, загрузку которого необходимо отслеживать.

```
<mx:ProgressBar source="{photo}"/>
<mx:Image id="photo" source="http://greenlike.com/photos/lydia.jpg" />
```

Рис. 3.11. *Отличный пример совместного использования элементов Image и ProgressBar*

Скажем прямо, некоторые из этих свойств придется вызубрить, но при работе в режиме Design все основные опции у вас под рукой. Кроме того, благодаря продуманному механизму действия и множеству общих черт принципы использования свойств просты и логичны.

Создание интерфейса пользователя

Познакомившись с особенностями работы в режиме Design, вы вполне можете начать создание своего проекта HelloWorld. Если вы успели добавить какие-либо компоненты на холст, удалите их, и перед вами снова чистое полотно. Компоненты можно удалить по одному, выделяя каждый мышью и нажимая клавишу Delete, или же выбрать все сразу и нажать Delete. Еще один способ: выберите Edit→Select All или нажмите Ctrl + A (⌘+A на Mac OS X) – это выделит все элементы, – а затем нажмите клавишу Delete.

Давайте создадим простое приложение, используя основные компоненты интерфейса.

1. Перетащите на холст компонент Panel. Так как он является контейнером, то расположен в разделе Layout панели Components.
2. Придумайте заголовок для Panel и введите его на панели Properties или дважды щелкнув по строке заголовка контейнера. Мое название будет Howdy Ya'll[1], но вы можете ввести любое приветствие, которое вам по душе.
3. Поместите компонент Label внутри элемента Panel. Последний является контейнером, что позволяет размещать внутри него другие элементы, как и внутри основного компонента Application (который

Flex говорит на вашем языке

Я не просто так сказал, что вы можете ввести любое приветствие, которое вам нравится, даже если вы здороваетесь словами Olaד你好, или こんにちは. Дело в том, что Flex поддерживает кодировку UTF-8, позволяя свободно вставлять символы большинства национальных алфавитов как в режиме Design, так и в режиме Source. Таким образом, можно создавать приложения Flex на любом языке. Если есть опасение, что у ваших потенциальных пользователей могут отсутствовать необходимые шрифты, вы можете воспользоваться возможностью вставить их прямо во Flex-приложение (подробнее об этом в главе 14). При этом увеличивается размер файла, однако вы можете быть уверены в точности отображения всех символов именно так, как было задумано.

[1] Привет всем! (англ.) – *Прим. перев.*

также является контейнером). Таким образом, вставьте компонент Label в верхней части Panel.

4. Измените свойство `text` компонента Label на `My name is` и измените начертание шрифта на жирное, выделив компонент и нажав на иконку Bold в разделе Style панели Flex Properties.
5. Разместите элемент TextInput под Label.
6. Еще ниже будет расположите элемент CheckBox.
7. Присвойте свойству `label` элемента CheckBox значение `I'm a Flex Expert!` [1]. Его можно изменить, дважды щелкнув по нему, как и в случае с другими компонентами.

Вот и все – вы только что создали интерфейс вашего первого Flex-приложения. Запустите его в броузере, нажав кнопку Run (вы ведь помните зеленую стрелочку?).

Сохранение перед запуском

При попытке запустить приложение, предварительно не сохранив внесенные изменения, появится диалоговое окно с вопросом о том, нужно ли это сделать (рис. 3.12). В нижней части окна вы обнаружите полезную опцию Always save resources before launching, позволяющую сохранять изменения перед запуском программы. Ею весьма удобно воспользоваться, ведь в таком случае перед запуском все внесенные изменения будут сохранены и будет произведена сборка программы. Я обычно оставляю эту опцию включенной – это позволяет сэкономить время и силы.

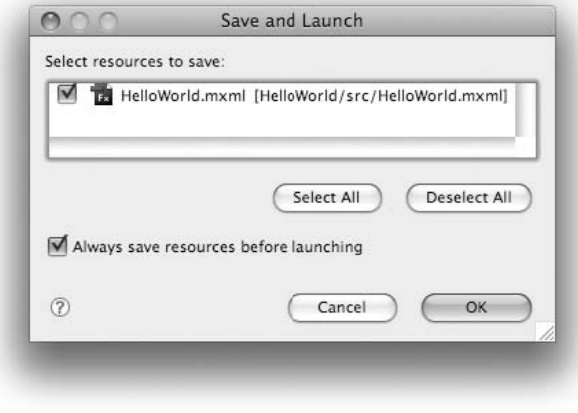

Рис. 3.12. *Диалоговое окно Save and Launch*

[1] Я гуру Flex! (англ.) – *Прим. перев.*

Вы увидите появившееся в броузере приложение (рис. 3.13). Не стесняйтесь, впишите свое имя в пустое поле для ввода текста и поставьте флажок напротив фразы I'm a Flex Expert! Вы увидите, как работают эти компоненты, и в любом случае, вы этого заслужили!

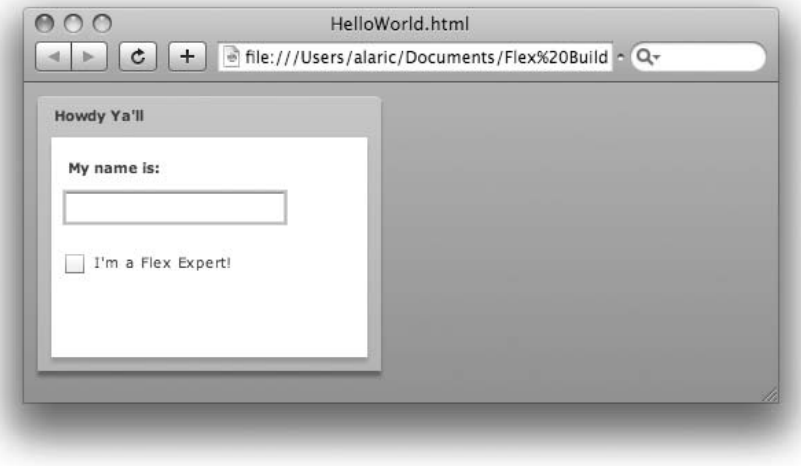

Рис. 3.13. *Ваше первое приложение*

Заключение

Вы освоили принципы работы в режиме Design и умеете создавать приложения, используя стандартные компоненты Flex. Вам еще предстоит научиться некоторым приемам, чтобы сделать свои приложения более разнообразными и интерактивными, но вы уже поднялись на несколько ступенек к вершинам Flex-мастерства. В следующей главе мы рассмотрим код, создаваемый при работе в режиме Design, и начнем писать его вручную.

В этой главе:

- Что происходит при работе в режиме Design
- Анатомия Flex-приложения
- Добавление компонентов в режиме Source
- Автозаполнение
- Постигая глубины MXML

4

Работа в режиме Source

В этой главе вы познакомитесь с языком MXML, лежащим в основе технологии Adobe Flex 3. Чтобы понять его главные особенности, мы будем использовать код, автоматически сгенерированный в примере предыдущей главы. Стоит вам освоить главные принципы языка, и вы сможете самостоятельно писать код во встроенном редакторе.

Что происходит при работе в режиме Design

Режим Design создает код вашего приложения автоматически. Когда вы перетаскиваете мышью компонент на сцену, в основном файле приложения с расширением *.mxml* появляется соответствующий ему тег. Если вы редактируете код приложения в режиме Source, при переключении в режим Design вы увидите, как отображаются внесенные изменения визуально.

Откройте приложение, над которым мы начали работать в предыдущей главе, и переключитесь в режим Source. Вы увидите созданный визуальным режимом код.

Анатомия Flex-приложения

Открыв исходный код нашего простого приложения, названного HelloWorld, вы увидите примерно следующее:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
layout="absolute">
  <mx:Panel x="10" y="10" layout="absolute" title="Howdy Ya'll">
```

Избавьтесь от предрассудков

Режим Design настолько функционален, что вы вряд ли когда-нибудь полностью откажетесь от его использования, посчитав, что это удел новичков. Я занимаюсь разработкой Flex-приложений уже многие годы, наизусть знаю MXML и большинство задач выполняю весьма быстро, но это не мешает мне работать в режиме Design время от времени. По простоте и логичности его использование сравнимо с MXML, и тем не менее, иногда бывает удобнее проектировать интерфейс приложения визуально, к тому же этот режим делает достойным вашего внимания огромное количество реализованных в нем полезных функций. Поэтому не обращайте внимания на тех, кто говорит, что вы не профессионал, если не пишете приложение от начала и до конца в текстовом редакторе. Признак профессионализма – умение правильно пользоваться инструментами для работы, а режим Design – отличный инструмент, буквально незаменимый во многих случаях.

```
    <mx:Label text="My name is:" fontWeight="bold" x="10" y="14"/>
    <mx:TextInput x="5" y="41"/>
    <mx:CheckBox label="I'm a Flex Expert!" x="10" y="71"/>
  </mx:Panel>
</mx:Application>
```

Теперь будем рассматривать данный код по частям. В первой строке вы видите объявление XML; это необязательный элемент, характерный для языка XML. Собственно «Flex» начинается со следующей строчки, а именно с расположенного на ней тега `Application`.

Любое Flex-приложение начинается с корневого тега. Если это веб-приложение, он будет называться `Application`, если это AIR-приложение, это будет `WindowedApplication`, отличающийся от предыдущего лишь несколькими дополнительными параметрами, характерными для настольных приложений. Так как наше приложение будет запускаться через броузер, корневым будет тег `<mx:Application>`. Обратите внимание на его атрибут `layout="absolute"`. Это значение определяется настройками Flex по умолчанию и означает, что любой размещаемый внутри элемент будет расположен в зависимости от значений координат `x` и `y`. Так как `Application` является контейнером, для внутренних элементов необходимо указать значения координат или установить расположение по горизонтали или вертикали (более подробно об этом во врезке «X и Y: Несколько слов о системе координат»).

Внутри `Application` расположен тег `Panel`, который также является контейнером, и по умолчанию расположение в нем внутренних элементов также является абсолютным (`layout="absolute"`). Заметьте, что благодаря наличию атрибутов `x="10"` и `y="10"` Flex располагает панель с отсту-

пом в 10 пикселов от левого угла Application. Для элементов Label, TextInput и CheckBox также заданы значения координат, но при этом отсчет ведется относительно края родительского элемента – контейнера Panel.

Естественно, у этих тегов могут быть и другие атрибуты. Посмотрите на Panel: ее заголовок определен значением атрибута title. Значение элемента Label определяет отображаемый элементом текст, а свойство fontWeight обеспечивает жирное начертание шрифта. Что касается элемента CheckBox, свойство label определяет его метку.

Попробуйте самостоятельно внести пару изменений в код. Например, можно изменить отображаемый элементом Label текст на «I'm a Flex Genius!»[1]. Затем переключитесь в режим Design, чтобы увидеть внесенные изменения.

Добавление компонентов в режиме Source

Теперь попробуем изменить интерфейс приложения, добавив в него новый компонент путем написания MXML-кода. Пусть это будет кнопка – введите <mx:Button label="Click me"/> под элементом CheckBox, но при этом не выходя за рамки тега Panel.

Перейдя в режим Design, вы увидите, что новая кнопка расположена на самом верху, а вовсе не под элементом CheckBox, как можно было бы предположить. Дело в том, что вы не задали координаты элемента, а по умолчанию Flex присваивает значения x="0" и y="0". В такой си-

Присваивание метки – свойства title, label и text

Вы вряд ли упустили из виду, что у каждого из элементов Panel, TextInput и Checkbox есть разные свойства, по сути выполняющие одну и ту же функцию – определение их метки. Почему бы не воспользоваться свойством label во всех случаях? Собственно говоря, элементу Panel присуще это свойство, но оно применяется не для создания заголовка, а для идентификации элемента при его использовании в элементах навигации (о них речь пойдет в главе 11). У панели также есть свойство status для отображения текста в верхнем правом углу. Что касается элемента Label, то придание ему свойства label было бы даже эстетически довольно бессмысленно, но главное – он предназначен для отображения текста и ему присуще свойство text, как и аналогичным элементам вроде Text, TextArea и RichTextEditor. Поначалу все это может ввести вас в смятение, но очень скоро вы разберетесь, какие свойства существуют и для чего используются.

[1] Я гений Flex! (англ.) – *Прим. перев.*

туации у вас есть два выхода – определить координаты и таким образом изменить значение по умолчанию или переместить кнопку в нужное место в режиме Design. На самом деле существует еще один способ. Если вы хотите расположить все вложенные элементы последовательно по вертикали, не задавая значения координат, этого можно достичь

X и Y: Несколько слов о системе координат

Если вы когда-нибудь изучали геометрию, то знакомы со значениями координат x и y. Напомню, что координата x определяет положение точки по горизонтальной оси (оси абсцисс), а y – по вертикальной оси (оси ординат). Вы наверняка заметили, что x и y во Flash работают немного по-другому. В Декартовой системе координат отсчет начинается от точки 0,0, и значения координат тем больше, чем правее и выше располагается точка (рис. 4.1). Например, точка с координатами x="3" и y="5" находится выше и правее точки отсчета.

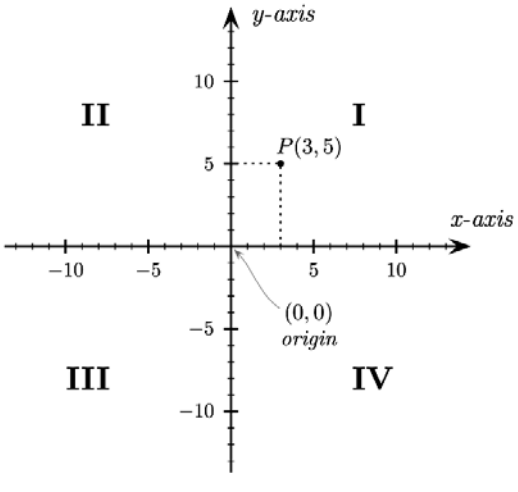

Рис. 4.1. *Прямоугольная система координат*

Во Flash точкой отсчета координат является левый верхний угол сцены, и значения координат x и y возрастают по мере увеличения расстояния по направлению вправо и к *нижнему* краю холста. Таким образом, по сути значение координаты y во Flash противоположно. Поэтому кнопка, чья координата по оси y равна 20, будет расположена с отступом в 20 пикселов от верхнего края сцены, а при значении –20 просто уйдет из поля зрения.

с помощью другого свойства. Вы, наверное, уже догадались, что это `layout`. Измените его значение на `vertical` прямо в режиме Source или с помощью панели Properties.

Теперь все элементы расположены друг за другом по вертикали, просто замечательно. Единственный недостаток – все они находятся у самого края панели, что выглядит не слишком привлекательно. И здесь на помощь приходит свойство `paddingLeft`. Установив его значение на равное 5, вы увидите, что появится соответствующий отступ от левого края контейнера до вложенных элементов – теперь все выглядит вполне гармонично.

Автозаполнение

Вы, конечно, заметили, что как только вы начинаете что-либо вводить в режиме Source, появляется список уже готовых возможных вариантов окончания. Это одна из самых полезных и необходимых функций – автозаполнение. Ваш проект могут испортить досадные опечатки, так что воспользуйтесь этой возможностью, что поможет избежать случайных ошибок и разочарований.

При вводе первых символов атрибута во Flex Builder всплывает список с подсказками, причем чаще всего одна из них (находящаяся на первом месте) уже выделена (рис. 4.2), так что вам остается лишь нажать клавишу Enter, и она встанет на свое место. Эта функция доступна при работе не только с атрибутами, но и с целыми тегами. Если вы хотите вставить кнопку, необязательно даже вводить `mx:` – достаточно открывающей угловой скобки и начальных букв названия элемента. А если вам хочется, чтобы список появлялся быстрее или необходимо открыть его снова после закрытия, просто-напросто нажмите Ctrl+пробел.

Рис. 4.2. *Меню автозаполнения*

Постигая глубины MXML

Исходный код вашего приложения основан на использовании языка разметки MXML. Он достаточно прост с точки зрения написания и чтения кода, и понимание основных принципов его использования сразу станет существенным подспорьем в вашей работе. MXML является разновидностью XML, поэтому целесообразно рассмотреть основные особенности последнего.

XML в MXML

Одной из первых прочитанных мною книг, связанных с вопросами программирования, была книга об XML. В то время я не изучал информатику, но серьезно интересовался программированием и веб-дизайном. Помню, мне было очень любопытно, что же представляет из себя этот модный язык, о котором столько говорят. Каково же было мое удивление, когда я узнал, что я все время использовал его с HTML (точнее, XHTML).

Главное – структура

Я узнал, что XML – это структурированный текст, *любой* текст с угловыми скобками (< и >). Данные представлены структурированно, с помощью тегов, образованных угловыми скобками. В XML отсутствует собственный словарь тегов, т. к. автор кода создает свои собственные теги в каждом конкретном случае. XML – это лишь синтаксис и структура.

Сейчас вы читаете книгу, которая состоит из различных частей – глав и разделов. Чтобы представить структуру книги средствами XML, можно создать свои собственные теги, такие как `<book>`, `<chapter>` и `<section>`, а затем с их помощью добавить информацию в документ.

```
<book>
  <chapter>
    <section/>
  </chapter>
</book>
```

XML расшифровывается как «расширяемый язык разметки» (англ. Extensible Markup Language). *«Расширяемый»*, т. к. позволяет создавать собственные теги, *«разметки»* – благодаря возможности представления с его помощью не только текста, но и дополнительной информации о нем, такой как форматирование текста. К примеру, пользователям первых текстовых процессоров приходилось вводить теги, подобные `<i>` и `<b>` для изменения начертания шрифтов на курсивное или жирное, соответственно. При печати такая разметка обрабатывалась соответствующим образом, а данный принцип разметки сохраняется по сей день в языке HTML.

Для Flex-разработчика еще важнее, что XML предназначен для описания иерархии и структуры любого объекта. Можно сказать, что MXML – это набор созданных специально для разработки Flex-приложений тегов XML. Так что к счастью для нас, Flex-разработчиков, для описания приложений можно использовать язык разметки.

Основополагающие принципы XML

Придерживайтесь следующих простых правил, и в процессе разработки у вас не будет возникать никаких проблем:

Все, что было открыто, необходимо закрыть. Очень важное правило использования XML состоит в необходимости закрытия всех тегов, т. е. если тег был открыт, то в каком-то месте он должен быть закрыт. Компьютеры – крайне логичные существа и не любят, когда правила нарушаются. Как вы уже видели, определение тега состоит из открывающей скобки (<), имени тега и закрывающей скобки (>), например, <book>. Такой тег считается открытым. Чтобы показать, что тег завершен и не содержит более вложенных элементов, используется левая косая черта (/).

Теги можно закрывать двумя способами. Во-первых, можно использовать закрывающие теги. В нашем примере, если есть открывающий тег <book>, ему должен соответствовать закрывающий тег </book>. Во-вторых, есть еще более быстрый способ, но его можно применять лишь при условии отсутствия вложенных тегов. Для этого добавьте косую черту прямо перед правой скобкой тега, например так: <book/>. Таким образом, выражение <book></book> эквивалентно <book/>.

Регистр имеет значение. XML чувствителен к регистру символов, т. е. прописные и строчные символы не равнозначны, и соответственно теги <book> и <Book> не тождественны. Причем <mx:Text> и <mx:text> также считаются разными тегами.

Объявления не обязательны, но желательны. Первая строка документа, написанного на XML, может (хотя и не обязательно) содержать объявление о том, на каком языке он написан и какая в нем используется кодировка. Оно выглядит примерно так:

```
<?xml version="1.0" encoding="utf-8"?>
```

Если вы пользуетесь Flex Builder, эта строка генерируется автоматически, так что об этом можно не беспокоиться.

MXML является версией XML, а значит, она наследует все эти правила.

Внимание

Перед объявлением XML в вашем документе не должно быть пробелов, пустых строк и т. п. В противном случае Flex выдаст предупреждение «Whitespace is not allowed before an XML Processing Instruction» и приложение не будет скомпилировано!

Все о теге

Тег может представлять информацию в качестве *атрибута* или *содержимого*. Содержимое – это текст, расположенный между двумя тегами, а атрибут помещается в открывающем теге, при этом представляемая информация (значение атрибута) заключается в кавычки. Посмотрите на следующий код:

```
<book title="Learning Flex" author="Alaric Cole">
   <chapter title="Getting Up to Speed"/>
   <chapter title="Setting Up Your Environment"/>
</book>
```

В данном примере `<book>` является корневым тегом, а информация о названии и авторе представлена с помощью его атрибутов `title` и `author`. Вложенные теги представляют две главы. А теперь рассмотрим код, содержащий ту же самую информацию, но представленную другим способом:

```
<book>
  <title>Learning Flex</title>
  <author>Alaric Cole</author>
  <chapter>
    <title>Getting Up to Speed</title>
  </chapter>
  <chapter>
    <title>Setting Up your Environment</title>
  </chapter>
</book>
```

По сути этот код похож на предыдущий, но нетрудно заметить, что он гораздо более громоздок. В первом случае используются атрибуты, во втором – вложенные теги. Информация, представленная с помощью атрибутов, выглядит гораздо компактнее, а это важно для восприятия. Давайте сравним предыдущие примеры со следующим кодом на MXML.

Вы, должно быть, уже привыкли к следующему виду записи:

```
<mx:Label text="Learning Flex"/>
```

Но вы, наверное, еще не знаете, что то же самое можно записать и так:

```
<mx:Label>
  <mx:text>Learning Flex</mx:text>
</mx:Label>
```

В первом случае используется атрибут для добавления свойства `text`, во втором с этой же целью используются вложенные теги. Чаще всего целесообразно использовать атрибуты по уже понятным причинам – для компактности и удобства восприятия.

Однако в определенных случаях будет предпочтительнее воспользоваться вторым способом представления информации. Вложенные теги позволяют помещать данные с более сложной организацией, нежели простые текстовые строки, поэтому их стоит использовать, если содержимое нельзя представить в качестве атрибута. Например, было бы весьма странно поместить целый абзац текста в значение атрибута `text`. Это еще важнее в случае с данными, структурированными определенным образом. Например, для элемента списка требуется предоставить данные, представляющие собой последовательность элементов, а не отдельную текстовую строку. Это справедливо и для других свойств, значением которых может быть набор данных, к примеру, для свойства `columns` элемента DataGrid, при этом возможно использовать свойства каждой отдельной колонки. Чтобы убедиться в этом на практике, перетащите данный элемент на холст в режиме `Design`. Сгенерированный код будет выглядеть примерно следующим образом:

```
<mx:DataGrid>
  <mx:columns>
    <mx:DataGridColumn headerText="Column 1" dataField="col1"/>
    <mx:DataGridColumn headerText="Column 2" dataField="col2"/>
    <mx:DataGridColumn headerText="Column 3" dataField="col3"/>
  </mx:columns>
</mx:DataGrid>
```

Свойство `columns` элемента DataGrid не может быть использовано в качестве атрибута, т. е. его использование подразумевает несколько вложенных тегов `DataGridColumn`, для которых в свою очередь определены свои свойства.

MX в MXML: несколько слов о пространстве имен

Нетрудно заметить, что любой тег в данных примерах MXML-кода по умолчанию начинается с обозначения `mx` и следующего за ним двоеточия. Это обозначает, что теги `Button` и `Panel` принадлежат к пространству имен (namespace) `mx`. Что такое пространство имен? Посмотрите на саму структуру словосочетания: пространство + имя. Пространство имен указывает на то, чем данное имя является по отношению в его месту в какой-либо структуре. Давайте возьмем следующий пример: скажем, вас зовут Джон Смит. Безусловно, на свете живет много людей с точно таким же именем, тогда как же можно вас отличить? Например, по местонахождению, в данном случае – домашнему адресу, ведь весьма маловероятно, что под одной крышей могут жить сразу несколько человек с именем Джон Смит (если, конечно, вы не назвали своего сына также Джоном, но это уже совсем другая история. И даже в таком случае, скорее всего его будут называть Джоном Смитом-младшим или Джоном Смитом II). Таким образом, в измерениях пространства имен XML к вам можно было бы обратиться *123.PineStreet:JohnSmith*, – такое имя вы вряд ли найдете где-либо еще.

Как видно из этого примера, пространство имен помогает отличать различные компоненты (которые могут иметь одинаковые имена) по их расположению. Если речь идет о MXML и ActionScript, принадлежность к какому-либо пространству имен указывает на структуру пакетов ваших компонентов (которую можно представить как организацию их расположения в папках), но об этом мы поговорим позднее. На данный момент просто примите к сведению, что Flex-компоненты принадлежат пространству имен `mx`, поскольку их источник расположен в сгруппированных определенным образом папках. Вы вполне можете создать свой собственный компонент, например кнопку, и назвать ее Button, т. к. ее исходный код будет находится в месте, отличном от того, где хранится Flex-компонент Button. В этом случае обращение к вашей кнопке может выглядеть как `<special:Button>`, чтобы отличить ее от `<mx:Button>`.

А если вы не собираетесь использовать свои собственные компоненты? Разве нельзя принять `mx` в качестве пространства имен по умолчанию и опустить все лишние обозначения `mx:` в начале тега? Почему бы не использовать теги `<Button/>` и `<Panel/>` вместо `<mx:Button/>` и `<mx:Panel/>`? На самом деле это вполне возможно, если есть такое желание. Взглянув в начало типичного документа, написанного на языке MXML, вы увидите следующее:

```
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
```

Пространства имен при использовании вложенных свойств

Обратите внимание, что вы должны указывать пространство имен (обычно `mx:`) при использовании свойства, вложенного в MXML-тег. При замене атрибута на свойство во вложенном теге вы можете машинально написать следующий (неправильный!) код для присваивания значения свойству `text` элемента Label:

```
<mx:Label>
  <text>Some Text</text>
</mx:Label>
```

Так как свойство `text` представлено в виде отдельного тега, он должен ссылаться на пространство имен `mx:`, чтобы соответствовать родительскому тегу:

```
<mx:Label>
  <mx:text>Some Text</mx:text>
</mx:Label>
```

При работе с приложением во Flex Builder функция автозаполнения обычно вводит необходимое пространство имен автоматически, поэтому в этом случае вам не придется об этом помнить.

Обратите внимание на выражение `xmlns:mx="http://www.adobe.com/2006/mxml"`. Говоря обычным языком, это значит: «Нужно создать пространство имен XML с именем mx и указать его путь „http://www.adobe.com/2006/mxml“».

На что ссылается пространство имен?

Если вы откроете адрес *http://www.adobe.com/2006/mxml* в своем броузере, вы скорее всего не увидите ничего примечательного. Эта строка напоминает ссылку на интернет-ресурс, но на самом деле является идентификатором. Использование адреса в стиле ссылки на веб-ресурс обеспечивает отсутствие конфликта имен. Таким образом, если вы создаете собственный идентификатор, у вас есть возможность указать свой собственный веб-адрес, ведь маловероятно, что такой же идентификатор может быть использован кем-то другим.

По умолчанию все Flex-компоненты принадлежат к пространству имен `mx`, но это можно изменить. К примеру, вместо `mx` можно использовать имя `flex` следующим образом:

```
<flex:Application xmlns:flex="http://www.adobe.com/2006/mxml">
```

Этот код позволяет использовать `<flex:Button>` вместо `<mx:Button>` в вашем коде. А чтобы писать просто `<Button>`, нужно изменить пространство имен на пустое:

```
<Application xmlns="http://www.adobe.com/2006/mxml">
```

Примечание

Тег `Application`, в котором задается пространство имен, также принадлежит этому пространству.

В этом случае пространство имен будет пустым, поэтому тег `<Button>` будет ссылаться на обычную кнопку Flex. Однако если вы захотите использовать собственные компоненты, придется использовать свое пространство имен, чтобы отличать их от доступных по умолчанию компонентов.

При использовании компонентов, созданных сторонними разработчиками или самостоятельно, вы скорее всего увидите имя пакета, используемое в качестве идентификатора пространства имен. Вместо текста, напоминающего ссылку на интернет-ресурс, вы увидите что-то вроде `xmlns:components="best.flex.components.*"`. Более подробно о пакетах мы поговорим в следующих главах, на данный момент просто запомните, что пространство имен предназначено для отличия различных наборов компонентов по признаку их расположения.

Что значит MX?

MX – акроним неизвестного происхождения, входящий в название языка MXML и названия предыдущих версий продуктов Macromedia (Flash MX, Dreamweaver MX). Некоторые считают, что это сокращение от Maximum eXperience[1], но вы можете придумать и свою собственную расшифровку.

[1] Высочайшая квалификация (англ.) – *Прим. перев.*

Повествовательный характер языков разметки

В использовании языков разметки есть какое-то изящество, видимо, состоящее в том, что они описательны или, по-другому, *повествовательны*. Это означает, что вы будто о чем-то повествуете. (Помните уроки родного языка в школе? Подождите, это же вопросительное предложение. А вот это было повествовательным. Найдите побудительное предложение. Вот же оно!). По сути, вы говорите: «Я хочу, чтобы здесь была кнопка», и Flex выполняет ваше пожелание. (Точнее, конечно, вам приходится печатать, если у вас нет хорошей программы распознавания речи). Вместо того чтобы строить интерфейс вашего приложения путем написания сценариев и создания процедур, можно просто вставить тег `Button` туда, где должна располагаться кнопка, а Flex позаботится об остальном.

В отличие от языка разметки, язык сценариев ActionScript, используемый во Flex, *повелевает*. Это означает, что вы создаете команды, которые компьютер должен выполнить.

Для наглядности рассмотрим пару примеров. Следующий код на ActionScript создает панель с кнопкой:

```
import mx.containers.Panel;
import mx.controls.Button;

var panel:Panel = new Panel();
var button:Button = new Button();

addChild(panel);
panel.addChild(button);
```

Данный способ инициализирует переменные Panel и Button и помещает кнопку внутрь панели. (`addChild()` – метод добавления компонентов в визуальный список, представляющий собой контейнер для всех визуальных объектов в вашем приложении). Обратите внимание, что в данном случае вы подаете команды («Компьютер, импортируй! Компьютер, добавь дочерний элемент! Я кому сказал?!»).

Сравните аналогичный случай с использованием MXML:

```
<mx:Panel>
  <mx:Button/>
</mx:Panel>
```

Гораздо лаконичнее, правда? И не нужно последовательно читать код, чтобы вникнуть в последовательность действий; достаточно одного взгляда на MXML-код, чтобы представить себе структуру приложения.

Кроме того, в такой код гораздо проще вносить изменения. Допустим, вы решили, что кнопка должна располагаться не внутри панели, а за ее пределами. Для этого достаточно поместить тег `Button` вне `Panel`, вот и все. В случае с ActionScript необходимо понимать логику построения сценария, чтобы определить, что нужно изменить метод `panel.addChild(button)` на `addChild(button)`, чтобы дочерний элемент был добавлен на сцену, а не в панель.

Заключение

Вы изучили достаточно, чтобы наконец серьезно взяться за разработку Flex-приложений в Flex Builder. Вы умеете работать в режимах `Design` и `Source` и уже приложили руку к написанию собственного кода. Вам известны основы XML и особенности его версии MXML. Теперь вы действительно готовы к новым свершениям. Следующий шаг – сделать наши приложения интерактивными. Для этого потребуется немного подробнее изучить язык сценариев ActionScript.

5

Основы написания сценариев

В этой главе:
- Общие сведения
- ActionScript, встроенный в MXML
- Точечная нотация
- Присваивание
- Функции
- Переменные
- Типы данных
- Объекты
- Классы
- Союз MXML и ActionScript
- Взаимоотношения ActionScript с MXML
- Комментарии

Образно говоря, ActionScript подобен клею, соединяющему части вашего приложения в единое целое. Если MXML используется для описания компоновки и структуры приложения, то ActionScript обычно используется, чтобы вдохнуть в приложение жизнь. Объем необходимого кода зависит от целей конкретного приложения, но, как говорится, и малые усилия могут дать большие результаты.

Понимание таких базовых вещей, как оптимальное расположение сценариев в приложении, способы создания кода, который можно использовать многократно и т. д., поможет вам создавать более мощные Flex-приложения. Кроме того, ключ к постижению сути механизма технологии Flex – в понимании принципов взаимодействия ActionScript и MXML. В этой главе вы узнаете, как работать с этим удивительно простым, но обладающим широчайшими возможностями языком программирования.

Общие сведения

В этой главе мы попробуем «оживить» приложение `HelloWorld`, созданное в главе 3. Для начала снова откройте его: на этот раз мы будем учиться прямо на практике.

Как правило, при нажатии на кнопку что-то должно произойти, так что начнем именно с кнопок. Кнопка в приложении `HelloWorld` будто просит «Нажми на меня!», поэтому начнем с этого.

Для примера я покажу, как автоматически отобразить ваше имя в текстовом поле TextInput при нажатии на кнопку. Для этого средствами ActionScript необходимо получить доступ к свойству text поля TextInput. Для обращения к элементу необходимо задать ему имя – самое время вспомнить об атрибуте id. Присвойте атрибуту id элемента TextInput значение fullNameTextInput в режиме Source или при помощи панели Properties в режиме Design.

Соглашение о наименовании

Существует весьма распространенный и популярный способ наименования компонентов, которым я также пользуюсь, – включать тип компонента в его идентификатор. К примеру, элемент TextInput в приложении HelloWorld предназначен для ввода имени – значит, его можно было бы назвать fullName. (Но не full name – в атрибуте id нельзя использовать пробелы.)

Но если в дальнейшем вам понадобится элемент Label, также отображающий ваше имя, то вы вряд ли назовете его fullName – ведь нужно же как-то различать эти компоненты. Можно выйти из положения, назвав соответствующие компоненты fullNameTextInput и fullNameLabel, т. е. добавить к названию (id) тип элемента. Если такие названия кажутся вам слишком длинными, можно использовать сокращения, например fullNameTI.

ActionScript, встроенный в MXML

Встроенный ActionScript – сценарий, расположенный прямо внутри MXML-тега. Как правило, сценарий выполняется при совершении пользователем каких-либо действий, например при щелчке мышью по объекту или при вводе текста с клавиатуры (подробнее об этом чуть ниже в этом же разделе). Можно заставить компонент реагировать определенным образом, когда происходит некоторое событие. Типичный пример – пользователь нажимает на кнопку, и это вызывает изменение свойства другого компонента приложения. Этого можно достичь с помощью следующего ActionSrcipt-кода:

```
<mx:Button id="myButton" click="someComponent.someProperty =
'something'" />
```

Все, что потребовалось сделать, – это добавить тегу Button атрибут click и в этот атрибут поместить код на ActionScript. Ниже я приведу пример с реальными значениями параметров, который можно использовать в вашем приложении:

```
<mx:Button label="Click me" click="fullNameTextInput.text =
'John Smith'"/>
```

Этот код указывает на то, что при нажатии на кнопку свойству text элемента fullNameTextInput должно быть присвоено значение John Smith. fullNameTextInput.text = 'John Smith' – код на ActionScript, помещенный внутри тега. Это самый простой способ добавления кода в приложение – размещение его прямо в теге MXML.

Точечная нотация

В рассмотренных примерах для изменения свойств компонента к соответствующему тегу добавлялись атрибуты. Например, добавление атрибута label тегу <mx:Button/> присваивает новое значение свойству label элемента Button. Это можно сделать и средствами ActionScript, используя *точечную нотацию*. С помощью атрибута text вы можете легко модифицировать свойство text созданного на MXML элемента TextInput с именем fullNameTextInput.

```
<mx:TextInput id="fullNameTextInput" text="John Smith"/>
```

То же самое можно сделать и с помощью ActionScript. Последовательно введите id компонента (fullNameTextInput) и имя изменяемого свойства, разделив их точкой:

```
fullNameTextInput.text
```

Точка указывает на осуществление доступа к чему-то, что принадлежит именно этому элементу TextInput. Если нужно совершить подобную операцию с кнопкой myButton, например изменить ее свойство label, доступ к нему можно получить следующим путем:

```
myButton.label
```

Присваивание

Для изменения такого свойства нужно осуществить операцию *присваивания*. Для присваивания нового значения свойству элемента используется знак равенства (=), за которым следует присваиваемое значение. Итак, изменить значение text элемента TextInput с помощью ActionScript можно таким образом:

```
fullNameTextInput.text = "John Smith";
```

В чем же разница между таким способом определения значения свойства text и его изменением посредством атрибутов MXML-тегов? Технически никакой разницы нет. Однако атрибуты задают значения свойств элементов в момент их создания. С помощью ActionScript присваивание можно осуществить в требуемый момент или отложить выполнение этой операции до появления определенного события, например нажатия пользователя на кнопку.

Вы, наверное, заметили, что значение атрибута click заключено в одинарные кавычки ('), а не в двойные ("). Это объясняется тем, что ис-

пользование двойных кавычек может привести к ошибке при компиляции из-за неправильной интерпретации окончания атрибута. Забавно, но на самом деле вы вполне можете поменять местами одинарные и двойные кавычки, главное при этом – соблюдать единство открывающей и соответствующей ей закрывающей кавычек.

```
<mx:Button label="Click me" click='fullNameTextInput.text =
"John Smith"'/>
```

А как быть, если при нажатии на кнопку должно происходить два или более события? Проще простого – вы можете добавить несколько выражений ActionScript в значение атрибута `click`. Допустим, при нажатии на кнопку необходимо активировать флажок `CheckBox`. Код такой операции будет чуть длиннее. Не забудьте задать `id` элементу CheckBox (`expertCheckBox`) и добавьте следующие несколько строчек в атрибут `click`:

```
<mx:Button label="Click me" click=" fullNameTextInput.text='John Smith';
expertCheckBox.selected=true "/>
```

В этом случае вы присваиваете свойству `selected` элемента CheckBox значение `true`; это означает, что элемент будет активирован. Здесь также осуществляется операция присваивания при помощи знака равенства. Обратите внимание, что присваивания разделяются точкой с запятой. Это при компиляции программы интерпретируется как две одновременно происходящих операции.

Как видите, код становится более запутанным. Если одновременно должно происходить три или четыре операции присваивания, значение атрибута `click` может растянуться на несколько строк. К счастью, код можно размещать куда более удобным образом – используя функции. Вдобавок это даст вам массу иных преимуществ.

Функции

Функция – это фрагмент кода, который может быть использован многократно. Вы помещаете некоторый код ActionScript внутрь функции, даете функции имя и, когда вам нужно выполнить этот код, ссылаетесь на эту функцию. В следующем разделе мы поместим код, осуществляющий присваивание значения свойствам элементов, внутрь функции, разместив его не в теге, а в другом месте.

Куда поместить функцию?

Код на ActionScript можно разместить в любом месте MXML-файла в качестве содержимого тега `<mx:Script/>`. Этот специальный тег может быть добавлен только в режиме Source, так как он не является визуальным компонентом – но вы ведь и не собираетесь писать код в режиме Design. Данный тег выглядит следующим образом:

```
<mx:Script>
   <![CDATA[
       //Здесь будет находиться ваш код
   ]]>
</mx:Script>
```

Вам наверняка любопытно, зачем нужен этот загадочный тег CDATA. Это специальный объект XML, сообщающий компилятору, что не нужно обрабатывать его содержимое как XML-код. Так как помещенный в него код на ActionScript может содержать такие символы, как кавычки и угловые скобки (< и >), которые могут быть неверно интерпретированы анализатором XML, он заключается в блок CDATA. Такую странную конструкцию вы вряд ли напечатаете по случайности. Она выглядит довольно сложной с точки зрения запоминания, но вам и не придется ее заучивать, т. к. благодаря функции автозаполнения при введении тега `<mx:Script/>` этот код будет добавлен автоматически. Всего лишь установите курсор после тега `<mx:Application/>`, введите первые символы тега `<mx:Script/>`, нажмите клавишу Enter, а Flex Builder введет всю эту абракадабру автоматически. Этот блок можно добавить и через меню, выбрав Source→Add Data Block.

Примечание

Во всех последующих примерах код на ActionScript будет размещен между тегами `<mx:Script>` и `</mx:Script>`.

Создание функции

Для создания функции используется ключевое слово `function`, за которым следует ее имя и пара скобок:

```
function setForm()
```

За описанием функции поместите пару фигурных скобок ({ и }). Они служат контейнером, вмещающим код функции. Поэтому далее вставьте код из атрибута `click` между ними.

```
function setForm()
{
  fullNameTextInput.text = 'John Smith';
  expertCheckBox.selected = true;
}
```

Теперь обе операции входят в состав функции. Для их выполнения остается лишь *вызвать* функцию. Как это сделать? Для этого используется имя функции с круглыми скобками:

```
<mx:Button label="Click me" click="setForm()"/>
```

Теперь при нажатии на кнопку будут определено значение элемента TextInput и активирован элемент CheckBox.

Доступность функции

Как правило, перед описанием функции можно встретить такие модификаторы доступа, как `public` или `private`. Они определяют возможность доступа к функции, т. е. возможность видеть функцию и использовать ее. По умолчанию задан вид доступа `internal`, позволяющий использовать функцию как в текущем приложении, так и в любых компонентах, входящих в состав ее пакета (о пакетах мы поговорим подробнее в главе 12). Часто вы будете сталкиваться с функциями с доступностью `public`. Это означает, что они доступны отовсюду, и именно такие функции я обычно использую в своих примерах. Значение `private` ограничивает доступность функции рамками текущего приложения.

Если вы сомневаетесь, какое значение следует использовать, оставьте настройку `public`. Такую функцию можно будет использовать в любом месте вашего кода, что поможет избежать путаницы. Вы можете изменить эту настройку позднее, если появилась необходимость ввести какие-либо ограничения.

Примечание

В действительности разные настройки доступа можно установить не только по отношению к функции, но и к любой переменной. О том, что такое переменные, говорится чуть ниже в этой главе.

Зачем может понадобиться ограничить доступ к функции? В вашем коде могут быть функции, необходимые только для вашего приложения (или компонента, если вы решите создать свой собственный). В качестве примера можно привести вспомогательную функцию, созданную для выполнения конкретной задачи в конкретном приложении, и ее доступность другим компонентам будет просто-напросто излишней.

Параметры функции

Для чего нужны круглые скобки? Их наличие сразу указывает на то, что перед вами функция. Но они используются не только из эстетических соображений, а для передачи функции информации, называемой *параметрами*. По сути, параметр передает в функцию динамическую информацию в момент ее вызова. Допустим, вы хотите передать функции текстовое значение элемента TextInput, как в уже рассмотренном примере. Это можно сделать через параметр функции.

```
public function setForm(txt)
{
  fullNameTextInput.text = txt;
  expertCheckBox.selected = true;
}
```

В данном случае определяется строковая переменная txt, доступная только внутри содержащей ее функции. Значение txt задается при вызове функции передачей значения в круглых скобках:

```
<mx:Button label="Click me" click="setForm('John Smith')"/>
```

Теперь отображаемый элементом текст не статичен, а может быть изменен с помощью данной функции.

Функция может иметь и несколько параметров:

```
public function setForm(txt, sel)
{
  fullNameTextInput.text = txt;
  expertCheckBox.selected = sel;
}
```

Теперь вызов функции можно осуществить следующим образом:

```
<mx:Button label="Click me" click="setForm('John Smith', true)"/>
```

Довольно удобно установить значения параметров по умолчанию. Это дает возможность выбора при вызове функции: если не вводится новое значение, используются предопределенные настройки. Они задаются в описании функции путем присваивания параметрам определенных значений следующим образом:

```
public function setForm(txt = "John Smith", sel = true)
{
  fullNameTextInput.text = txt;
  expertCheckBox.selected = sel;
}
```

Теперь вы можете вызывать функцию без параметров, с одним или с двумя параметрами:

```
<mx:Button label="Click me" click="setForm()"/>
```

Использование одного параметра:

```
<mx:Button label="Click me" click="setForm('John Smith')"/>
```

Вызов функции с двумя параметрами:

```
<mx:Button label="Click me" click="setForm('John Smith', true)"/>
```

Методы

Теперь вы знаете, что такое функции; самое время поговорить о методах. Все достаточно просто, поскольку по сути функции являются методами. Говоря о методе, я имею в виду функцию, являющуюся частью класса. Вы создали функцию setForm(); при этом она становится методом вашего приложения. Как правило, классам присущи как свойства, так и методы, и большинство элементов управления и другие объекты имеют определенные методы, которыми вам не раз придется воспользоваться.

Вы уже знаете метод addEventListener(), который регистрирует в системе обработчик событий. Метод addChild() помещает заданные компоненты в список для отображения. У каждого элемента управления есть метод setFocus() – он делает элемент активным. К примеру, он помещает курсор в поле элемента TextInput, так что можно сразу начинать ввод текста, и пользователю не нужно самостоятельно выбирать мышью соответствующее поле формы.

Давайте добавим эту функцию к нашему скромному приложению, чтобы сделать работу пользователя с ним еще более удобной. Вызов функции setFocus() для элемента fullNameTextInput должен быть осуществлен по окончании загрузки приложения (если это произойдет раньше, функция может работать не так, как предполагалось). Для этого нужно зарегистрировать в системе обработчик события applicationComplete. Добавьте следующий атрибут в тег Application:

```
applicationComplete="fullNameTextInput.setFocus()"
```

Примечание

Теперь, как только ваше приложение будет загружено, элемент TextInput активен и готов к вводу данных. Однако существуют тонкости, связанные с фокусировкой Flash Player, т. е. вам, возможно, придется щелкнуть мышью по приложению, чтобы переключить фокус с броузера на Flash Player.

Переменные

Переменные используются для хранения информации и последующего ее использования в приложении. К примеру, ваше приложение может содержать такие данные, как имена пользователей; их можно хранить в виде переменных. В ActionScript переменные определяются при помощи выражения var, за которым следует имя переменной.

```
var userName;
```

Затем можно присвоить переменной определенное значение.

```
username = "Tom";
```

Это можно сделать и сразу при создании переменной:

```
var username = "Tom";
```

Можно установить разный уровень доступа к переменным, как и в случае с функциями:

```
public var username = "Tom";
```

Вы уже имели дело с переменными, ведь ими являются используемые нами свойства компонентов. К примеру, в исходном коде элемента TextInput свойство text было определено как переменная.

Типы данных

В программировании *типизацией* называется способ определения вида значений, который может быть присвоен данной переменной. Это помогает разработчику сориентироваться, какого рода информация требуется. Когда заранее известен тип обрабатываемых данных, увеличивается производительность программы.

Еще одним преимуществом типизации данных является тот факт, что разные типы данных обладают различными свойствами и методами. (Вы будете регулярно использовать основные типы данных, описанные в табл. 5.1. Внимательно прочтите ее, и вы уже практически профессионал в вопросах типизации.) Безусловно, с разными типами данных можно производить совершенно разные операции, и такая предусмотрительность существенно облегчит вам жизнь.

Таблица 5.1. Основные типы данных

Имя	Описание	Пример	Значение по умолчанию
String	Простой текст, содержащий произвольное количество символов. «String» – это сокращение от «string of characters», т. е. «последовательность символов».	`var hi:String = "Hello!"`	`null`
Number	Числовое значение, допускающее наличие дробной (десятичной) части.	`var pi:Number = 3.14`	`NaN` («Not A Number», «не является числом»).
uint	Целое число «без знака», т. е. положительное. Диапазон значений – от 0 до 4.294.967.295.	`var ultimate:uint = 42`	`0`
int	Любое целое число (дроби не допускаются). Диапазон значений – от -2,147,483,648 до 2,147,483,647.	`var neg:int =-12`	`0`
Boolean	Логическая переменная, наподобие переключателя. Возможные значения – `true` или `false`.	`var isHappy:Boolean = true`	`false`
void	Означает отсутствие возвращаемых функцией значений. Может принимать единственное значение – `undefined`.	`function doNothing():void { }`	`undefined`

Допустим, ваше приложение содержит форму, в которой пользователь должен указать свое имя и возраст. Очевидно, что имя будет представлять собой информацию в виде строки, а возраст будет иметь числовое значение. Но какое? Так как это положительное целое число, то, обратившись к табл. 5.1, вы увидите, что лучше всего использовать тип uint, однако подойдет и Number.

При объявлении переменной можно с легкостью задать тип ее значений. Для этого используется двоеточие, следующее сразу за именем переменной (или за круглыми скобками, если речь идет о функции). Для определения типа значений переменной `userName` как String нужно сделать следующее:

```
var userName:String = "Hello";
```

То есть необходимо всего лишь добавить двоеточие и тип данных. То же самое и с числовым значением. В следующем коде создается переменная `pi` и типизируется как Number:

```
var pi:Number = 3.14;
```

Все это справедливо и для функций. Они способны возвращать значение, которое можно типизировать. Для определения типа значения введите двоеточие после круглых скобок, а затем тип возвращаемого значения, а для возврата собственно значения используйте ключевое слово `return`. Например, следующая функция возвращает значение суммы 2+2.

```
public function doSomeMath():Number
{
    return 2 + 2;
}
```

Для дальнейшего обращения к полученному значению можно присвоить возвращаемое функцией значение переменной:

```
var myMath:Number = doSomeMath();
// значение переменной myMath будет равно 4
```

Теперь вы можете усовершенствовать созданную ранее функцию `setForm()`, задав тип входящих в нее переменных. Можно типизировать параметры функции, а также саму функцию:

```
public function setForm(txt:String = "John Smith", sel:Boolean =
true):void
{
  fullNameTextInput.text = txt;
  expertCheckBox.selected = sel;
}
```

Вам известно, что значением параметра `txt` будет строка, а значением `sel` – логическое значение, т. к. элемент Checkbox предоставляет лишь две возможности (флажок может быть включен или выключен). Но

ведь в результате выполнения функции не возвращается никакого значения, не так ли? В таком случае используется специальный «тип данных», который, впрочем, вовсе не является типом данных. Он именуется void и означает примерно следующее: «не стоит ожидать возвращения какого-либо значения от этой функции». Я рекомендую вам упоминать это.

Примечание

Тип свойства можно определить, поместив на него курсор мыши. Через мгновение появится всплывающая подсказка, содержащая дополнительную информацию о данном свойстве, включая тип данных.

Объекты

С точки зрения объектно-ориентированного языка программирования, такого как ActionScript, вы имеете дело исключительно с разного рода объектами. Поэтому в тексте этой книги вы еще не раз встретите это понятие. «Но мне все равно непонятно, что же такое объект», скажете вы. Что ж, сейчас я все объясню.

В некотором смысле *объект* – это универсальный контейнер. Он может содержать текст или числовое значение, метод управления данными и даже входить в состав других объектов. Объект можно представить как нечто, обладающее определенным состоянием (т. е. он может включать в себя переменные) и поведением (методы работы с этими переменными). Это основа любого Flex-приложения, т. к., опять же, оно целиком и полностью состоит из объектов. Приложение является объектом. Элементы вашего приложения также являются объектами, любая создаваемая вами переменная является объектом (к примеру, свойства элементов). И так далее.

Я приведу пример создания собственного объекта. Допустим, требуется создать объект car со всеми присущими такого рода объекту свойствами, такими как тип автомобиля или его цвет. С помощью ActionScript данная задача может быть выполнена следующим образом. Сначала создается экземпляр Object, которому присваиваются определенные свойства:

```
var car:Object = new Object();
car.type = "sports car";
car.color = "red";
car.topSpeed = 170;
car.isInsured = false;
car.driver = undefined;
```

Таким образом, у нас получился мощный спортивный автомобиль, причем незастрахованный. Что ж, риск – дело благородное. Такому автомобилю необходим водитель. Решим эту задачу путем создания еще одного объекта:

```
var person:Object = new Object();
person.name = "Steve";
person.age = 19;
```

Теперь, когда у вас есть прекрасная кандидатура на роль водителя (объект), можно изменить свойство `driver` (также объект) объекта `car`:

```
car.driver = person;
```

А что делать, если в вашем приложении используется множество таких автомобилей и водителей, причем каждый объект должен обладать определенным набором свойств? В этом случае стоит прибегнуть к использованию классов.

Классы

Если мы имеем дело только с объектами, что же такое класс? *Класс* – это совокупность информации и свойств, характерных для определенной категории объектов, при этом объект является *экземпляром* класса.

Вы создали объект `car`, обладающий свойствами `color` и `type`. Чтобы создать еще один подобный объект, нужно провести такую же операцию, наделив его необходимыми свойстами. Можно пойти более логичным путем, создав класс `Car` и указав все свойства, которые вы намереваетесь использовать для такого рода объектов. Таким образом, при создании экземпляра данного класса `car` вам уже известен набор присущих ему свойств. Это даст вам массу преимуществ, ведь теперь можно не бояться опечаток и нет необходимости ломать голову, вспоминая, как называлось то или иное свойство (`hasInsurance`? `isInsured`? Или еще как-то?)

Класс `Car`, созданный на языке ActionScript, выглядит следующим образом (не думайте сейчас о компиляции этого кода, рассмотрим это просто в качестве примера):

```
public class Car
{
  var type:String;
  var color:uint;
  var topSpeed:int;
  var isInsured:Boolean;
  var driver:Person;
}
```

Обратите внимание, что свойство `driver` класса `Car` имеет тип данных `Person`. Это указывает на существование класса `Person`, который может выглядеть примерно так:

```
public class Person
{
  var name:String;
```

```
    var age:int;
}
```

Заметьте, что все свойства класса строго типизированы, т. е. age объекта Person может быть только целым числом, а свойство isInsured – принимать одно из значений true или false. Теперь вы даже знаете, что в качестве значения свойства driver будут выступать объекты класса Person. Создавая новые экземпляры данных классов, вы заранее знаете, какими свойствами (и каких типов) они будут обладать. Используя новые классы, снова создадим объекты Car и Person.

```
var car:Car = new Car();
car.type = "sports car";
car.color = 0xFF0000;
car.topSpeed = 170;
car.isInsured = false;
car.driver = undefined;

var person:Person = new Person();
person.name = "Steve";
person.age = 19;

car.driver = person;
```

Вам может поначалу показаться, что все это излишне, но на практике вы уже могли ощутить преимущества классов, т. к. используемые во Flex элементы управления основаны на классах. Например, элемент Button является экземпляром класса Button, и вне зависимости от того, на каком языке вы пишете код (ActionScript или MXML), он будет обладать одним и тем же набором свойств. Вполне вероятно, что у вас никогда не возникнет необходимости создать свой собственный класс на ActionScript, но общее понимание принципов их работы неоднократно поможет вам при разработке во Flex.

Союз MXML и ActionScript

MXML не может служить заменой ActionScript. Скорее, ActionScript и MXML дополняют друг друга. Очень скоро вы увидите, что сценарии открывают перед разработчиком новые возможности и обычно необходимы для создания интерактивного приложения. По правде говоря, приложение можно написать полностью на ActionScript без единого MXML-тега. Однако я уверен в том, что вы оцените удобство и простоту MXML при его использовании в качестве инструмента создания структуры приложения.

Взаимоотношения ActionScript с MXML

В основе технологии Flex лежит использование двух языков – MXML и ActionScript. MXML прекрасно подходит для построения структуры приложения, ActionScript отвечает за взаимодействие с пользовате-

лем. В чем же разница между этими языками? Во многих отношениях они выполняют одни и те же задачи.

MXML = ActionScript

В процессе компиляции разметка на MXML превращается в код на ActionScript. Это можно представить так: ActionScript – сердце Flash, и все созданное с помощью Flex должно быть переведено на ActionScript. В этом смысле любое Flex-приложение можно написать целиком и полностью на ActionScript. Однако использование языка разметки дает массу преимуществ; это гораздо более простой и логичный способ создания приложений.

Примечание

Компиляция – это процесс перевода или конвертирования исходного кода на другой язык. То есть компилятор – инструмент для компиляции исходного кода. Сборка – это компоновка всех необходимых частей Flex-проекта и их компиляция.

Теги – это классы

MXML-тег, описывающий какой-либо элемент приложения, при компиляции перерабатывается в соответствующий код на ActionScript. Например, для создания кнопки необходимо написать следующий MXML-код:

```
<mx:Button id="myButton" />
```

В результате компиляции данного фрагмента получится код на ActionScript:

```
import mx.controls.Button;

var myButton:Button = new Button();

addChild(myButton);
```

Таким образом, посредством ActionScript можно создавать компоненты динамически, не прибегая к использованию MXML. Выбор за вами – все зависит от конкретного приложения.

Атрибуты – это свойства

При добавлении атрибутов тега вы по сути дела изменяете свойства экземпляра объекта. Например, MXML позволяет изменить свойство `label` элемента Button таким способом:

```
<mx:Button id="myButton" label = "Click Me"/>
```

Средствами ActionScript та же самая задача решается по-другому:

```
import mx.controls.Button;
```

```
var myButton:Button = new Button();
myButton.label = "Click Me";

addChild(myButton);
```

При более детальном рассмотрении вышеприведенного примера в первой строке вы увидите выражение `import`, осуществляющее импорт `mx.controls.Button`. Это «говорит» компилятору о необходимости подготовить определенный компонент или группу компонентов к использованию.

Примечание

Не беспокойтесь, что можете забыть, что именно нужно импортировать, – Flex Builder всегда придет вам на помощь. Если вы используете функцию автозаполнения, то при вводе `Button` или иного названия класса Flex автоматически добавит необходимое выражение импорта.

Вторая строка содержит описание элемента Button с именем `myButton`, что аналогично использованию атрибута `id` в MXML. Таким образом, вы сможете ссылаться на этот конкретный элемент в вашем коде в дальнейшем.

Однако во второй строке содержится гораздо больше информации. Двоеточие и следующее за ним слово `Button` определяет тип переменной `myButton`; в данном случае это Button. Так как существуют различные типы переменных, то необходимо указать, что требуется создать элемент Button, а не Panel, строку текста или что-либо еще.

Далее следует знак равенства и выражение `new Button();`. Таким образом – введением слова `new` и затем конструктора класса – создаются новые объекты в ActionScript. Конструктор – метод, названный тем же именем, что и класс; он используется для создания новых экземпляров класса (объектов). Недостаточно только ввести переменную и определить ее тип; нужно присвоить ей определенное значение при помощи знака равенства. В случае с основными типами переменных (такими, как String или Number) можно прямо определять их значения, опустив соответствующее ключевое слово. Например, `var name:String = "hello"` или `var num:Number = 23`. Но можно использовать и такую конструкцию:

```
var name:String = new String("hello");
```

Атрибуты – это стили

Если вы заметили, при отображении свойств компонента в Category View панели Properties располагается секция Styles. Стили (styles) – особые свойства Flex-компонентов, отвечающие за визуальное впечатление, производимое компонентом. Их можно считать разновидностью свойств, но доступ к ним нельзя получить обычным путем (точечной нотацией в ActionScript). При написании кода на MXML стили легко

присваивать с помощью атрибутов. Однако они отличаются от других свойств Flex-элементов, поэтому в ActionScript используются специальные методы доступа к ним: `getStyle()` и `setStyle()`.

К примеру, стиль `cornerRadius`, определяющий степень округлости формы кнопки, задается в MXML следующим образом:

```
<mx:Button id="myButton" cornerRadius="14" />
```

Так как это свойство является стилем, к нему нет прямого доступа, и такой код будет неверен:

```
myButton.cornerRadius = 14;
```

Вместо этого следует использовать метод `setStyle`, имеющий два параметра – имя стиля и присваиваемое ему значение.

```
myButton.setStyle("cornerRadius", 14);
```

Значение стиля элемента можно получить с помощью метода `getStyle()` с единственным параметром – именем стиля, к которому осуществляется доступ. К примеру, чтобы получить значение свойства `cornerRadius` элемента myButton, можно использовать следующий код на ActionScript, в котором возвращаемый результат сохраняется в переменной `roundness`:

```
var roundness:Number = myButton.getStyle("cornerRadius");
```

Примечание

О стилях мы поговорим в главе 14, а в следующей главе во врезке «Как работать с документацией Flex» содержится полезная информация о способах изучения свойств, стилей и событий для каждого конкретного компонента.

Атрибуты – это обработчики событий

Компоненты реагируют определенным образом на такие события, как щелчок мышью, благодаря обработчикам событий. Вы уже убедились в этом, когда присваивали тегу Button атрибут `click`. Однако, хотя обработчики событий и выглядят как атрибуты в MXML, они не являются свойствами компонента. Поэтому они задаются специальным методом.

Добавление обработчика события `click` к элементу Button в MXML-коде происходит следующим образом:

```
<mx:Button id="myButton" click="doSomething()" />
```

Соответствующий код на ActionScript таков:

```
import mx.controls.Button;

var myButton:Button = new Button();
myButton.addEventListener("click", doSomething);

addChild(myButton);
```

Закулисные тайны

Вам интересно увидеть код на ActionScript, в который Flex превращает разметку на MXML? В диалоговом окне Properties вашего проекта (Project→Properties) перейдите на вкладку свойств компилятора, выбрав Flex Compiler в списке слева. Вы увидите поле Additional compiler options. Введите в него аргумент -keep, отделив его от других аргументов пробелом.

При этом создаваемый компилятором код на ActionScript будет сохранен в папке generated, и вы сможете увидеть всю работу, проделанную для вас Flex. (Обратите внимание, что при каждой компиляции содержимое папки generated обновляется, поэтому не стоит изменять что-либо в ней.)

Обратите внимание, что для регистрации событий в ActionScript используется метод addEventListener() с двумя параметрами: именем события (в данном случае click) и именем функции для вызова при его наступлении (doSomething). В отличие от кода на ActionScript, размещаемого внутри тега, метод addEventListener() не позволяет осуществлять присваивание в форме fullNameTextInput.text = txt – оно может происходить лишь в теле специально создаваемой для этого функции. Это еще одна причина, по которой необходимо иметь представление о функциях.

При использовании метода addEventListener() в качестве второго параметра следует указать только имя функции без следующих за ним круглых скобок. Например, такая запись неверна:

```
myButton.addEventListener("click", doSomething() );
```

Комментарии

Очень часто удобно оставлять комментарии прямо в коде – как пояснения для себя или других или для того, чтобы сделать часть кода временно недоступной. Оформление комментариев в MXML и ActionScript различно.

При написании кода на ActionScript комментарий может располагаться как на одной, так и на нескольких строках. Однострочный комментарий отделяется от основного кода двойной косой чертой:

```
//This code won't be run
//public var foo:String = "No Comment";
```

Если текст комментария не умещается на одной строке, удобнее воспользоваться косой чертой со звездочкой:

```
/*
public var foo:String = "No Comment";
public var bar:String = "Don't want to see it";
*/
```

MXML основан на XML, и здесь способ вставки комментариев совсем иной – комментарий заключается между последовательностями символов <!-- и -->, как и в HTML.

```
<!-- This is a comment in MXML -->
<mx:Button label="Button to Keep"/>
<!-- <mx:Button label="Button to Remove"/> -->
```

Примечание

Работая во Flex Builder, вы можете добавлять комментарии, даже не задумываясь об их правильном синтаксическом оформлении. Просто выделите соответствующий фрагмент кода и выберите Source→Add Block. В зависимости от того, на каком языке (MXML или ActionScript) написан код приложения, Flex Builder автоматически вставит необходимую синтаксическую конструкцию. Убедитесь, что фрагмент выбран целиком во избежание ошибок при компиляции.

Возможность добавления комментариев – еще один довод в пользу создания приложений с помощью языков разметки. Они позволяют временно убрать фрагмент приложения (который можно снова добавить в любое время), сохранив при этом его исходный код. Например, в вашем приложении может быть панель для настройки предпочтений, которую вы не успели закончить в отведенный срок. Код незавершенной панели можно оставить в качестве комментария и доработать его позднее, при этом компилятор не станет его обрабатывать.

Заключение

Эта книга не является учебником по ActionScript – я поставил перед собой цель представить общий обзор возможностей этого языка. Этих сведений будет вполне достаточно для создания простых способов взаимодействия с пользователем, однако для более глубокого изучения предмета рекомендую вам обратиться к дополнительной литературе, например к книге «Learning ActionScript 3.0: A Beginner's Guide» из серии «Learning», выпущенной в свет издательством O'Reilly, – это отличный вводный курс для начинающего программиста на ActionScript.

6

В этой главе:
- Что такое событие
- Обработка событий внутри MXML-кода
- Константы событий
- Приложение в действии
- Отладка в свое удовольствие

Создание интерактивных приложений с помощью ActionScript

Возможность взаимодействия с пользователем – то, что вдохнет жизнь в приложение. Сама суть приложения – в возможности перетаскивания объектов, выполнения различных действий с помощью мыши, ввода текста с клавиатуры и т. д. – иначе вы имеете дело с обычной анимацией или статичным изображением, а это, согласитесь, скучновато.

Однако, чтобы предугадать, как пользователь будет работать с вашим приложением, и создать соответствующие способы реакции на его действия, требуется тщательное планирование. Необходимо продумать все возможные методы взаимодействия с пользователем – как правило, нужно четко представлять, что будет делать пользователь и каким образом будет вести себя приложение при этом. Конечно, это непростая задача, но игра стоит свеч.

Что такое событие

Реакция Flex-приложения на производимые пользователем действия называется *событием*. Событие происходит при взаимодействии пользователя с приложением или при совершении каких-либо иных действий, таких как загрузка изображения или поступление данных с сервера (Основные сведения о получении данных с сервера можно найти в главе 10). Например, событие происходит, когда пользователь нажимает на кнопку (как вы наверняка уже догадались, такое событие называется `click`). Создание кнопки – тоже событие (оно названо `creationComplete`). Когда событие происходит, то говорят, что оно *было инициировано или обработано*. Чтобы определить реакцию приложения

на происходящее событие, необходимо установить *обработчик событий* (*event handler*) или *создать слушателя событий* (*event listener*).

Обработка событий внутри MXML–кода

Возможности MXML включают эффективные способы обработки событий. Достаточно добавить соответствующий атрибут события в тег и задать его значение: это может быть вызов функции или просто изменение какого-либо свойства. В приведенном коде при нажатии на кнопку `myButton` изменяется текст элемента Label (с именем `myLabel`), что происходит благодаря соответствующему изменению его свойства `text` (рис. 6.1). Для достижения такого эффекта нужно всего лишь добавить в тег Button атрибут `click`:

```
<mx:Label id="myLabel" text="The Button will change my text" />
<mx:Button id="myButton" label="Change it!" click="myLabel.text = 'Some
new text'"/>
```

Такой способ подходит и для обработки других событий, таких как изменение свойства `text` поля TextInput при вводе в него текста пользователем. Такое событие называется *change*; в приведенном ниже коде при изменении элемента TextInput изменится текст элемента Label:

```
<mx:Label id="myLabel" text="The TextInput will change my text" />
<mx:TextInput id="myTextInput" change="myLabel.text =
myTextInput.text"/>
```

***Рис. 6.1.** Событие click шаг за шагом*

В данном примере при каждом изменении свойства text элемента TextInput соответствующим образом изменяется текст, отображаемый элементом Label. На самом деле существует более подходящий способ обновления значения свойства в ответ на изменение другого свойства, и называется он *связыванием данных*. Об этом методе мы поговорим подробнее в главе 7.

Количество возможных событий и их имен не ограничено, однако существует несколько испытанных временем и часто используемых стандартных событий. Их список представлен в табл. 6.1.

Таблица 6.1. Основные события

Имя события	Константа	Описание
Change	Event.CHANGE	Выбран другой пункт списка или элемента навигации (такого как TabBar) или изменено свойство text компонента.
Click	MouseEvent.CLICK	Щелчок по элементу. То есть пользователь нажимает кнопку мыши над элементом, а затем отпускает ее.
creationComplete	FlexEvent.CREATION_COMPLETE	Создание Flex-компонента.
mouseDown	MouseEvent.MOUSE_DOWN	Нажатие кнопкой мыши на компонент.
mouseUp	MouseEvent.MOUSE_UP	Отпускается кнопка мыши.
Resize	Event.RESIZE	Изменение размеров приложения при изменении размера окна броузера.
rollOut	MouseEvent.ROLL_OUT	Курсор мыши покидает область компонента.
rollOver	MouseEvent.ROLL_OVER	Перемещение курсора мыши в область компонента.

Конечно, существует масса других событий, и многие из них характерны для определенных компонентов, но события, перечисленные в табл. 6.1, уже открывают перед вами широкие возможности. Чтобы узнать, какие события можно использовать с конкретным элементом, воспользуйтесь одним из нижеперечисленных способов:

- В режиме Design выберите расположенный на сцене компонент и обратитесь к панели Properties (переключите ее в Category View). В разделе Events будут перечислены все доступные для данного элемента события.
- В режиме Source при работе с включенной функцией автозаполнения при вводе названия компонента всплывает список всех его свойств. При этом события будут отмечены значком с изображением молнии.

- Ознакомьтесь с документацией, выбрав меню Help→Help Contents и найдя нужный компонент, или выделив интересующий вас компонент в режиме Source и затем открыв Help→Find in Language Reference. Вверху страницы, посвященной конкретному компоненту, расположен перечень ссылок на все его свойства, методы и события. Более подробная информация содержится в списке событий. (О том, как работать с документацией Flex, вы можете прочитать во врезке на следующей странице).

Как работать с документацией Flex

Flex предоставляет прекрасную документацию, доступную в сети Интернет по адресу *http://livedocs.adobe.com/flex/3/* или через меню Flex (Help→Help Contents).

Документация Flex содержит большое количество статей о различных аспектах технологии, так что если вы внимательный читатель, вы сможете найти практически любую интересующую вас информацию для совершенствования ваших навыков, включая примеры кода. Благодаря хорошо организованному поиску вы сможете найти именно то, что нужно.

Flex 3 Language Reference – одна из наиболее полезных частей документации, включающая всю необходимую информацию об использовании компонентов Flex. Поначалу вы можете несколько растеряться при обращении к ней, но как только вы освоитесь, вы еще не раз к ней обратитесь.

Доступ к Language Reference осуществляется одним из следующих способов. Можно просто искать информацию о компоненте по его имени (например, Button). Если вы работаете в режиме Source, можно выделить определенный компонент и выбрать в меню Help→Find in Language Reference. Кроме того, можно просмотреть оглавление Language Reference, выбрав Adobe Flex 3 Help и далее Adobe Flex 3 Language Reference. Как правило, вам может потребоваться информация о конкретном компоненте, поэтому я рекомендую воспользоваться поиском или функцией Find in Language Reference в Flex Builder.

Страница, посвященная свойствам определенного компонента, выглядит примерно как на рис. 6.2. В верхней части страницы вы увидите перечень всех свойств, методов и событий компонента и т. д., включая даже ссылку на примеры использования.

Все доступные для элемента события можно увидеть, щелкнув по ссылке Events или прокрутив страницу в броузере до заголовка Events. Там расположен список событий (рис. 6.3). Однако, как и в случае со свойствами, у компонента могут быть унаследованные события, которые можно увидеть, щелкнув по ссылке

Show Inherited Events. Как правило, компоненты наследуют события от элементов, стоящих выше в иерархии, поэтому, если вы хотите увидеть свойства базовых компонентов, выберите этот пункт. Это довольно ценная информация, ведь многие необходимые вам свойства могут быть унаследованы элементом. К примеру, компонент Button наследует событие `click` от класса UIComponent (являющегося базовым классом, на котором основаны все визуальные компоненты Flex).

Flex 3 Language Reference – всеобъемлющий справочник, содержащий всю необходимую разработчику Flex-приложений информацию об использовании компонентов в ActionScript и MXML.

Рис. 6.2. *Информация Flex Language Reference об элементе Button*

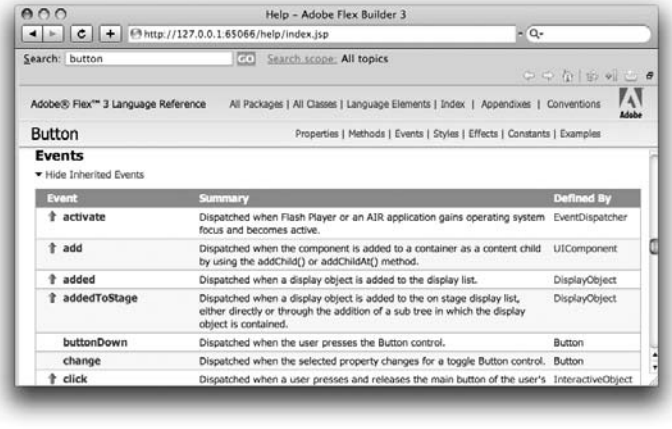

Рис. 6.3. *Информация Flex Language Reference об элементе Button – раздел Events*

Константы событий

Константой во Flex называется постоянное значение, т. е. способ хранения переменной, не предполагающей изменений. Константы часто используют для наименования числовых значений, ведь, согласитесь, слова запоминаются гораздо легче, чем числа. Вы помните число π (пи) до пятнадцатой цифры дробной части? Сомневаюсь. А код клавиши Caps Lock в ASCII? Так что не зря кто-то очень мудрый придумал константы. Теперь значение пи можно получить при помощи константы `Math.PI`, а код клавиши – при помощи `Keyboard.CAPS_LOCK`.

Но константы применяются не только для хранения цифровых значений. Во Flex часто используются строковые константы, они часто описывают типы событий. К примеру, событие выхода указателя мыши за границы Flex-приложения называется `mouseLeave`. Строки «mouse leave» или «mouse-leave» не подходят, поэтому во Flex существует специальная константа `Event.MOUSE_LEAVE`, благодаря чему нет необходимости помнить точное написание.

Вы можете подумать, что это ничуть не легче для запоминания. Что ж, не буду спорить, но учтите, что поскольку константы являются свойствами классов (`MOUSE_LEAVE` – свойство класса `Event`, `PI` – свойство класса `Math` и т. д.), использование функции автозополнения позволит избежать опечаток, т. к. вам достаточно ввести несколько первых символов, а Flex Builder позаботится об остальном.

Вы наверняка заметили, что название констант состоит полностью из заглавных букв – это их отличительный признак. Использование заглавных букв на самом деле не обязательно, но это считается хорошим тоном и просто удобно. Поскольку зрительно достаточно трудно разделить текст, записанный прописными буквами, на отдельные слова, для разделения слов используется символ подчеркивания (_).

Приложение в действии

Давайте вернемся к приложению HelloWorld и заставим его совершить кое-какие действия.

Во-первых, добавим еще пару элементов для взаимодействия с пользователем. Поле для ввода имени у нас уже есть, теперь добавим поле для возраста. Поместите новый элемент Label с текстом «My age is» под полем для ввода имени, а еще ниже разместите элемент NumericStepper и задайте значения их атрибутов `id` соответственно `ageLabel` и `ageNS`. Элемент NumericStepper похож на TextInput, но предназначен для ввода числовых значений и потому дает пользователю возможность увеличивать и уменьшать значение с помощью стрелок на клавиатуре или кнопок со стрелками (при помощи мыши). Нам следует изменить пару настроек данного компонента. Его свойство `maximum` определяет максимальное возможное значение (установите его равным 120), `minimum` –

соответственно минимальное (пусть в нашем случае оно будет равняться 18). Дополнительный код будет выглядеть следующим образом:

```
<mx:Label id="ageLabel" text="My age is:"/>
<mx:NumericStepper id="ageNS" maximum="120" minimum="18" />
```

Отображение информации во всплывающем окне

Теперь у нас есть простая форма; попробуем вывести полученные с ее помощью данные во всплывающем окне. Для этого создадим функцию, использующую класс Alert, который предназначен для вывода окна поверх приложения:

```
import mx.controls.Alert;

public function showInfo():void
{
  Alert.show("Your name is " + fullNameTextInput.text +
    " and your age is " + ageNS.value);
}
```

Обратите внимание на выражение `import`, осуществляющее импорт необходимого для выполнения данной функции класса Alert. При использовании в работе возможности автозаполнения большинство требующихся выражений импорта будут введены автоматически.

Эта функция обладает некоторыми до сей поры незнакомыми для вас особенностями. Во-первых, это метод `show()` класса Alert – статический

Статические свойства и методы

Еще раз внимательно рассмотрите код, осуществляющий отображение всплывающего окна. Синтаксис метода `show()` может показаться вам довольно непривычным, поскольку данный метод существует в классе, а не в экземпляре этого класса. Таким образом, вы используете имя класса (Alert) и вызываете его метод с помощью точечной нотации.

Как и константы, являющиеся свойствами класса, статические методы также присущи классу. Вместо того чтобы создавать переменную типа Alert (выражением `new Alert()`), а затем вызвать метод `show ()` данного экземпляра, вы вызываете метод самого класса Alert.

Константы – это, в общем, статические свойства класса, т. е. поля, присущие самому классу, поэтому при работе с ними и статическими методами используются схожие синтаксические конструкции. (Это достигается использованием модификатора доступа `static` при создании таких методов или переменных в ActionScript.)

метод, присущий самому классу, который отображает окно поверх всех остальных элементов приложения.

Обратите внимание, что элемент NumericStepper обладает свойством value, а не text, как можно было бы предположить, исходя из схожести их функций с соответствующим свойством TextInput. Такое название обусловлено тем, что элемент NumericStepper может содержать лишь числовое значение, т. е. данные типа Number. Поэтому со значением свойства value можно проводить математические операции, например, ageNS.value + 10 увеличит ваш возраст на 10 лет. (Хотя, пожалуй, убавить лет 10 было бы гораздо лучше).

В связи с этим возникает еще один вопрос. Знак плюс (+) используется в ActionScript как для суммирования числовых значений, так и для объединения текстовых строк. Приведенный пример с возрастом – это сумма чисел, но в выражении "and your age is " + ageNS.value знак плюс используется для конкатенации, т. е. объединения, данных. В вашем приложении формируется строка текста с использованием динамической информации, полученной из полей формы. Как вы уже знаете, значение свойства value может быть только числовым, но Flex самостоятельно определяет, что нельзя провести операцию суммирования числа и текстовой строки (такой как " and your age is "), поэтому числовое значение преобразуется в текстовое. Возможно, вы не обращали на это никакого внимания, однако об этом важно помнить, т. к. в определенных случаях такая возможность Flex может работать не так, как вы ожидаете. Для получения дополнительной информации по этому вопросу обратитесь к врезке «Конвертация, явное и неявное преобразование типов».

Последнее, на что необходимо обратить внимание в данном примере – на разрыв цельной строки кода. Как правило, отдельное выражение должно целиком располагаться на одной строке, – даже довольно длинное значение параметра функции show(). Но у этого правила есть исключение: если в выражении используются знак плюс или запятая, оно может переходить на следующую строку. Для удобства восприятия перед началом второй строки делается небольшой отступ, чтобы показать, что это продолжение предыдущей строки. Конечно, такой длинный код можно расположить в одной строке, но это создает неудобства для пользователей с небольшими экранами: чтобы прочитать код целиком, им придется использовать горизонтальную полосу прокрутки.

Примечание

Во избежание появления горизонтальных ползунков я рекомендую вам установить максимальную длину строки кода в 80 символов. Это оптимальный вариант для большинства пользователей.

Конвертация, явное и неявное преобразование типов

Итак, вы имеете четкое представление о типизации переменных. Вы продумали все до мелочей, определили тип каждой переменной и теперь не остается ни тени сомнений в правильности вашего кода. Но однажды утром вы просыпаетесь с мыслью о том, что нужно изменить тип определенной переменной. Как же это сделать?

Вы, профессионал Flex, можете воспользоваться одним из способов преобразования типов данных. Неявное преобразование подойдет в том случае, если в вашем коде содержится переменная с числовым типом значения, которую вы используете как строку. В качестве примера можно привести случай, когда значением свойства `text` элемента TextInput становится число. Как правило, необходимые операции преобразования Flex произведет автоматически, но иногда при компиляции может произойти ошибка. Тогда нужно произвести явное преобразование, при котором в процессе компиляции производится (по крайней мере временное) изменение типа данных. Например, если в поле, предназначенное для текста, вводится числовое значение, его нужно преобразовать в текстовое.

В вашем распоряжении несколько способов явного преобразования типа значения. Во-первых, с этой целью можно использовать конструктор соответствующего типа. Конструктор – это специальный метод создания новых объектов класса, например, `new String()`. Название метода совпадает с названием класса. Например, в следующем коде конвертируется помещенное в круглые скобки значение: `fullNameTextInput.text = String(ageNS.value)`. Еще одно средство преобразовования – с помощью ключевого слова `as`, которое указывает компилятору, как интерпретировать переменную. Пример применения такого способа – `fullNameTextInput.text = ageNS.value as String`. И, наконец, последняя возможность преобразования состоит в использовании метода `toString()`, которым обладает большинство классов Flex. Это очень удобный способ, в результате которого возвращается значение типа `String`. Пример его использования – `fullNameTextInput.text = ageNS.value.toString()`.

Следующим шагом будет замена функции `setForm()` на `showInfo()` в обработчике событий `click`, так что при нажатии пользователем на кнопку появится всплывающее окно. Запустите приложение, чтобы увидеть результат проделанной работы (все должно выглядеть примерно как на рис. 6.4). Код вашего приложения будет примерно следующим:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
layout="absolute"
applicationComplete="fullNameTextInput.setFocus()">

  <mx:Script>
      <![CDATA[
          import mx.controls.Alert;

          public function showInfo():void
          {
              Alert.show("Your name is " + fullNameTextInput.text +
                   " and your age is " + ageNS.value);
          }
      ]]>
  </mx:Script>

  <mx:Panel id="panel" x="10" y="10" width="250" height="200"
      layout="vertical" title="Howdy Ya'll" paddingLeft="5">
      <mx:Label text="My name is:" fontWeight="bold"/>
      <mx:TextInput id="fullNameTextInput"/>
      <mx:Label id="ageLabel" text="My age is:" fontWeight="bold"/>
      <mx:NumericStepper id="ageNS" maximum="120" minimum="18" />
      <mx:CheckBox id="expertCheckBox" label="I'm a Flex Expert!"/>
      <mx:Button label="Click me" click="showInfo()"/>
  </mx:Panel>

</mx:Application>
```

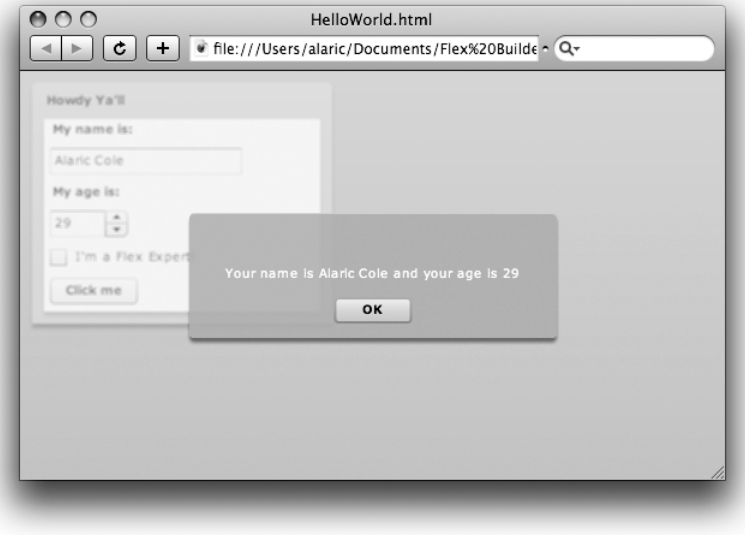

Рис. 6.4. *Всплывающее окно с информацией*

Событие change

До сих пор мы имели дело только с событиями click, происходящими при нажатии на кнопку или иной элемент управления. Теперь рассмотрим другой тип событий, именуемый change. Такое событие происходит при выборе пользователем другого объекта или изменении текста элемента управления, например NumericStepper или TextInput. Добавим обработчик события change к элементу NumericStepper, который вызывает функцию showInfo():

```
<mx:NumericStepper id="ageNS" maximum="120" minimum="18"
change="showInfo()" />
```

Теперь при изменении значения NumericStepper (при помощи мыши, стрелками с клавиатуры или просто путем введения числа и подтверждения выбора, т. е. переключения на другой элемент клавишей Tab

Что делает Flex втайне от вас

Вы наверняка уже не раз задумывались над механизмом работы Flex. Вы пишете код своего шедевра на MXML или работаете над ним в режиме Design, но что же происходит при нажатии на ту маленькую зеленую кнопочку?

Flex – компилятор как MXML, так и ActionScript-кода. При компиляции приложения с помощью Flex Builder в фоновом режиме запускается сценарий, выполняющий следующие операции:

1. Конвертирование MXML-кода в код на ActionScript.
2. Конвертирование метаданных в аргументы компиляции.
3. Компиляция сгенерированного кода на ActionScript в байт-код (файл с расширением .swf).

ActionSctript остается основным языком Flex-приложений. Богатые возможности MXML, включающие возможность использования стилей, создания различных сервисов и эффектов, существенно упрощают процесс создания приложений. Но компилятор Flex превращает ваш четкий MXML-код в паутину классов ActionScript, по сути создавая большое количество низкоуровневого кода вместо вашего высокоуровневого. Затем в результате повторной компиляции кода получается байт-код SWF, содержащий набор указаний, интерпретируемых проигрывателем Flash Player. В файл SWF также включаются специальные метаданные (данные, заключенные в квадратные скобки), например встроенный источник внешних ресурсов, настройки частоты смены кадров и т. д.

или указателем мыши), происходит событие и появляется всплывающее окно. Обратите внимание, что оба события – `click` элемента Button и `change` элемента NumericStepper вызывают одну и ту же функцию `showInfo()`. Это одна из самых сильных сторон программирования – возможность вызова одной и той же функции из разных событий, как и вызов различных функций одним событием.

Отладка в свое удовольствие

Пример с отображением всплывающего окна прекрасно подходит для изучения событий, но появление такого окна каждый раз при малейшем изменении в программе создает одни неудобства. К тому же вы вряд ли будете часто использовать эту возможность в реальных приложениях; это, скорее, способ изучения механизмов работы и тестирования вашей программы. Для отображения информации о событиях существует гораздо более эффективный способ, называемый *отладкой*. Несмотря на такое название, это вовсе не означает, что в ваше приложение закралась ошибка, которую необходимо найти и исправить. Отладка Flex-приложения помогает лучше понять работу его внутреннего механизма, а также используется для отображения некоторой информации в процессе разработки.

Вывод информации на консоль

Для отладки приложения используется метод `trace()`, который выводит текст на консоль (панель `Console`). Замените выражение `Alert.show` на `trace` следующим образом:

```
public function showInfo():void
{
  trace("Your name is " + fullNameTextInput.text +
    " and your age is " + ageNS.value);
}
```

При запуске приложения эта информация отображаться не будет – данная функция будет работать при его отладке. Как запустить отладку? Проще простого. Вы запускали приложения, нажимая на кнопку с зеленой стрелкой; теперь нажмите на расположенную справа от нее кнопку с изображением зеленого жучка. Приложение будет запущено в режиме отладки. В большинстве случаев отображаемый на консоли результат будет примерно одинаковым, но при изменении значения элемента NumericStepper на ней появится заданная строка текста.

Итак, запустите отладку и несколько раз измените значение NumericStepper, а затем вновь переключитесь во Flex Builder и посмотрите на панель `Console` – на ней выводится информация, отображаемая ранее во всплывающем окне (причем при каждом изменении значения элемента выводится новая строка).

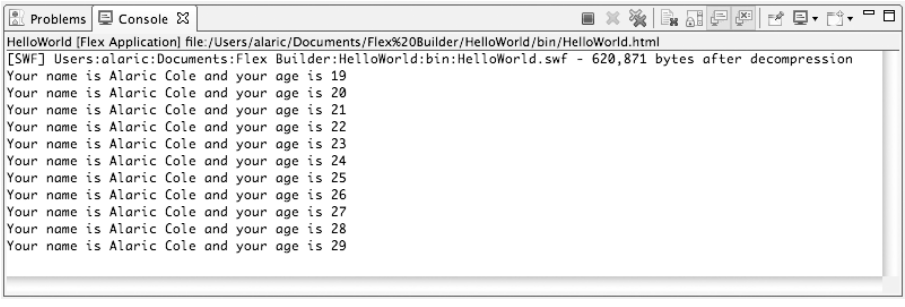

Рис. 6.5. *Результат использования метода trace, отображаемый на панели Console*

Примечание

По умолчанию панель Console расположена в нижней части окна Flex Builder рядом с панелью Problems. Если она не отображается, откройте ее через основное меню (Window→Console).

Простота использования выражения `trace` обманчива. Однако со временем она станет вашим незаменимым инструментом при работе с режимом отладки. А теперь мы попробуем использовать этот режим для получения сведений о происходящем событии.

Использование параметров событий при отладке

Надеюсь, вы не забыли о параметрах, которые можно передать функции? Добавьте в функцию `showInfo()` параметр `event` с типом данных `Event`. (Параметру можно дать любое другое имя, но я предпочитаю называть его *event* (строчными буквами). Теперь выражение `trace` поможет нам получить некоторые сведения о событии, а конкретнее – значения его свойств `type` и `currentTarget`. Код функции таков:

```
public function showInfo(event:Event):void
{
  trace("The type is " + event.type +
    " and the current target is " + event.currentTarget);
}
```

Таким образом, в функцию будет передан объект `event`, который относится к типу `Event`, а значит, обладает соответствующими свойствами, дающими представление об этом конкретном событии. Свойство `type` содержит тип события, в нашем случае это событие `change` («изменить»). Свойство `currentTarget` ссылается на объект, вызвавший функцию.

Вы уже догадались, что для достижения поставленной цели необходимо совершить еще один шаг: организовать передачу параметра функции при вызове. Во Flex существует специальный встроенный параметр `event`, используемый для передачи объектов событий в функцию,

его мы и будем использовать. Вы не создавали переменной с именем event, она будет создана Flex во время компиляции. Чтобы передать само событие в обработчик события change элемента NumericStepper, используйте следующий код:

```
change="showInfo(event)"
```

А чтобы передать событие в обработчик события click элемента Button, используйте:

```
click="showInfo(event)"
```

Функция ожидает передачи параметра event, поэтому при его отсутствии вызов функции приведет к ошибке компиляции.

Итак, вы запустили отладку приложения и изменили значение элемента NumericStepper; теперь на панели Console должна отображаться примерно следующая информация: «The type is change, and the current target is HelloWorld0.panel.ageNS.» Как вам? Вы заранее знали, что тип события будет change, но значение свойства currentTarget выглядит немного запутанно. Обратите внимание, что ageNS – это значение идентификатора id элемента NumericStepper. Но что означает начальная часть этой записи – *HelloWorld0.panel*? Дело в том, что при создании приложения во Flex создается и класс, имя которого основано на имени приложения. В данном случае к названию приложения был добавлен ноль. Что касается выражения *.panel*, это свойство id элемента, в котором располагается NumericStepper. Иерархия приложения такова, что элемент с идентификатором ageNS является частью панели, которая в свою очередь считается частью приложения HelloWorld.

При нажатии на кнопку значение выражения trace будет несколько иным. В этом случае происходит событие click и будет отображена соответствующая информация, а свойство currentTarget примет значение кнопки. (Если свойство id кнопки не определено, Flex присвоит ему определенное значение автоматически. Оно может выглядеть примерно так: Button0 или Button23 или что-то в этом роде.)

При создании функций бывает полезно использовать такие сведения о событиях, чтобы сделать их более универсальными. Можно создать функцию, выполняющую различные действия в зависимости от вызвавшего ее объекта (currentTarget) или события (тип Event).

Использование контрольных точек[1]

Далее мы поговорим о достаточно сложных вещах, рассчитанных на опытных пользователей, однако, на мой взгляд, умение пользоваться такими методами, как установка контрольных точек крайне важно; это поможет разобраться в работе приложения.

---

[1] Также их называют точками останова (от англ. breakpoint). – *Прим. перев.*

Контрольная точка – это место в логике вашего кода, где вам бы хотелось все остановить. Структура приложения может быть достаточно сложной, и иногда очень удобно воспользоваться возможностью остановить процесс в определенный момент. Такая опция доступна при использовании контрольных точек: они позволяют сделать паузу и увидеть текущее состояние приложения.

Чтобы вставить контрольную точку, дважды щелкните мышью в левой колонке, содержащей нумерацию строк, напротив соответствующей строки кода. В нашем случае сделать это имеет смысл только в единственном месте – внутри нашей функции. Итак, поместите контрольную точку слева от выражения `trace`, как на рис. 6.6.

```
 8
 9          public function showInfo(event:Event):void
10          {
11              trace("The type is " + event.type +
12                  " and the current target is " + event.currentTarget);
13          }
```

Рис. 6.6. *Контрольная точка в строке 11*

Теперь при изменении значения элемента NumericStepper в режиме отладки будет осуществлен вызов функции и произойдет останов. Вероятно, в первый раз перед вами откроется диалоговое окно Flex Builder, предлагающее переключиться в перспективу Flex Debugging (рис. 6.7). Рекомендую вам установить флажок напротив Remember my decision (Запомнить мое решение), чтобы каждый раз при установке контрольных точек открывалась нужная перспектива. Нажмите на кнопку Yes, и вы увидите совершенно новую перспективу.

Рис. 6.7. *Диалоговое окно Comfirm Perspective Switch*

Новая перспектива

Перспективы в Flex Builder – это готовые варианты взаимного расположения панелей в окне программы. Когда вы начинаете разработку во Flex Builder, вы используете раскладку Flex Development, содержащую панели Flex Properties и Flex Navigator. В процессе отладки вам потребуются другие панели, которые открыты в раскладке Flex Debugging.

Переключившись на перспективу Debugging, вы увидите несколько новых панелей. Во-первых, это панель Variables, по умолчанию расположенная в верхней части окна. Flex Builder позволяет вам изменять размер панелей и перетаскивать их мышью в любое удобное для вас место. Так как панель Variables содержит древовидный список переменных, ее удобнее расположить вертикально. Я рекомендую вам разместить ее в правой части окна Flex Builder. Просто перетащите панель мышью к его правому краю. При перемещении панели появится тонкая черная рамочка, помогающая расположить панель. Панель можно переместить и за пределы окна Flex Builder; в этом случае она будет выглядеть как отдельное окно.

Теперь при каждом нажатии на кнопку или изменении значения NumericStepper на панели Variables будет отображаться информация о текущем состоянии вашего приложения в момент останова в контрольной точке. В данном случае посмотрите в списке на пункты `this` и `event`. Первый из них ссылается на приложение в целом, второй – на параметр `event`, который был передан функции.

Щелкните по стрелке, расположенной слева от пункта `event`; откроется список дочерних узлов. Нас интересует список, озаглавленный `[inherited]`; раскройте его аналогичным образом. Вы увидите все свойства объекта `event`, включая уже известные вам `type` и `currentTarget`, причем будут отображены и их текущие значения (рис. 6.8).

Завершение процесса отладки

Не бойтесь самостоятельно осмотреться. Вполне вероятно, что вы еще будете пользоваться режимом отладки, т. к. это мощный инструмент, помогающий понять работу Flex изнутри. А сейчас закройте раскладку Flex Debugging и завершите процесс отладки. Этот режим может заблокировать работу вашего броузера, т. к. он ставит на паузу проигрыватель Flash Player. Для завершения отладки нажмите на красную квадратную кнопку на панели Console либо Debug (рис. 6.9). Также можно воспользоваться командой Run→Terminate. Чтобы снова переключиться на раскладку Flex Development, используйте переключатель, расположенный правом верхнем углу Flex Builder или команду Window→ Perspective→Flex Development.

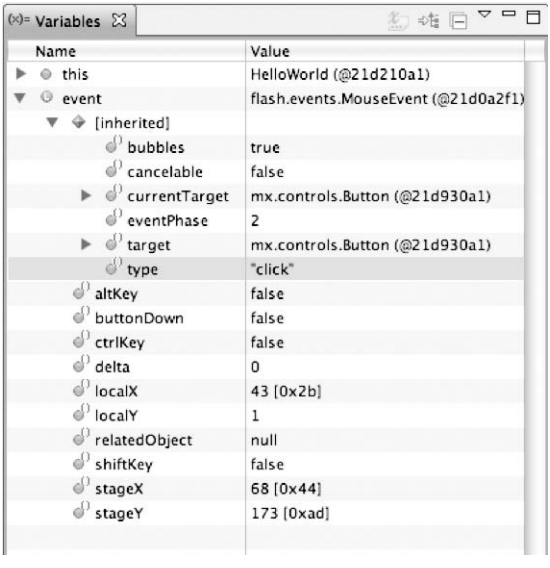

Рис. 6.8. *Панель Variables*

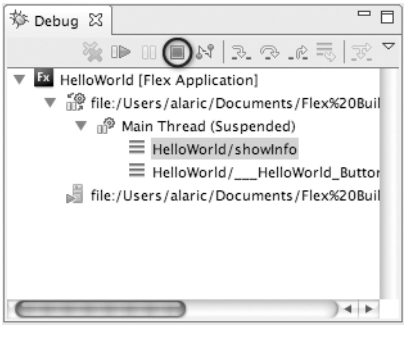

Рис. 6.9. *Панель Debug*

Заключение

Вы узнали много нового о событиях и их использовании в приложении. Не волнуйтесь, если что-то пока кажется непонятным: проектирование взаимодействия с пользователем – самая сложная задача при создании приложения. При необходимости обращайтесь в данной главе еще и еще раз. Что касается вашего приложения, оно не сильно изменилось, но вы узнали крайне важные принципы создания интерактивности, которыми в скором времени сможете воспользоваться на практике.

В следующей главе мы рассмотрим очень полезные возможности Flex, и тогда ваше приложение станет по-настоящему функциональным.

В этой главе:

- Что такое связывание данных?
- Как его использовать?
- Двунаправленное связывание
- Хранение структурированной информации
- Создание связываемых переменных с помощью ActionScript
- Когда не стоит использовать связывание данных
- Применение метода связывания данных на практике

7

Использование связывания данных

Допустим, ваше приложение хранит информацию об отображаемом имени пользователя (к примеру, Jed90210), и вы хотели бы, чтобы оно появлялось в нескольких местах: на кнопке выхода из приложения, на странице профиля и экране приветствия. Можно просто хранить это имя в виде переменной, на которую приложение будет ссылаться при работе. Но что произойдет, если пользователь решит изменить свое имя, например на Jed75961? В этом случае вам придется написать дополнительный код, отслеживающий обновления отображаемого имени и производящий соответствующие обновления в ссылках на него. Даже если имя отображается всего в нескольких местах, объем кода будет весьма существенным.

Использование технологии Flex предлагает альтернативный вариант. Вы можете хранить отображаемое имя в одном месте и ссылаться на него при необходимости. При его изменении все ссылки будут обновлены автоматически. Это становится возможным благодаря методу *связывания данных*.

Что такое связывание данных?

Метод связывания данных является одним из преимуществ Flex – он позволяет легко оперировать информацией. Это простой способ обращения к данным, позволяющий отслеживать их изменения без дополнительных сложностей. Данные могут представлять собой текст, например отображаемое имя, или информацию в виде упорядоченного списка, как, например, набор биржевых сводок, или что-то подобное;

данные – это просто информация в любой форме. По сути, связывание данных предоставляет разработчикам удобный способ передачи и использования информации внутри приложений. Связывание может осуществляться между определенными свойствами различных элементов, между свойствами элемента и моделью данных и между различными моделями данных.

Как его использовать?

Связывание данных открывает перед разработчиком широчайшие возможности, при этом его механизм на удивление прост. Существует несколько вариантов использования этого метода. В зависимости от конкретной задачи и вашего стиля программирования вы можете выбрать наиболее подходящий для вашего приложения вариант.

Основные принципы использования

В качестве примера, демонстрирующего сущность данного метода, приведем процесс связывания свойств различных элементов. В следующем примере свойство text элемента Label привязывается к свойству text элемента TextInput:

```
<mx:TextInput id="helloTextInput" text="Hello, World"/>
<mx:Label text="{ helloTextInput.text }"/>
```

В данном примере фраза «Hello, World» будет отображаться в обоих элементах – TextInput и Label – благодаря осуществлению привязки свойства text элемента Label к соответствующему свойству text элемента TextInput. Фигурные скобки ({ и }), окружающие ссылку на свойство, к которому осуществляется привязка, используются для отличия связываемых данных от обычного текста MXML – если их опустить, свойству text элемента Label будет присвоено значение «helloTextInput.text». Данный пример показывает наиболее простой и часто используемый способ связывания данных во Flex.

При запуске приложения отображаться одинаково будет не только уже существующий текст, но внесенные в TextInput изменения отразятся и в Label. То есть после ввода текста в TextInput этот текст тут же автоматически будет отображаться и в Label.

Обратите внимание, что привязка осуществляется не к элементу TextInput в целом, а только к его свойству text. Таким образом, следующий код является некорректным:

```
<mx:TextInput id="helloTextInput" text="Hello, World"/>
<mx:Label text="{ helloTextInput }"/>
```

При выполнении программы не произойдет ошибки, но и результат будет не слишком впечатляющим. Flex сочтет, что осуществлена привязка ко всему элементу TextInput с идентификатором helloTextInput,

и элемент Label отобразит что-то вроде «ApplicationName0.helloTextInput».

В качестве еще одного примера создадим две переменные для имени и фамилии, используя тег String. Тег String создает строковую переменную, которую затем мы привязываем к свойству text элемента Label.

```
<mx:String id="firstName">Alaric</mx:String>
<mx:String id="lastName">Cole</mx:String>

<mx:Label id="nameLabel" text="{firstName}"/>
```

Несколько приложений в одном проекте

Возможно, вам захочется поэкспериментировать с представленным здесь кодом или попробовать написать собственный, чтобы на практике применить рассмотренные выше приемы. Если вы боитесь испортить свой текущий рабочий проект, можно без труда создать новый проект, руководствуясь сведениями, полученными в главе 2.

Кроме того, вы можете создать новое приложение в уже существующем проекте. Да, Flex позволяет создавать несколько приложений внутри одного проекта. Для этого выделите проект во Flex Navigator, а затем выберите File→New→MXML Application. Остается лишь заполнить поля в появившемся диалоговом окне, и все готово. Обратите внимание, что хотя в проект Flex может входить несколько приложений, одно из этих приложений всегда считается приложением по умолчанию. Статус *приложения по умолчанию* означает, что, если не выбрано какое-либо другое приложение, именно приложение по умолчанию будет выполнено при нажатии на кнопку Запуск. В том случае если в проект входит более одного приложения, при нажатии кнопки Запуск выполняются следующие правила:

- если в момент нажатия на кнопку одно из приложений редактируется в режиме Design или Source, запущено будет именно оно;
- если ни одна из программ в данный момент не редактируется, будет запущена программа, выбранная во Flex Navigator;
- если во Flex Navigator не выбрано ни одно из приложений, будет запущено приложение по умолчанию.

Также следует обратить внимание на выпадающее меню, расположенное рядом с кнопкой запуска. В нем содержится список всех приложений, входящих в состав выбранного в Flex Navigator проекта. Данное меню можно использовать для выбора приложения для запуска.

Обратите внимание, что имя переменной `firstName`, используемое в качестве значения свойства `text`, заключено в фигурные скобки. Благодаря их наличию компилятор Flex присваивает свойству `text` элемента Label значение элемента `firstName`. В противном случае произошло бы присваивание собственно строки firstName. Использование фигурных скобок также указывает Flex на необходимость слежения за изменениями данной переменной. Если вместо Alaric будет введено другое имя, соответствующим образом будет обновлено свойство `text` элемента Label. Фигурные скобки указывают на осуществление привязки.

Переменная `firstName` является *источником* привязки, свойство `text` элемента Label – его *адресатом*.

Примечание

При связывании необходимым условием является задание идентификатора (атрибута `id`) ваших элементов. Для элемента, являющегося источником привязки, необходимо задать `id` для дальнейших ссылок на него.

Множественность адресатов привязки

Выше был описан способ привязки отображаемого имени к различным компонентам приложения. Однако связывание данных не ограничивается единственным источником и адресатом; несколько разных адресатов могут быть привязаны к одному и тому же источнику. Рассмотрим пример, в котором отображаемое имя привязывается к свойству text элемента Label *и* свойству label элемента Button.

```
<mx:String id="displayName">Jed90210</mx:String>
<mx:Label id="nameLabel" text="{displayName}"/>

<mx:Button id="nameButton" label="{displayName}"/>
```

Конкатенация

В выражениях связывания вы не ограничены использованием только свойства или названия переменной. Внутри фигурных скобок вы также можете производить *конкатенацию* (соединение) и иные операции с данными. Например, для создания приветствия можно использовать следующий несложный код:

```
<mx:String id="displayName">Jed90210</mx:String>

<mx:Label text="{'Hello, ' + displayName}"/>
```

В этом примере знак «+» объединяет строки «Hello» и имя «Jed90210».

Возможны и более сложные комбинации, к примеру, если вы захотите соединить имя, фамилию и приветствие. Важно отметить, что при изменении имени или фамилии приветственная строка изменится соответствующим образом.

```
<mx:String id="firstName">Alaric</mx:String>
<mx:String id="lastName">Cole</mx:String>

<mx:Label text="{'Hello, ' + firstName + ' ' + lastName}"/>
```

Строки можно также сцеплять, используя набор фигурных скобок вместо кавычек и знаков плюс, – просто добавив несколько выражений связывания в атрибут. При выполнении следующего примера мы получим такой же результат, как и в вышеописанном примере.

```
<mx:Label text="Hello, {firstName} {lastName}"/>
```

В этом случае обычный текст соединяется с выражением связывания. Элемент Label содержит текст «Hello», затем следует выражение связывания, далее пробел и снова выражение связывания. Естественно, при запуске приложения отображается текст «Hello, Alaric Cole».

Закавыки с кавычками

Обратите внимание, что в выражениях привязки, расположенных внутри фигурных скобок, обычно следует использовать одинарные кавычки. Такая синтаксическая структура препятствует путанице, связанной с использованием двойных кавычек при обозначении атрибутов в языке XML. Когда компилятор Flex видит набор из открывающих и закрывающих двойных кавычек, он обрабатывает их как обрамление атрибута, и при появлении дополнительных двойных кавычек может возникнуть неоднозначность интерпретации.

В действительности важнее всего, чтобы открывающие кавычки соответствовали закрывающим; для этого можно даже поменять местами одинарные и двойные кавычки, как в следующем примере:

```
text='{"Hello, " + " " + lastName}'.
```

Также для предотвращения ошибок, связанных с использованием недопустимых символов, можно использовать режим Design во Flex Builder. Выберите элемент в данном режиме и найдите свойство, которое хотите изменить, на панели Flex Properties. При вводе значения или выражения привязки, Flex Builder автоматически проверяет допустимость символов, при необходимости используя *экранирующие символы* (*escape characters*). Это определенные последовательности символов, которые служат для замены символов, которые могут вступить в противоречие с исходным кодом. При вводе кавычек в режиме Design, Flex Builder, вероятно, заменит их на символы ", интерпретируемые как двойные кавычки.

Пробелы между фигурными скобками и связываемыми свойствами не оказывают влияния на работу программы, в то время как пробелы за пределами фигурных скобок позволяют разделить отображаемые строки.

Между прочим, многим Flex-разработчикам нравится использовать пробелы между скобками и ссылкой на переменную для удобства зрительного восприятия кода. Например, так:

```
<mx:Label text="Hello, { firstName } { lastName }"/>
```

Более подробное описание можно найти на рис. 7.1.

***Рис. 7.1.** Использование пробелов в выражениях привязки*

Другие функции фигурных скобок

Синтаксическая конструкция с фигурными скобками использовалась нами для изменения или конкатенации привязываемых строк и автоматического обновления взаимосвязанных полей. Функция фигурных скобок здесь двояка: они используются для определения связываемых свойств и отличия динамического текста от обычного. Однако фигурные скобки можно использовать и для выделения динамического текста вне связывания. Далее я привожу пример простой математической операции, размещенной внутри тега, в которой соединяется динамический и обычный текст:

```
<mx:Label text="Eleven times forty-two equals {11 * 42}"/>
```

Пример использования сразу нескольких пар фигурных скобок:

```
<mx:Label text="Hey {firstName}, eleven times forty-two equals
  {11 * 42}"/>
```

Тег <mx:Binding/>

При создании крупных приложений бывает удобно хранить данные о связывании отдельно. В предыдущих примерах заключение значения атрибута `text` тега `Label` в фигурные скобки интерпретировалось Flex как его привязка к определенной переменной, например имени пользователя. Это правильный и наиболее часто встречаемый способ связывания. Однако по некоторым причинам, будь то особенности структуры вашего приложения или вашего стиля программирования, вы можете посчитать, что элемент Label содержит слишком много ин-

формации о том, к чему он привязан, т. к. он явно ссылается на соответствующую переменную (в нашем случае это firstName).

Основные принципы использования

Для хранения выражений привязки в отдельном месте, вне тегов компонентов MXML предлагает следующий способ. Чтобы указать, что к чему привязано, используется тег <mx:Binding/>:

```
<mx:String id="firstName">Alaric</mx:String>

<mx:Label id="nameLabel"/>

<mx:Binding source="firstName" destination="nameLabel.text"/>
```

В данном случае тег <mx:Binding/> содержит информацию о том, что переменная firstName привязана к свойству text элемента Label с идентификатором nameLabel. Для данного тега необходимо установить значения свойств source и destination, поэтому очень важно четко различать, что является источником, а что адресатом привязки. Тег <mx:Binding/> ссылается как на переменную firstName, так и на элемент Label, поэтому последнему необходимо задать идентификатор id.

Примечание

Значения свойств source и destination тега <mx:Binding/> *не заключаются в фигурные скобки,* поэтому следующая запись некорректна: <mx:Binding source="{firstName}" destination="{nameLabel.text}"/>.

Множественность источников привязки

Мы уже говорили об использовании нескольких адресатов привязки, и вам известно, что, к примеру, имя пользователя может отображаться и в поле Label, и на кнопке Button. Одним из преимуществ тега <mx:Binding/> является возможность задания нескольких источников привязки для одного адресата. К примеру, Flex позволяет привязать значение свойства text элемента Label к нескольким различным источникам. Этого можно достичь использованием нескольких тегов <mx:Binding/> (но не нескольких атрибутов внутри одного тега).

```
<mx:Binding source="oneTextInput.text" destination=
"confusedLabel.text"/>
<mx:Binding source="anotherTextInput.text" destination=
"confusedLabel.text"/>

<mx:TextInput id="oneTextInput"/>
<mx:TextInput id="anotherTextInput"/>

<mx:Label id="confusedLabel"/>
```

В данном примере свойство text элемента Label привязывается к двум различным текстовым полям TextInput. При изменении значения поля с идентификатором oneTextInput оно передается элементу Label. При изменении поля anotherTextInput элементу Label передается его значение.

В нашем случае для определения привязки совсем не обязательно использовать теги <mx:Binding/>. Назначение нескольких источников привязки с помощью только фигурных скобок невозможно, но последние можно сочетать с использованием тегов. Однако порой такая запись может стать слишком запутанной. Посмотрите на следущий код: с точки зрения функциональности он аналогичен приведенному выше примеру, но в нем для привязки текста, отображаемого элементом Label, к двум различным полям TextInput используется один тег <mx:Binding/> и одно выражение в фигурных скобках.

```
<mx:Binding source="oneTextInput.text" destination="confusedLabel.
text"/>

<mx:TextInput id="oneTextInput"/>
<mx:TextInput id="anotherTextInput"/>

<mx:Label id="confusedLabel" text="{anotherTextInput.text}"/>
```

Примечание

Тег <mx:Binding/> может располагаться в любом месте вашего кода между открывающим и закрывающим тегами Application, но только не внутри контейнеров. Рекомендуется размещать данные теги в одном месте, лучше всего в начале кода приложения. Если вы попытаетесь разместить тег <mx:Binding/> в контейнере или ином неподходящем месте, Flex выдаст предупреждение «<mx:Binding/> is not allowed here» («<mx:Binding/> нельзя разместить здесь»). Вполне однозначно, правда?

Тег <mx:Binding/> или фигурные скобки?

Использование тега <mx:Binding/> или фигурных скобок – всего лишь два разных способа достижения одной и той же цели. Тег <mx:Binding/> позволяет хранить определения всех привязок в одном месте, что весьма упрощает впоследствии внесение в них необходимых изменений. Однако существенным недостатком такого подхода является отсутствие возможности создания сложных схем привязки. Ваши возможности ограничены одним адресатом и одним источником привязки, в отличие от случая с использованием фигурных скобок. Однако хранение наборов данных в отдельном месте позволит вам осуществлять более сложную привязку. Подробнее об этом в разделе «Хранение структурированной информации», расположенном чуть ниже.

Двунаправленное связывание

По своей природе связывание данных – процесс односторонний. Вы определяете источник и адресат привязки, и информация передается от источника к адресату. Однако ваши возможности не ограничены таким механизмом. Можно осуществлять и двустороннее связывание следующим образом:

И еще один способ связывания данных

Во Flex существует еще один способ связывания данных. Он не слишком часто используется и относится к усложненным приемам, поэтому его подробное рассмотрение выходит за рамки данной книги, но разработчику полезно знать о том, что такая возможность в принципе существует. Речь идет о связывании данных средствами ActionScript – с помощью класса `BindingUtils`. Зачем это может понадобиться? Такой метод обеспечивает полный контроль за осуществлением связывания и позволяет включать и выключать связывание при необходимости. Если возможностей тега `<mx:Binding/>` или фигурных скобок недостаточно для достижения ваших целей, ознакомьтесь с документацией Adobe Flex 3 Language Reference по теме класса `mx.binding.utils.BindingUtils` (она доступна через меню Help→Help Contents).

```
<mx:TextInput id="oneTextInput" text="{anotherTextInput.text}"/>
<mx:TextInput id="anotherTextInput" text="{oneTextInput.text}"/>
```

В этом случае два элемента TextInput привязаны друг к другу. При изменении значения одного из них значение второго изменяется соответствующим образом. Того же самого можно достичь и используя тег `<mx:Binding/>`:

```
<mx:Binding source="oneTextInput.text" destination=
  "anotherTextInput.text"/>
<mx:Binding source="anotherTextInput.text" destination=
  "oneTextInput.text"/>

<mx:TextInput id="oneTextInput"/>
<mx:TextInput id="anotherTextInput"/>
```

Хранение структурированной информации

Flex – это и удобство хранения структурированной информации в модели данных. *Модель данных* – это отдельный объект с некоторым количеством задаваемых свойств, т. е. способ хранения информации в одном месте. Например, вместо того, чтобы создавать отдельные переменные, содержащие имя и фамилию человека, не лучше ли включить эти данные в одну переменную `name`, обладающую свойствами `first`, `middle`, `last`, даже `title`, `suffix` и т. д.? Это и есть модель данных. Можно усложнить задачу и создать модель данных, содержащую полную информацию о пользователе, включая его адрес, телефон, адрес электронной почты и даже собственно этот объект `name`.

Основные принципы использования

Использование моделей данных существенно упрощает организацию вашего кода, что является неоспоримым преимуществом. Если со временем вам понадобится осуществлять доступ к данным, хранящимся на сервере (о чем я расскажу подробнее в главе 9), гораздо практичнее сразу запрашивать совокупность данных, чем каждый раз посылать запрос, чтобы получить сначала имя, потом адрес электронной почты, адрес и т. д. Ведь можно хранить всю информацию в модели данных, тогда достаточно будет одного запроса для ее получения. Для этого используется тег `<mx:Model/>`. Ниже вы увидите пример создания модели данных с именем `model` при помощи данного тега:

```
<mx:Model id="model">
  <info>
    <name>
      <firstName>Tim</firstName>
      <lastName>O'Reilly</lastName>
    </name>

    <email>tim@oreilly.com</email>

    <phone>(707)827-7000</phone>

  </info>
</mx:Model>

<mx:Binding source="areaCode.phone" destination="nameLabel.text"/>

<mx:Label id="nameLabel"/>
```

`<mx:Model/>` – единственный MXML-тег, внутри которого можно структурировать данные с помощью XML. Таким образом, четко организованная и представленная в удобном для восприятия виде информация о пользователе хранится в одном месте. Сведения об имени пользователя, адресе его электронной почты и номере телефона созданы при помощи XML-тегов. Обратите внимание на корневой тег `<info/>`. Его наличие необходимо для безошибочной интерпретации XML-тегов, но он может называться и по-другому – если хотите, придумайте ему свое имя.

Для доступа к этой информации в выражении привязки необходимо указать идентификатор модели данных и следующее через точку свойство. К примеру, обращение к номеру телефона пользователя будет выглядеть так: `model.phone`.

Примечание

Для доступа к номеру телефона используется выражение `model.phone`, а не `model.info.phone`. Это обусловлено особенностями тега `<mx:Binding/>`: идентификатор `model` ссылается прямо на корневой узел данных, расположенных в данной модели.

Многоуровневое связывание

А теперь самое интересное: выражения привязки можно использовать *внутри* модели! В следующем примере создается строковая переменная с именем `areaCode`, содержащая текст «707». Использование выражений привязки, заключенных в фигурные скобки, между тегами `<mx:Binding>` и `</mx:Binding>` добавляет код страны в начало телефонного номера пользователя, являющегося частью данной модели.

```
<mx:String id="areaCode">707</mx:String>

<mx:Model id="model">
  <info>
    <name>
      <firstName>Tim</firstName>
      <lastName>O'Reilly</lastName>
    </name>

    <email>tim@oreilly.com</email>

    <phone>{areaCode}827-7000</phone>

  </info>
</mx:Model>

<mx:Binding source="model.phone" destination="nameLabel.text"/>

<mx:Label id="nameLabel"/>
```

Как и в предыдущих случаях, свойство `text` элемента Label привязывается к телефонному номеру в этой модели данных. При запуске программы текст, отображаемый элементом Label, будет выглядеть следующим образом: «707827-7000».

Иными словами, было осуществлено многоуровневое связывание. Прежде всего, при создании телефонного номера используется привязка к коду страны. Затем осуществляется привязка элемента Label к значению телефонного номера. Таким образом, изменение кода страны влечет за собой соответствующее изменение номера телефона и отображаемого в Label текста. Это очень мощный инструмент с широчайшими возможностями.

В моделях данных вам доступны все функции метода связывания с использованием фигурных скобок, в том числе приведение типов данных. Это означает, что внутри модели можно сцеплять строки и использовать несколько синтаксических конструкций с фигурными скобками. Иногда это называется *подгонкой данных* (*data massaging*) – это конкатенация строк, выполнение математических операций и, возможно, форматирование данных для отображения (о последнем читайте в главе 9). Мы рассмотрели простой пример, в котором для получения полного номера телефона объединялись код города и номер телефона. Можно пойти еще дальше и расширить нашу модель данных таким образом:

```
<mx:Model id="model">
  <info>

    <name>
      <title>Mr.</title>
      <firstName>Tim</firstName>
      <lastName>O'Reilly</lastName>
      <displayName>{model.name.title} {model.name.lastName}
        </displayName>
    </name>

    <greeting>Hello, {model.name.displayName}</greeting>

    <email>tim@oreilly.com</email>

    <areaCode>707</areaCode>

    <phone>{model.areaCode}827-7000</phone>

  </info>
</mx:Model>

<mx:Binding source="model.greeting" destination="nameLabel.text"/>
```

Такой код, возможно, покажется вам излишне усложненным, но он ярко иллюстрирует широчайшие возможности метода многоуровневого связывания данных. В данном случае вы создали свойство `displayName` и привязали его к фамилии пользователя и способу обращения к нему. В свою очередь это свойство используется в свойстве `greeting`, которое также привязано к элементу Label. Ну и ну!

Примечание

Для более сложной схемы подгонки данных, вероятно, будет удобнее использовать модели на основе классов. Более подробную информацию об этом можно найти в документации Flex.

Создание связываемых переменных с помощью ActionScript

До сих пор для осуществления связывания мы использовали свойства элементов, теги `<mx:Binding/>` и `<mx:Model/>`, и все отлично работало. Однако при создании переменных в ActionScript необходимо явно объявлять их связываемыми. В противном случае при запуске приложения все будет работать как положено, но в дальнейшем изменения связанных переменных отображаться не будут, то есть вначале все значения будут скопированы в соответствии со схемой связывания, но при обновлении значения какой-либо переменной связанные с ней переменные обновлены не будут.

Теги `<mx:String/>` и `<mx:Model/>` и большинство свойств элементов во Flex приспособлены к работе со связыванием данных. Но при создании пе-

ременных, которые будут участвовать в связывании, вам придется совершить несколько дополнительных действий. Чтобы значения ActionScript-переменных могли участвовать в связывании, необходимо использовать специальное обрабатываемое компилятором Flex объявление метаданных, заключенное в квадратные скобки. Для этого перед объявлением переменной поместите тег [Bindable]:

```
<mx:Script>
  <![CDATA[
    [Bindable]
     public var firstName:String = "Alaric";

    [Bindable]
     public var lastName:String = "Cole";
  ]]>
</mx:Script>
```

Этот код аналогичен рассмотренному ранее примеру с использованием тегов <mx:String/> с той лишь разницей, что в данном случае вместо MXML применяется ActionScript.

Зачем нужно добавлять эти метаданные? На самом деле, если вы создаете переменные со связыванием посредством тега или сценария, Flex автоматически пишет большое количество дополнительного кода, задавая обработчики, следящие за изменениями переменных. Если бы вы не могли задавать, будет ли данная переменная участвовать в связывании, создавалось бы большое количество ненужного кода для переменных, которые ни к чему не привязаны. Это привело бы к увеличению размера приложения и снижению его производительности.

Когда не стоит использовать связывание данных

Вы могли убедиться, что связывание данных – прекрасный инструмент управления информацией в вашем приложении. Так неужели есть причины его не использовать? Дело в том, что связывание данных – не универсальное средство решения любых задач, и иногда далеко не лучшее. Если ваше приложение отображает данные в зависимости от работы некого временного механизма, использование связывания данных может оказаться неверным решением, т. к. при связывании вы не можете контролировать момент обновления значений – данные обновляются автоматически. (Однако при создании связывания посредством ActionScript (о чем подробнее говорится во врезке «И еще один способ связывания данных») для вас не существует таких ограничений.)

При изменении значения источника тут же обновляются значения адресатов связывания, и если в вашем приложении множество свойств привязано к одному источнику с большим объемом часто обновляемой информации, изменения будут происходить, возможно, чаще, чем вам бы того хотелось. Это также может влиять на производительность программы, т. к. слишком много операций будет происходить одновременно.

В этом случае удобнее задавать значения переменных вручную посредством сценариев. Однако в подавляющем большинстве случаев связывание данных станет для вас незаменимым инструментом для решения многих задач.

Применение метода связывания данных на практике

Теперь попробуем применить полученные знания о связывании данных на практике. Создайте новый Flex-проект с названием `ContactManager` и разместите следующий код в основном файле приложения. Сохраните файл под именем *ContactManager.mxml*.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application
  xmlns:mx="http://www.adobe.com/2006/mxml"
  layout="absolute">

  <mx:Panel
    x="10"
    y="10"
    layout="vertical"
    title="Contact Editor"
    paddingLeft="5"
    width="200"
    height="300">

    <mx:Label
      text="First Name"
      fontWeight="bold"/>
    <mx:TextInput id="firstNameTextInput"/>
    <mx:Label
      text="Last Name"
      fontWeight="bold"/>
    <mx:TextInput id="lastNameTextInput"/>
    <mx:Label id="ageLabel"
      text="Age"
      fontWeight="bold"/>
    <mx:NumericStepper id="ageNS"
      maximum="120"
      minimum="18"/>
    <mx:CheckBox id="dogsCheckBox"
      label="Likes Dogs"/>
    <mx:Label
      text="Favorite Color"
      fontWeight="bold"/>
    <mx:ColorPicker id="favoriteColorPicker"
      selectedColor="#FFFFFF"/>
```

```
    </mx:Panel>
</mx:Application>
```

Этот код создает панель Panel с именем Contact Editor, содержащую несколько различных элементов управления. С одним из них вы еще не сталкивались – я говорю о ColorPicker. Данный элемент функционирует подобно аналогичным инструментам графических программ (таких, как Adobe Photoshop) или панели Properties в режиме Design – он позволяет подбирать цвета с помощью выпадающего списка с образцами. В данном случае свойству `selectedColor` присваивается шестнадцатеричное значение, которое является способом компактного представления десятичного числа (подробнее об этом – во врезке «Цвет роз – FF0000, фиалок – 0000FF»). Таким образом, цвет представлен в виде числового значения, которое можно использовать в выражениях привязки. (В этом вы скоро убедитесь сами.)

Примечание

Форматирование MXML-кода не является необязательным. Но для удобства восприятия можно располагать атрибут `id` на одном уровне с содержащим его тегом, а остальные атрибуты – на отдельных строках.

Цвет роз – FF0000, фиалок – 0000FF

Шестнадцатеричная система счисления позволяет представить 24-битные цвета в более компактном виде. На самом деле ее основная функция – сжатое представление больших чисел, но в данном случае я упоминаю о ней в связи с числовым представлением цветов. Значение цвета во Flex передается с помощью *шестнадцатеричных* **троек**, имеющих вид RRGGBB, где RR определяет количество красного, GG – зеленого, а BB – синего. В данной книге мы не будем подробно рассматривать шестнадцатеричную систему; достаточно сказать, что ее основание равняется 16 (а не 10, как в десятичной системе, к которой мы все привыкли), и таким образом существует 16 **«цифр»**: 0–9 и A–F (которыми можно представить числа от 10 до 15). Полный список шестнадцатеричных цифр – 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E и F. К примеру, черный цвет представлен в виде 000000, что означает полное отсутствие красного, зеленого и синего. Чистый красный цвет обозначается FF0000, потому что красная компонента на максимуме, а синяя и зеленая на нуле. По аналогии зеленый – это 00FF00, а синий – 0000FF. В результате смешивания чисто красного (FF0000) и чисто синего (0000FF) получится фиолетовый (FF00FF). Если вы не забыли, что белый – это смесь всех цветов, то уже наверняка догадались, что он обозначается FFFFFF.

Вам незачем писать все это вручную, но понимание принципов такой системы поможет вам разобраться во многих особенностях вашего кода. Стоит отметить, что шестнадцатеричные значения во Flex могут быть представлены несколькими способами. Во-первых, можно использовать знак «решетка» (#) перед собственно значением. Таким образом, красный цвет можно представить следующим образом: #FF0000. Этот метод используется в CSS, и его можно успешно использовать в MXML-коде. Во-вторых, если вы работаете с цветами или иными шестнадцатеричными значениями в ActionScript, следует использовать формат 0x плюс значение. К примеру, красный можно представить как 0xFF0000. Поначалу, если вам нужно установить шестнадцатеричное значение цвета, удобно воспользоваться режимом редактирования Design в Flex Builder. С помощью панели Properties можно с легкостью подбирать цвета в визуальном режиме.

Если вас увлек мой рассказ о шестнадцатеричной системе счисления, вам, возможно, будет интересно узнать о происхождении самого слова «шестнадцатеричный». В английском языке это слово звучит как «hexadecimal». «Неха» восходит к греческому ἑξ (hex), что означает «шесть», а «decimal» – к латинскому слову, обозначающему «десять». Если это для вас равносильно китайской грамоте, отмечу, что некоторые люди считают, что данное слово должно звучать «sexadecimal» – это будет чисто латинской формой.

Затем создайте еще одну панель и назовите ее Contact Details. В нее мы добавим несколько элементов Label, привязанных к соответствующим элементам панели Contact Editor. Поместите нижеследующий код под первой панелью (но, конечно, до закрывающего тега Application).

```
<mx:Panel
  layout="vertical"
  x="227"
  y="10"
  width="200"
  height="300"
  paddingLeft="5"
  title="Contact Details">
  <mx:Label
    text="Full Name:"
    fontWeight="bold"/>
  <mx:Label
    text="{firstNameTextInput.text} {lastNameTextInput.text}"/>
  <mx:Label
    text="Age:"
```

```
      fontWeight="bold"/>
    <mx:Label
      text="{ageNS.value} years old"/>
    <mx:Label
      text="Likes Dogs:"
      fontWeight="bold"/>
    <mx:Label
      text="{dogsCheckBox.selected}"/>
    <mx:Label
      text="Favorite Color:"
      fontWeight="bold"/>
    <mx:Canvas
      width="60"
      height="60"
      backgroundColor="{favoriteColorPicker.selectedColor}"/>
  </mx:Panel>
```

Это простой пример связывания данных, в котором информация, вводимая посредством элементов одной панели, отображается на другой панели. На панели Contact Editor вы видите несколько элементов TextInput, один NumericStepper, CheckBox и ColorPicker. При изменении их значений происходит мгновенное обновление элементов панели Contact Details. При запуске приложения вы увидите следующее (рис. 7.2).

Рис. 7.2. *Простое приложение, демонстрирующее возможности связывания данных*

Стоит обратить внимание на контейнер Canvas, используемый для отображения любимого цвета. Он обладает свойством backgroundColor, значение которого привязано к выбранному в ColorPicker цвету. Тип присваиваемых данному свойству значений – числовой. Теперь при изменении пользователем цвета при помощи ColorPicker им заполняется фон Canvas. В главе 10 вы узнаете, что Canvas – это контейнер расположения, который позволяет размещать внутри элементы в соответствии с заданными координатами x и y. Но его можно использовать и в качестве простого графического компонента, предназначенного в данном случае для отображения квадрата, заполненного цветным фоном.

Примечание

Связывание свойства backgroundColor невозможно осуществить с помощью тега <mx:Binding/>, т. к. оно относится к категории стилей и отличается от обычных свойств. <mx:Binding/> можно использовать для связывания обычных свойств, стили же можно связывать лишь посредством фигурных скобок.

Вы можете захотеть попробовать установить цвет фона основного приложения посредством связывания, подобно тому, как это делалось в приведенном примере. В этом есть смысл, ведь свойством backgroundColor обладают как Canvas, так и другие контейнеры. Однако тег <mx:Application/> не позволяет использовать связывание в своих свойствах.

Осуществить установку цвета фона приложения с помощью ColorPicker можно посредством написания сценария. Проще всего отслеживать изменения выбранного в ColorPicker цвета и прямо задавать его в качестве фона приложения с помощью метода setStyle() тега <mx:Application/>.

Этот метод работает со всеми компонентами Flex, поэтому стоит рассмотреть его действие подробнее. У него есть два параметра – название изменяемого стиля и присваиваемое ему значение (тип данных String). В примере с цветом фона это стиль backgroundColor, а его значением должен стать выбранный с помощью ColorPicker цвет. Таким образом, в нашем случае вызов метода будет иметь вид setStyle('backgroundColor', favoriteColorPicker.selectedColor). Вызов метода следует осуществить при изменении значения ColorPicker. Так как при этом происходит событие типа change, можно установить его обработчик следующим образом:

```
<mx:ColorPicker id="favoriteColorPicker"
  change="setStyle('backgroundColor',
    favoriteColorPicker.selectedColor)"/>
```

Теперь при изменении значения ColorPicker происходит вызов данного метода, изменяющего цвет фона приложения.

Однако данное событие не происходит при первоначальной загрузке приложения, поэтому фон приложения может не соответствовать вы-

бранному в текущий момент цвету в ColorPicker. Для решения этой проблемы нужно осуществить вызов данного метода по событию `applicationComplete` тега `Application`. Данное событие происходит при полной загрузке приложения, когда созданы все компоненты верхнего уровня. Если данное событие произошло, вы можете быть уверены, что элемент с идентификатором `favouriteColorPicker` создан и его свойства

Преобразование типа связанных данных

В результате связывания данных их тип может быть изменен – на этом следует остановиться подробнее. В качестве примера возьмем элемент CheckBox. Его значением могут быть данные типа Boolean, т. е. либо `true` (при условии, что флажок установлен), либо `false` (в противном случае). Однако вы скорее всего не задумывались о том, что происходит, когда вы привязываете такое значение к свойству `text` элемента Label, а ведь оно предназначено для текстовых данных, а не данных Boolean. Таким образом, при осуществлении связывания это значение преобразовывается в текстовое, т. е. в соответствующие строки «true» и «false».

Подобное преобразование происходит и при привязке к свойству `value` элемента NumericStepper. Числовое значение этого свойства (типа Number) конвертируется в текстовое (типа String). Если вам интересно узнать, что именно делает при этом Flex, я объясню подробнее. Каждый Flex-объект имеет метод `toString()`, возвращающий значение типа String. В случае со значением Number данный метод возвращает текстовое представление числового значения. При обработке связанных данных Flex вызывает данный метод, когда это необходимо. В противном случае при запуске программы вы столкнулись бы с ошибками из-за несоответствия типов данных и невозможности их преобразования. К счастью, Flex решит эту задачу автоматически.

Однако все вышесказанное справедливо лишь для преобразования данных в текстовые. Если вам понадобится привязать значение свойства `value` элемента NumericStepper к свойству `text` элемента Label, т. е. поменять местами адресат и источник привязки, Flex выдаст ошибку «Implicit coercion of a value of type String to an unrelated type Number[1]», что означает, что преобразование текстового значения в числовое невозможно.

Чтобы избежать подобной ситуации, можно преобразовать текстовое значение в числовое следующим образом: `<mx:NumericStepper value="{Number('33')}"/>`.

---

[1] «Неявное преобразование значения типа String к не связанному с ним типу Number» (*англ.*). – *Прим. перев.*

доступны. При запуске приложения происходят и другие события, например initialize, но в нашем случае лучший и надежный выбор – applicationComplete, поскольку данное событие происходит последним.

Для решения подобной задачи рекомендуется создать функцию, вызывающую метод setStyle(), и осуществить ее вызов по событиям applicationComplete и change, как в следующем примере:

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

  <mx:Script>
  <![CDATA[
    public function modifyBackgroundColor():void
    {
      setStyle('backgroundColor', favoriteColorPicker.selectedColor);
    }
  ]]>
  </mx:Script>

<!-- Код приложения -->

<mx:ColorPicker id="favoriteColorPicker"
  selectedColor="0xFFFFFF"
  change="modifyBackgroundColor()"/>

<!-- Код приложения -->
```

Заключение

Теперь вы знакомы с механизмом использования конструкций с фигурными скобками. Вы имели возможность убедиться, что технология связывания данных – незаменимый инструмент работы с информацией в приложении, и вы знаете несколько способов работы с ним. Вы создали небольшой пример, демонстрирующий, что Flex позволяет связывать даже значения цветов.

Связывание данных очень часто используется при разработке Flex-приложений, и по ходу прочтения данной книги вы сможете значительно расширить и развить собственные навыки работы с этим методом. Рассмотренные в данной главе основные принципы его использования открывают перед вами новые возможности. В следующей главе я подробно расскажу о построении схемы расположения элементов вашего приложения.

В этой главе:

- Расположение компонентов приложения
- Список отображения
- Размер компонента
- Возможности контейнеров размещения
- Контейнеры с расширенными возможностями
- Элементы расположения
- Выравнивание
- Использование ограничителей

8

Расположение элементов в вашем приложении

Одним из достоинств Flex является наличие удобного механизма разработки схемы расположения элементов в приложении. Если вам приходилось работать в среде Flash IDE, вы знаете, что размещение компонентов интерфейса – одна из самых сложных задач при создании приложения. Иногда вас вполне удовлетворит статичное расположение элементов. В этом случае каждый компонент расположен в определенном месте и имеет четко заданный размер; предполагается, что изменение этих параметров не потребуется. Flash IDE – идеальный инструмент для создания таких интерфейсов. Однако чаще всего схема расположения элементов требует большей гибкости, и здесь Flex как нельзя кстати.

При нынешнем разнообразии разрешений и размеров мониторов обязательное требование к приложению – возможность растягивания и сжатия его элементов. Если ваше приложение запускается через броузер, пользователь может изменить размер его окна. Если вы проектируете настольное приложение при помощи Adobe AIR, функция изменения размера окна может стать его неотъемлемой частью. Возможно, вам потребуется также возможность изменения расположения компонентов в зависимости от текущего размера приложения.

Система проектирования расположения элементов в Flex основана на концепции слоев и ограничителей, так что если у вас есть опыт в сфере веб-дизайна или веб-программирования, он сослужит вам хорошую службу. Однако даже если вы никогда не слышали о каскадных табли-

цах стилей (CSS), режим редактирования Design поможет вам при проектировании привлекательного интерфейса приложения.

Расположение компонентов приложения

Существует несколько способов расположения элементов приложения: можно устанавливать значения координат каждого элемента, использовать контейнеры расположения, ограничители или их сочетание.

Абсолютное позиционирование

При создании нового Flex-приложения в Flex Builder тегу `Application` автоматически присваивается атрибут `layout="absolute"`. Это означает, что элементы будут расположены в соответствии с принципами *абсолютного* позиционирования, т. е. в зависимости от значений заданных координат `x` и `y`.

Таким образом, для расположения элемента в заданном месте вам необходимо задать его координаты `x` и `y`. Такое позиционирование называется абсолютным, в противоположность *относительному* позиционированию, поскольку в этом случае при размещении элемента не учитывается расположение других элементов в том же самом контейнере. То есть элемент располагается внутри контейнера безотносительно остальных его компонентов. При абсолютном позиционировании вы полностью контролируете местонахождение компонентов; при этом компоненты могут даже частично или полностью перекрывать друг друга.

Если вам нужно создать приложение быстро, возможности абсолютного позиционирования работают на вас. Вам достаточно лишь разместить компоненты в режиме Design, а Flex определит значения их координат автоматически. Однако в большинстве случаев при проектировании интерфейса приложения необходим более тонкий поход к вопросам позиционирования, например вам может потребоваться регулировать не только местоположение, но и размер компонента. Для этого применяются ограничители или относительное позиционирование с использованием специальных контейнеров размещения.

Контейнер Canvas, будучи именно контейнером размещения, позволяет размещать элементы абсолютно. Однако большинство контейнеров размещения позиционируют элементы относительно в соответствии с предопределенной для каждого из них схемой.

Относительное позиционирование

При использовании *относительного* позиционирования части вашего приложения расположены в зависимости от расположения других его частей. Как правило, этого можно достичь путем использования контейнеров размещения.

Контейнеры размещения были кратко рассмотрены в главе 3, но теперь имеет смысл поговорить о них подробнее. *Контейнеры* – особые компоненты, позволяющие размещать внутри себя элементы управления и даже другие контейнеры. При написании кода MXML для создания контейнера используется соответствующий тег, например `<mx:Panel/>`, в котором располагается содержимое контейнера. К примеру, в следующем коде создается контейнер Panel со вложенным элементом TextInput:

```
<mx:Panel>
  <mx:TextInput/>
</mx:Panel>
```

Совсем не сложно, не правда ли? Вполне логично, что элемент, расположенный между открывающим и закрывающим тегами `<mx:Panel>` и `</mx:Panel>`, находится внутри панели Panel. Оперируя терминологией Flash, можно сказать, что TextInput является *дочерним* элементом панели Panel. Соответственно Panel называется *родительским* элементом по отношению к TextInput.

Существует два основных вида относительного позиционирования:

Вертикальное

Дочерние элементы расположены последовательно по вертикали. Используется атрибут `layout="vertical"`.

Горизонтальное

Дочерние элементы расположены последовательно по горизонтали. Используется атрибут `layout="horizontal"`.

Такого расположения элементов можно достичь, используя контейнеры, подобные HBox и VBox. Использование таких контейнеров достаточно просто: они размещают любое количество дочерних элементов один за другим по вертикали (HBox) или по горизонтали (VBox).

Настройки по умолчанию и их настройки по умолчанию

По умолчанию в новом Flex-приложении используется абсолютное позиционирование элементов, т. е. при его создании Flex автоматически добавляет атрибут `layout="absolute"` в тег `Application`. То же самое произойдет и с соответствующим тегом элемента Panel при его добавлении на сцену в режиме `Design`. Однако необходимо отметить, что по умолчанию контейнеры Application и Panel на самом деле должны размещать элементы вертикально. То есть при создании элемента без указания значения свойства `layout` его дочерние элементы будут выравниваться по вертикали. Flex Builder перекрывает эту настройку, по умолчанию добавляя атрибут `layout="absolute"`.

При использовании других контейнеров, таких как Panel и Application, перед вами встает возможность выбора из трех вариантов. Такие контейнеры называют также *«смешанными»*, т. к. они могут функционировать аналогично другим контейнерам в зависимости от значений определенных свойств. Удобство их использования неоспоримо, ведь можно изменять расположение элементов в контейнере, всего лишь присваивая свойству `layout` одно из значений `vertical`, `horizontal` или `absolute`.

Чтобы по достоинству оценить преимущества использования контейнеров размещения, нужно вначале разобраться в понятии «список отображения» (display list) во Flex, что поможет вам правильно спроектировать расположение элементов в вашем приложении.

Список отображения

Списком отображения во Flex, а также в любом контенте на базе Flash-технологии называется список всех графических элементов конкретного приложения. Говоря о Flex-приложении, можно представить его как иерархию компонентов, начиная от корневого тега Application и заканчивая элементом Button, расположенным внутри панели Panel. На самом деле он включает еще графические элементы, из которых состоит кнопка, но на начальном этапе изучения Flex достаточно разобраться на уровне цельных компонентов и контейнеров.

Для чего он нужен?

Если вы когда-либо работали в графических редакторах, вам известна концепция использования *слоев*. Расположенные в списке отображения верхние элементы подобно добавленным позже слоям перекрывают находящиеся под ними нижние элементы.

При добавлении элементов к панели Panel средствами MXML они добавляются в список отображения данной панели в том же порядке, в котором они следуют в коде. Вновь проводя аналогию со слоями, представьте, что последние добавленные элементы перекрывают расположенные ранее элементы. Программисты говорят, что такие элементы обладают более высоким *индексом*. Самый маленький индекс – у элемента, расположенного первым; при добавлении каждого нового элемента он увеличивается на единицу. Рассмотрим следующий MXML-код, в котором в контейнер Panel добавляется три контейнера Canvas различных цветов:

```
<mx:Panel id="colorsPanel"
  width="250"
  height="250"
  layout="absolute">

  <mx:Canvas id="redBox"
```

```
      x="70"
      y="70"
      width="50"
      height="50"
      backgroundColor="#FF0000" />
    <mx:Canvas id="greenBox"
      x="90"
      y="90"
      width="50"
      height="50"
      backgroundColor="#00FF00" />
    <mx:Canvas id="blueBox"
      x="100"
      y="60"
      width="50"
      height="50"
      backgroundColor="#0000FF" />
</mx:Panel>
```

Вначале в список отображения добавляется элемент с идентификатором `redBox`, за ним следует `greenBox` – это происходит в соответствии с порядком их появления в MXML-коде. Элемент с именем `blueBox` обладает самым высоким индексом, поскольку он был добавлен последним. На рис. 8.1 показан результат выполнения данного кода.

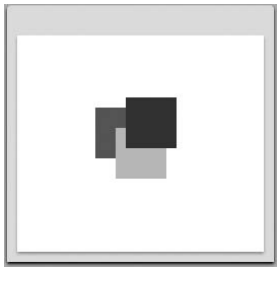

Рис. 8.1. *Перекрытие цветных элементов при абсолютном позиционировании*

Осуществление доступа к дочерним элементам

Доступ к любому дочернему элементу контейнера можно получить по его индексу в списке отображения. В ActionScript для этого используется метод `getChildAt()` с единственным параметром, значением которого может быть целое число, соответствующее номеру элемента, к которому осуществляется доступ. Отсчет начинается с нуля, т. е. элемент с индексом 0 считается первым, с индексом 1 – вторым, с индексом 2 – третьим и т. д. В рассмотренном примере результатом вызова функции `getChildAt(0)` будет `redBox`, а функции `getChildAt(1)` – `blueBox`.

Добавление и удаление дочерних элементов

В главе 4 я уже вскользь упоминал метод `addChild()`. Он используется в ActionScript для *добавления* дочерних элементов в контейнеры или список отображения. Чтобы элемент появился в списке отображения, недостаточно лишь создать его; для его добавления нужно воспользоваться указанным методом. Следующий код на ActionScript эквивалентен рассмотренному выше примеру на MXML:

```
//import the required classes
import mx.containers.Panel;
import mx.containers.Canvas;
public function createBoxes():void
{
  //create a Panel
  var colorsPanel:Panel = new Panel();
  colorsPanel.layout = "absolute";
  colorsPanel.width = 250;
  colorsPanel.height = 250;

  //add the Panel to the Application
  addChild(colorsPanel);

  //create a red box
  var redBox:Canvas = new Canvas();
  redBox.x = 70;
  redBox.y = 70;
  redBox.width = 50;
  redBox.height = 50;
  redBox.setStyle("backgroundColor", 0xFF0000);

  //create a green box
  var g:Canvas = new Canvas();
  greenBox.x = 90;
  greenBox.y = 90;
  greenBox.width = 50;
  greenBox.height = 50;
  greenBox.setStyle("backgroundColor", 0x00FF00);

  //create a blue box
  var blueBox:Canvas = new Canvas();
  blueBox.x = 100;
  blueBox.y = 60;
  blueBox.width = 50;
  blueBox.height = 50;
  blueBox.setStyle("backgroundColor", 0x0000FF);

  //add the boxes to the Panel
  colorsPanel.addChild(redBox);
  colorsPanel.addChild(greenBox);
  colorsPanel.addChild(blueBox);
}
```

Примечание

Для вызова функции createBoxes() недостаточно поместить ее внутри тега <mx:Script/> в MXML-файле – для этого нужно установить соответствующий обработчик событий.

Как и в MXML-коде, Canvas с именем greenBox добавляется в контейнер Panel после redBox, т. к. вызов метода panel.addChild(greenBox) осуществляется после вызова panel.addChild(redBox). Таким образом, greenBox обладает индексом 1, а redBox – индексом 0, как и ранее. Это означает, что позиция greenBox в списке отображения выше, чем позиция redBox, а blueBox расположен выше всех элементов.

При изменении свойства layout панели Panel на vertical цветные квадраты будут расположены по вертикали друг за другом, как показано на рис. 8.2. Их последовательность соответствует порядку их появления в MXML-коде.

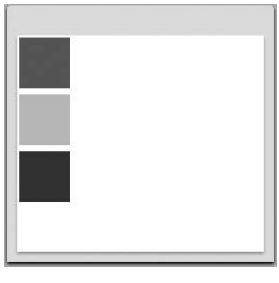

Рис. 8.2. *Вертикальное выравнивание цветных квадратов*

Примечание

При создании элемент не добавляется в список отображения автоматически; так же дело обстоит и с удалением элемента из списка: он более не отображается, но это не значит, что он не существует. Вы можете снова добавить его в список в любое время.

Теперь вы знаете, как добавить элемент в список отображения. Удаление элемента из списка происходит аналогичным образом – при помощи метода removeChild(). Если по каким-либо причинам вам больше не нужен квадрат синего цвета, его можно удалить, вызвав функцию colorsPanel.removeChild(blueBox).

Перестановка дочерних элементов

При желании можно с легкостью поменять расположение элементов в списке отображения. Если вы хотите разместить greenBox поверх остальных элементов, это можно сделать с помощью ActionScript. Вначале для удаления элемента из списка отображения Panel используйте

метод `colorsPanel.removeChild(greenBox)`. Дальнейший вызов функции `colorsPanel.addChild(greenBox)` снова добавит данный элемент в список отображения, который теперь будет обладать более высоким индексом, чем `redBox` и `blueBox`, поскольку был добавлен последним (рис. 8.3). Если вы используете вертикальное выравнивание компонентов в панели Panel, `greenBox` окажется в самом низу списка (рис. 8.4).

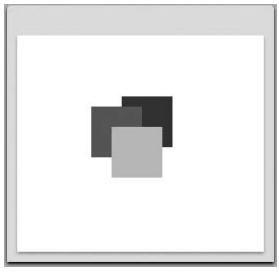

Рис. 8.3. *Результат перемещения элемента greenBox в начало списка отображения*

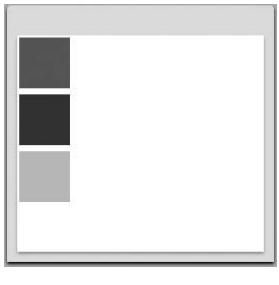

Рис. 8.4. *Вертикальное расположение элементов, показывающее, что элемент greenBox находится вверху списка отображения*

Следующий код поможет вам еще лучше разобраться в особенностях функционирования списка отображения. Здесь к каждому элементу Canvas добавляется обработчик события `click`. При щелчке мышью по цветному квадрату он перемещается в верхнюю часть списка.

Примечание

Поменять местами два элемента в списке отображения можно с помощью метода `swapChildren()`. Например, чтобы поменять местами красный и зеленый квадрат, используйте конструкцию `colorsPanel.swapChildren(redBox, greenBox)`.

```
<mx:Script>
  <![CDATA[
  public function moveUp(event:Event):void
  {
```

```
        // мы предполагаем, что текущая цель события – это Canvas
        var box:Canvas = event.currentTarget as Canvas;
        // удаляем квадрат, по которому щелкнули
        colorsPanel.removeChild(box);
        // помещаем его обратно поверх остальных элементов
        colorsPanel.addChild(box);
      }
    ]]>
  </mx:Script>

  <mx:Panel id="colorsPanel"
    width="250"
    height="250"
    layout="absolute">

    <mx:Canvas id="redBox"
      x="70"
      y="70"
      width="50"
      height="50"
      backgroundColor="#FF0000"
      click="moveUp(event)" />
    <mx:Canvas id="greenBox"
      x="90"
      y="90"
      width="50"
      height="50"
      backgroundColor="#00FF00"
      click="moveUp(event)" />
    <mx:Canvas id="blueBox"
      x="100"
      y="60"
      width="50"
      height="50"
      backgroundColor="#0000FF"
      click="moveUp(event)" />

  </mx:Panel>
```

Примечание

Доступ к компоненту, расположенному внутри другого компонента, нельзя осуществить с помощью точечной нотации. Таким образом, в нашем примере с разноцветными квадратами невозможно получить доступ к красному квадрату, используя выражение `colorsPanel.redBox`. Дело в том, что `redBox` является частью списка отображения панели Panel, а не ее свойством. Поэтому следует использовать метод `colorsPanel.getChildByName ("redBox")`.

Попробуйте поменять способ расположения элементов в контейнере Panel (изменив значение атрибута `layout`) и посмотрите, как будут отображаться элементы при их перемещении.

Внешний вид компонентов

Особенности внешнего вида некоторых контейнеров призваны создать у пользователя определенное впечатление. В качестве примера элемента с характерным внешним видом можно привести Panel. Наличие строки заголовка и границ придают ей схожесть с окном пользовательского интерфейса. Другие контейнеры не обладают какими-либо характерными особенностями с визуальной точки зрения (например, контейнер Canvas). Однако поскольку все они являются компонентами, их внешний вид можно изменять с помощью стилей. К примеру, добавление свойства `backgroundColor` к контейнеру Canvas задаст цвет его фона, как в нашем примере с цветными квадратами. Более подробную информацию о стилях и влиянии их применения на внешний вид приложения можно получить в главе 14.

Обратите внимание, что элемент Panel в свою очередь является частью списка отображения контейнера Application. Перемещение элементов не ограничено лишь содержащим их списком отображения. Например, цветные квадраты могут изменять свое расположение не только внутри Panel, но и перемещаться по всему приложению, например, их можно переместить в другой контейнер, включая Application.

Допустим, в вашем приложении есть две панели, названные `panel1` и `panel2`. Вы можете перемещать дочерние элементы из одной панели в другую. Например, чтобы переместить из `panel1` в `panel2` дочерний элемент с идентификатором `someChild`, вызовите `panel1.removeChild(someChild)`, а затем `panel2.addChild(someChild)`.

Размер компонента

Размер компонента можно регулировать, задавая значение его ширины и высоты. Как правило, они измеряются в *пикселах*. Пиксел – единица измерения, соответствующая мельчайшей части экрана компьютера. При большом увеличении монитора вы сможете увидеть множество квадратиков. Это и есть пикселы.

Однако размер компонента может определяться не только значениями ширины или высоты в пикселах. При размещении элементов внутри контейнера последний не только обеспечивает их выравнивание, но и изменяет свой размер в зависимости от величины дочерних элементов или же изменяет их размеры, чтобы уместить их в доступном пространстве.

Явное задание размеров

Размеры компонента можно задавать явно с помощью свойств `width` и `height`. Кроме того, каждый компонент обладает размером по умолчанию. Если контейнеру (к примеру, VBox, HBox или Panel) не заданы значения высоты и ширины, он автоматически определит свой размер таким образом, чтобы вместить все вложенные элементы.

Если величина контейнера не определена или же установленного пространства недостаточно, его содержимое может быть *обрезано*, т. е. не уместившиеся компоненты видны не будут. В большинстве таких случаев у контейнера появятся полосы прокрутки, с помощью которых пользователь сможет увидеть не уместившиеся в поле зрения объекты.

Примечание

Многие компоненты изменяют свои размеры в зависимости от некоторых свойств. К примеру, если размер CheckBox или Button не задан напрямую, ему автоматически будет отведено пространство, достаточное для размещения его метки.

Относительное задание размеров в процентах

Размеры компонентов можно задавать относительно, т. е. измеряя их значения в процентах, а не пикселах. К примеру, кнопка может иметь ширину 22 пиксела при установке значения ее свойства `width="22"`, но вместо этого можно задать значение, скажем, 50%. При этом элемент будет занимать половину родительского элемента по ширине.

Посмотрите на следующий код, создающий HBox с шириной 400 пикселов и вложенный в него элемент Button с шириной 50%:

```
<mx:HBox width="400">
  <mx:Button label="Button" width="50%"/>
</mx:HBox>
```

На рис. 8.5 видно, что в результате выполнения данного кода кнопка занимает половину пространства содержащего ее контейнера HBox.

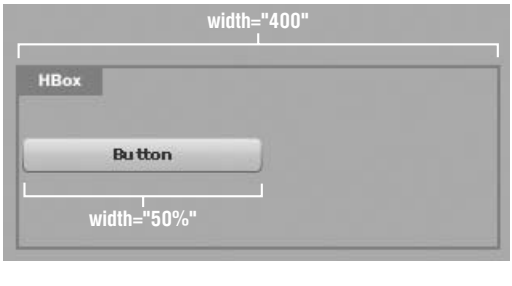

Рис. 8.5. *Ширина элемента, задаваемая в процентах*

Ширина контейнера равняется 400 пикселам, соответственно ширина кнопки составит 200 пикселов. Размеры контейнера HBox также можно задать в процентах. К примеру, установив его значение 100%, вы увидите, что данный элемент заполнит все свободное пространство содержащего его контейнера (т. е. Application), при этом кнопка займет половину *этого* пространства. При изменении размеров приложения соответствующим образом изменятся и размеры HBox и кнопки.

Примечание

Средствами MXML можно задавать значение размеров как в пикселах, так и в процентах, однако в ActionScript значения свойств `width` и `height` могут иметь лишь числовое значение. Для определения процентного значения ширины и высоты элемента в ActionScript используются свойства `percentWidth` и `percentHeight` с диапазоном от 0 до 100 в соответствии с процентным значением.

Что же произойдет, если значение свойства `width` элемента HBox не будет установлено? Ширина кнопки должна составить 100% от ширины родительского элемента, но какова же ширина последнего? В такой ситуации размеры кнопки будут определены настройками по умолчанию, т. е. в соответствии с длиной ее метки.

Примечание

Задавая размеры дочерних элементов контейнера в процентах, убедитесь, что задан размер самого контейнера. В противном случае оба элемента примут размер дочернего элемента, принятое по умолчанию.

Не забывайте, что контейнеры обладают возможностью изменять свои размеры в зависимости от размеров вложенных элементов. Если вы задаете ширину дочернего элемента напрямую или оставляете ее значение по умолчанию, перед вами встает необходимость следующего выбора:

- *Задать* размеры контейнера. Если совокупность размеров дочерних элементов превысит установленное значение, часть содержимого будет обрезана.
- *Не задавать* размеры контейнера. В этом случае размер контейнера будет зависеть от размера вложенных элементов. Но если свободного пространства для размещения контейнера окажется недостаточно, часть содержимого будет обрезана.

Минимальный и максимальный размер элемента

Вы можете также регулировать максимально и минимально возможные значения размеров элемента. Эта опция незаменима при задании ширины и высоты элемента в процентах. К примеру, высота вашей кнопки составляет 100 процентов, но при этом она не должна превы-

шать 300 пикселов. Выполнения этого условия можно достичь с помощью свойства maxHeight следующим образом:

```
<mx:Button height="100%" maxHeight="300" />
```

Аналогичным образом можно использовать свойство maxWidth по отношению к ширине объекта.

Чтобы размеры компонента ни при каких условиях не оказались меньше определенных значений, используйте свойства minWidth и minHeight для регулирования минимальной ширины и высоты компонента соответственно.

Примечание

При задании размеров компонентов в процентах они заполняют доступное свободное пространство. К примеру, если в контейнере HBox с шириной 300 пикселов расположены два элемента Button, ширина каждого из которых равняется 100 процентам, их реальная ширина составит 150 пикселов, т. е. половину свободного пространства. Если бы кнопок было три, ширина каждого из них составила бы 100 пикселов или 33 процента. В случае с двумя кнопками со значениями ширины 100 и 70 процентов первая из них займет примерно 60 процентов пространства, а вторая – примерно 40 процентов.

Возможности контейнеров размещения

Чтобы добиться необходимого расположения элементов с помощью контейнеров размещения, таких как HBox и Panel, в вашем распоряжении есть несколько полезных опций.

Свободное пространство внутри контейнера

Если вы много работали с технологией CSS, то, вероятно, вам хорошо знакомо понятие «padding», т. е. свободное пространство, окружающее расположенные внутри контейнера элементы и позволяющее визуально отделить их от контейнера. Для создания такого пространства в Flex используются свойства paddingLeft, paddingRight, paddingTop и paddingBottom.

Например, при размещении нескольких элементов Button внутри контейнера Panel с вертикальным выравниванием все они будут располагаться у верхней и левой границы панели друг под другом, как бы «прижимаясь» к ее краям. Чтобы добавить немного пустого пространства между краями панели и элементами, попробуйте использовать различные значения вышеописанных свойств (рис. 8.6).

Важно отметить, что данные свойства присущи не только контейнерам, но и некоторым другим элементам – например списку List. В этом случае соответствующие свойства контролируют объем свободного пространства между границами элемента и его содержимым.

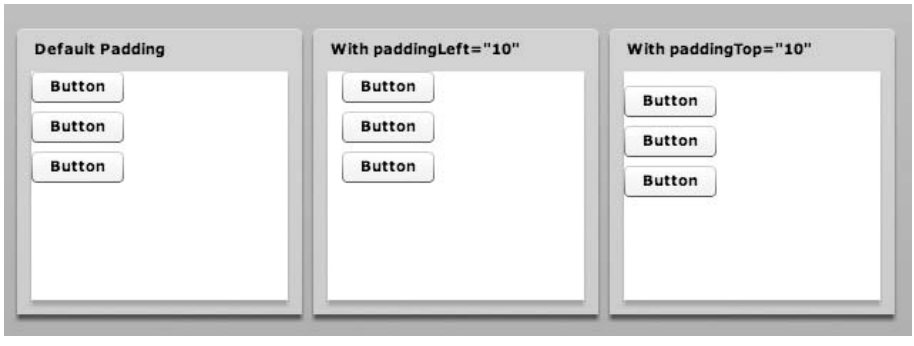

Рис. 8.6. *Различные значения свойств, регулирующих свободное пространство, окружающее внутренние элементы панели Panel*

Промежутки

При использовании большого числа контейнеров размещения можно отделять компоненты друг от друга с помощью промежутков. В вашем распоряжении два независимых свойства, `horizontalGap` и `verticalGap`, задающих объем необходимого пространства между компонентами. Действие данных свойств наглядно представлено на рис. 8.7.

Примечание

Эти свойства легко устанавливать и модифицировать в MXML-коде, но важно помнить, что они относятся к категории стилей, и потому недоступны в ActionScript при помощи точечной нотации. То есть выражение `panel1.paddingLeft=0` не имеет смысла. Для доступа к таким свойствам используйте метод `setStyle()`.

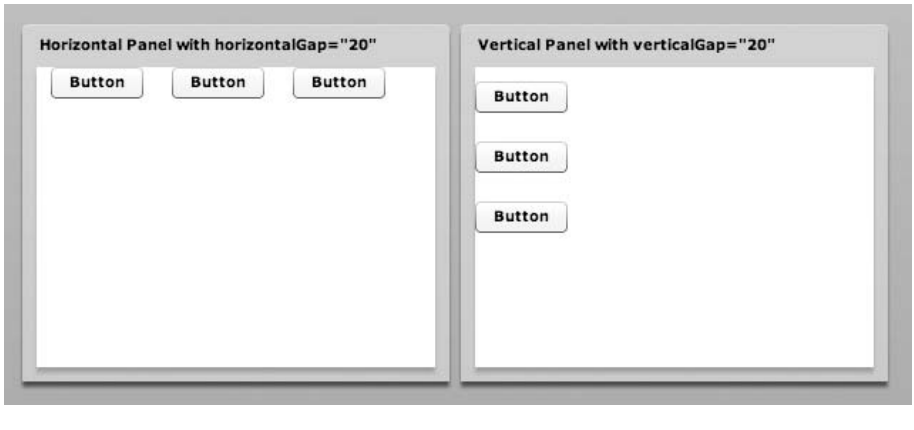

Рис. 8.7. *Панели с различным типом выравнивания и различными промежутками между элементами*

Процесс расположения элементов интерфейса Flex-приложения

Flex обладает чрезвычайно мощным механизмом создания интерфейса приложения, использующим большое число переменных. Как правило, это большое преимущество для разработчика, который тратит меньше времени на размещение компонентов приложения. Если вам интересны принципы работы данного механизма, обратитесь к рис. 8.8, на котором они представлены схематично. В целом расположение элементов рассчитывается в три этапа. Во-первых, устанавливается размер компонентов типа Button и CheckBox, исходя из их меток в том случае, если размеры не заданы напрямую. Затем размеры компонентов обрабатываются, начиная с самого низкого уровня в древовидной структуре приложения. К примеру, в данном коде элемент Button является самым низшим вложенным элементом, далее следует Panel, затем Canvas и, наконец, HBox:

```
<mx:HBox>
  <mx:Canvas width="300">
    <mx:Panel width="100%">
      <mx:Button width="50%"/>
    </mx:Panel>
  </mx:Canvas>
</mx:HBox>
```

***Рис. 8.8.** Три этапа формирования расположения компонентов в Flex-приложении*

На этом этапе обрабатываются только размеры элементов, заданные в пикселах; размеры в процентах рассматриваются на следующем этапе. На завершающем этапе размер и месторасположение элементов определяются в обратном порядке. В вышеприведенном примере вначале будет изменен размер HBox таким образом, чтобы вместить Canvas шириной в 300 пикселов. Следующий в очереди – контейнер Canvas. Поскольку его размер задан напрямую, никакие изменения не требуются. Затем механизм Flex займется панелью Panel. Ее ширина составляет 100 процентов. Соответственно, она должна занять все пространство родительского элемента Canvas. И, наконец, определяется размер Button, который составит половину содержащего его элемента Panel.

При добавлении, удалении компонентов или изменении их размеров (в коде приложения или пользователем при изменении размеров окна приложения) процесс начинается заново.

Контейнеры с расширенными возможностями расположения элементов

Существуют контейнеры с более широкими возможностями с точки зрения размещения элементов, чем простое выравнивание по вертикали или горизонтали. На рис. 8.9 изображен интерфейс, сочетающий несколько таких контейнеров.

DividedBox

Для вертикального и горизонтального выравнивания дочерних элементов используются контейнеры VBox и HBox, но существуют контейнеры с гораздо более широкими возможностями. Функции контейнеров HDividedBox и VDividedBox схожи с действием HBox и VBox с тем лишь отличием, что они позволяют изменять секции, содержащие компоненты.

Между дочерними компонентами такого контейнера есть условный разделитель секций, перемещение которого позволяет изменять их размеры. Такие контейнеры особенно удобно использовать при установке относительных размеров.

Tile

Если вам нужен контейнер расположения «на все случаи жизни», выбор очевиден – это Tile. Расположенные в нем элементы похожи на плитку: они размещаются рядами по горизонтали; после заполнения всего доступного пространства начинается новый ряд. По умолчанию

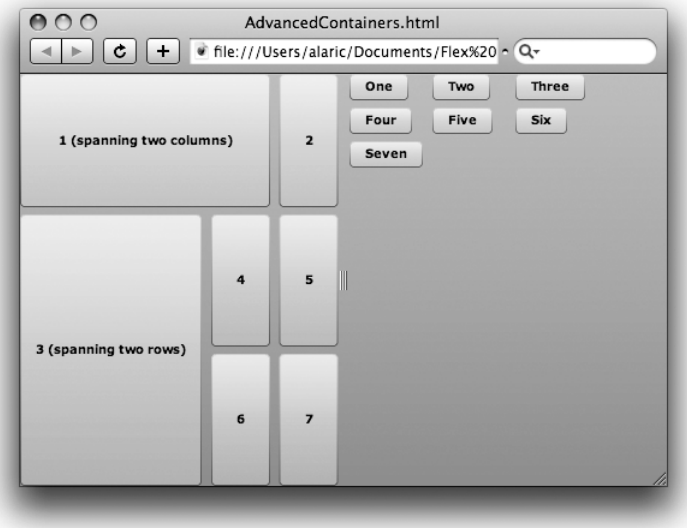

Рис. 8.9. *Сочетание контейнеров с расширенными возможностями расположения элементов: слева Grid, справа Tile, расположенные внутри HDividedBox*

ряды располагаются горизонтально, но с помощью свойства `direction` можно изменить их направление, и вместо рядов элементы будут располагаться колонками.

Одинаковые по размеру «плитки» создают идеальную решетку, причем размер плитки определяется размером самого крупного дочернего элемента, если он не задан напрямую с помощью свойств `tileWidth` и `tileHeight`.

Допустим, в вашем приложении есть несколько элементов для составления панели инструментов. Вы поместили их в верхней части приложения, заключив в контейнер HBox, подобно панели инструментов в Microsoft Word. Однако отказалось, что пространства для всех кнопок панели недостаточно, и часть их была обрезана. Для решения такой проблемы можно использовать контейнер Tile вместо HBox – когда пространство по горизонтали закончится, не уместившиеся кнопки будут просто-напросто расположены в следующем ряду.

Grid

Если вам приходилось когда-либо работать с таблицами гипертекстового языка разметки (HTML), то вам уже известно, что такое Grid. В противном случае вполне достаточно сказать, что контейнер Grid располагает элементы подобно плиткам контейнера Tile с той лишь разницей, что он позволяет дочерним элементам занимать несколько ячеек в ряду или колонке. При работе с электронными таблицами вы

наверняка сталкивались с ситуацией, когда было необходимо объединить пространство, занимаемое двумя смежными ячейками ряда таким образом, чтобы получившаяся ячейка занимала по ширине две колонки вместо одной.

Такая возможность Grid достигается наличием в качестве непосредственных дочерних элементов контейнеров GridRow и GridItem (которые можно в общих чертах сравнить с тегами <tr/> и <td/> HTML-таблицы.) Объединение ячеек по вертикали или горизонтали осуществляется при помощи свойств colSpan и rowSpan.

При перетаскивании на сцену контейнера Grid в режиме Design откроется диалоговое окно, к котором нужно установить требуемое количество рядов и колонок. Это весьма простой и удобный способ создания Grid, ведь в этом случае весь код будет написан автоматически, включая необходимые контейнеры GridItem и GridRow. Затем вам остается лишь переместить нужные компоненты в предназначенные для них ячейки GridItem. Контейнер GridItem может содержать дочерние элементы, что обеспечивает еще большую гибкость при создании пользовательского интерфейса.

Примечание

Если принципы действия контейнера Grid кажутся вам чересчур сложными, то вы можете не использовать его. Большинство его функций доступны вам при использовании ограничителей, о которых мы поговорим позже в этой главе. Однако если у вас богатый опыт работы с HTML-таблицами, вы оцените достоинства данного инструмента.

Form

Расположение элементов в контейнере Form напоминает HTML-формы. Оно достигается благодаря специальному внутреннему контейнеру FormItem, в который заключается элемент формы. Он также может присваивать элементу метку (поэтому нет необходимости использовать элемент Label). Рассмотрим следующий пример:

```
<mx:Form>

  <mx:FormItem label="A Short Label">
    <mx:TextInput/>
  </mx:FormItem>

  <mx:FormItem label="A Very, Very Long Label">
    <mx:TextInput/>
  </mx:FormItem>

</mx:Form>
```

Этот код создает совокупность элементов, представленную на рис. 8.10. Мы видим два вертикально расположенных поля для ввода текста (элементы TextInput) с метками, выровненными соответствующим об-

Рис. 8.10. *Пример простой формы*

разом. Размер меток подогнан специальным образом, так что если одна метка длиннее другой, она немного отодвинет поля формы, чтобы достичь наилучшего взаимного расположения.

Примечание

Контейнер FormItem обладает широкими возможностями для проверки введенных данных. Об этом мы поговорим более подробно в главе 9.

Форма может также содержать элемент FormHeading, задающий ее заголовок. Он выравнивается автоматически по отношению к полям формы.

Формы проще всего создавать в режиме Design – достаточно перетащить контейнер Form на сцену и поместить внутри такие поля формы,

Перемещение по полям формы

Обычно пользователи переходят от одного элемента приложения к другому не только с помощью мыши; с этой целью нередко используется клавиша Tab. Обычно пользователь предполагает, что нажатие клавиши Tab позволяет перемещаться от одного элемента формы к другому, и отсутствие такой возможности вызовет негативную реакцию большинства пользователей.

Как правило, разработчику не стоит даже задумываться об этом, поскольку во Flex по умолчанию существует система навигации с помощью Tab, основанная на близости расположения текущего элемента к определенному полю, которое и будет выбрано при нажатии на клавишу. При использовании таких контейнеров, как Form, при навигации с помощью Tab фокусировка обычно смещается именно на тот элемент, который нужен пользователю.

Для более тонкого контроля за навигацией по компонентам можно использовать свойство tabIndex. Оно принимает значение целого числа, определяющего порядок обхода элементов. Обратите внимание, что необходимо указать значение tabIndex каждого элемента во избежание неправильной интерпретации. Но как правило, такие меры излишни; вполне достаточно использовать контейнеры, располагающие элементы в нужном порядке.

как TextInput, NumericStepper или DateField. При этом Flex Builder генерирует необходимые элементы FormItem автоматически, а для изменения их меток достаточно двойного щелчка мышью. Если метка не нужна, можно оставить пустую строку или вовсе убрать атрибут `label` в режиме Source.

Внимание

Для выравнивания полей формы не забывайте заключать их в контейнер FormItem, даже если вы не хотите присваивать им метку.

Чтобы еще глубже понять работу этого контейнера, вернемся к созданному нами в главе 7 проекту ContactManager и переработаем схему расположения его элементов, используя только что изученный контейнер. Можно внести изменения прямо в режиме Design или заменить панели Panel на контейнеры Form и FormItem:

```
<mx:Form>
  <mx:FormHeading
    label="Contact Editor"/>
  <mx:FormItem
    label="First Name">
  <mx:TextInput id="firstNameTextInput"/>
  </mx:FormItem>
  <mx:FormItem
    label="Last Name">
    <mx:TextInput id="lastNameTextInput"/>
  </mx:FormItem>
  <mx:FormItem
    label="Age">
    <mx:NumericStepper id="ageNS"
      maximum="120"
      minimum="18" />
  </mx:FormItem>
  <mx:FormItem>
    <mx:CheckBox id="dogsCheckBox"
      label="Likes Dogs" />
  </mx:FormItem>
  <mx:FormItem
    label="Favorite Color">
    <mx:ColorPicker id="favoriteColorPicker"/>
  </mx:FormItem>
</mx:Form>
```

```
<mx:Form x="300">
  <mx:FormHeading
    label="Contact Details"/>
  <mx:FormItem
    label="Full Name">
    <mx:Label
      text="{firstNameTextInput.text} {lastNameTextInput.text}"/>
  </mx:FormItem>
```

```
    <mx:FormItem
      label="Age">
      <mx:Label
        text="{ageNS.value} years old"/>
    </mx:FormItem>
    <mx:FormItem
      label="Likes Dogs">
      <mx:Label
      text="{dogsCheckBox.selected}"/>
    </mx:FormItem>
    <mx:FormItem
      label="Favorite Color">
      <mx:Canvas
        width="60"
        height="60"
        backgroundColor="{favoriteColorPicker.selectedColor}"/>
    </mx:FormItem>
  </mx:Form>
```

Приложение ContactManager теперь должно выглядеть примерно как на рис. 8.11. Сохраните этот проект, и на протяжении всей книги мы будем к нему обращаться, внося все новые и новые изменения.

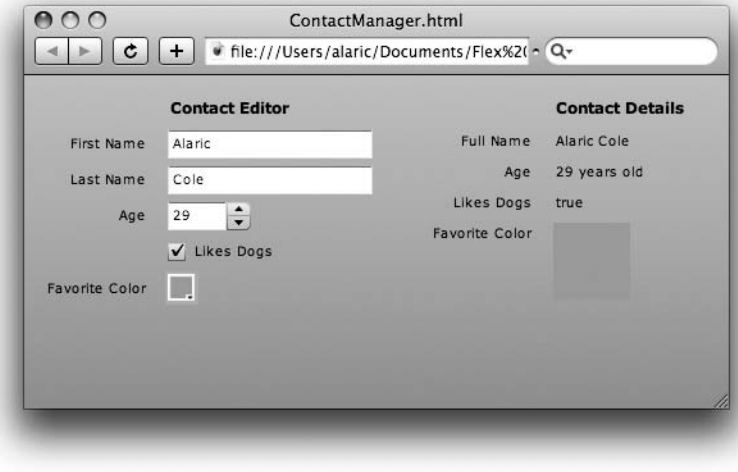

Рис. 8.11. *Вид приложения ContactManager с использованием форм для размещения элементов*

Элементы расположения

Существует несколько *элементов*, используемых для построения схемы расположения компонентов приложения. Они применяются при относительном позиционировании и позволяют добавить в контейнер пустое пространство.

Spacer

Spacer – невидимый элемент, служащий своеобразной «распоркой» для дочерних элементов, заключенных в контейнер расположения. Несмотря на то, что визуально он никак не отображается, у него может быть длина и ширина, задаваемая в пикселах или процентах. Также есть возможность задания минимальной и максимальной величины элемента Spacer.

К примеру, Spacer, размещенный в контейнере HBox между двумя элементами, отодвинет их к правому и левому краю, «освободив» пустое место. Например, благодаря Spacer с шириной 100% визуальные компоненты окажутся у противоположных границ контейнера, как показано на рис. 8.12.

```
<mx:HBox width="250">
  <mx:Button label="Left"/>
  <mx:Spacer width="100%"/>
  <mx:Button label="Right"/>
</mx:HBox>
```

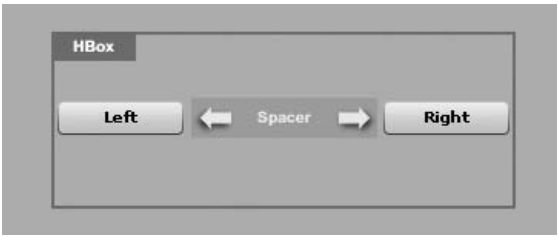

Рис. 8.12. *Использование Spacer в контейнере HBox*

Проектирование расположения элементов в приложении

Планирование не окажется лишним на любом этапе разработки программного обеспечения, в том числе и при проектировании функционального и привлекательного пользовательского интерфейса. По своему опыту могу сказать, что имеет смысл сделать набросок внешнего вида приложения, в особенности если планируется использовать множество контейнеров размещения. Это поможет разобраться во взаимном расположении компонентов.

Для вычисления оптимального расположения и размера элементов интерфейса Flex производит множество расчетов, и при

наличии большого количества вложенных контейнеров работа программы может существенно замедлиться. Поэтому так важно заранее продумать схему расположения элементов интерфейса и исключить появление лишних элементов. К примеру, не забывайте, что тег Application также является контейнером. Поэтому нижеприведенный код – не самый лучший вариант размещения двух кнопок по вертикали:

```
<mx:Application
  xmlns:mx="http://www.adobe.com/2006/mxml"
  layout="absolute">
  <mx:VBox>
    <mx:Button/>
    <mx:Button/>
  </mx:VBox>
</mx:Application>
```

Такую запись можно сократить, удалив теги контейнера Vbox и присвоив свойству `layout` тега Application значение `vertical`:

```
<mx:Application
  xmlns:mx="http://www.adobe.com/2006/mxml"
  layout="vertical">
  <mx:Button/>
  <mx:Button/>
</mx:Application>
```

Таким образом, после избавления от ненужных элементов код становится более простым, благодаря чему его стало гораздо легче воспринимать, перемещаться по нему и поддерживать, не говоря уже об увеличении производительности приложения.

HRule и VRule

HRule и VRule функционируют подобно Spacer, но они отображаются визуально в виде тонкой горизонтальной или вертикальной линии (по умолчанию). Их можно использовать, к примеру, внутри контейнера Form для разделения разных секций. Пример использования элемента HRule показан на рис. 8.13.

Примечание

В некоторых случаях элемент Label также можно использовать для изменения расположения компонентов приложения. Увеличение ширины и высоты данного элемента не влияет на размер текста, поэтому такой прием может быть использован при организации расположения элементов.

Рис. 8.13. *Использование HRule в качестве разделителя различных секций контейнера Form*

Визуальное представление структуры вашего приложения

Во Flex Bilder доступно несколько способов представления иерархии компонентов приложения и навигации по ней.

Во-первых, на панели Outline (если ее нет в вашем рабочем пространстве, ее можно открыть через меню Window→Outline) отображается древовидная структура вашего приложения. При выборе элемента списка выделяется соответствующий ей элемент на сцене и наоборот.

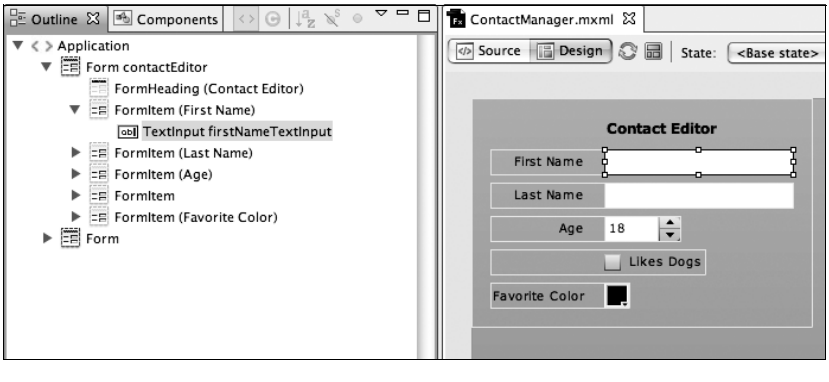

Рис. 8.14. *Панель Outline*

Во-вторых, существует опция Show Surrounding Containers. Она доступна через меню Design→Show Surrounding Containers. При этом подсвечиваются окружающие данный элемент контейнеры, что показано на рис. 8.15. Это позволяет увидеть иерархию компонентов визуально, что особенно удобно при использовании невидимых контейнеров. Эту опцию полезно использовать и при перетаскивании новых компонентов на сцену, поскольку при этом создается немного пустого пространства вокруг выделенного элемента, и становится легче позиционировать элементы, как было задумано.

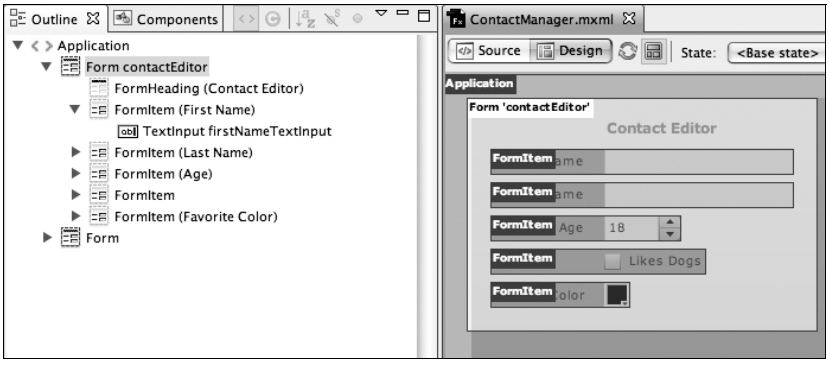

Рис. 8.15. *Пример отображения окружающих элемент контейнеров*

Совместное использование панели Outline и функции Show Surrounding Containers – мощный инструмент, помогающий понять особенности структуры ваших приложений. С его помощью можно находить и выделять контейнеры и элементы либо в списке, либо прямо на сцене.

Выравнивание

Многие контейнеры Flex, такие как HBox и VBox, предоставляют возможность горизонтального и вертикального выравнивания благодаря наличию свойств `horizontalAlign` и `verticalAlign`, которым можно присвоить одно из значений: `left`, `center` или `right`. В качестве примера возьмем контейнер VBox шириной 400 пикселов, в котором расположены три кнопки. По умолчанию свойство `horizontalAlign` имеет значение `left`, что означает расположение всех трех кнопок у левого края контейнера. Чтобы центрировать эти элементы, достаточно изменить значение данного свойства на `center`, как в следующем примере:

```
<mx:VBox
  width="400"
  height="400"
  horizontalAlign="center">

  <mx:Button label="Button"/>
  <mx:Button label="Button"/>
  <mx:Button label="Button"/>

</mx:VBox>
```

Существует еще один способ центрирования элементов или их привязки к левому или правому краю родительского элемента – использование ограничителей.

Внимание

Нет смысла в использовании свойств для выравнивания, если размер контейнера не задан, поскольку в этом случае для него будет определен размер, достаточный ровно для размещения дочерних элементов, и между его «стенками» и дочерними элементами не останется пустого пространства.

Использование ограничителей

Существует еще один эффективный способ расположения элементов вашего приложения, который можно применять совместно с контейнерами расположения. Это использование ограничителей, позволяющее не только позиционировать компоненты, но и регулировать их размер.

Основные принципы использования

Ограничители можно рассматривать как дополнение возможностей абсолютного позиционирования, поскольку они могут быть использованы с Canvas, Application, Panel и другими контейнерами в соответствии с абсолютной схемой расположения элементов. Они позволяют привязать компонент к краю контейнера так, что при изменении размера последнего он перемещается или изменяет свой размер, оставаясь привязанным к данной точке.

Чтобы лучше понять тонкости использования ограничителей, попробуем изменить интерфейс, позиционируя с помощью ограничителей элементы, изначально расположенные абсолютно.

Рассмотрим код, в котором кнопке Button, расположенной внутри контейнера Application, заданы абсолютные значения координат. В режиме Design кнопка отображается на расстоянии 5 пикселов от правого и нижнего края сцены, поскольку значения координаты x составляет 600, а y – 400 (рис. 8.16).

```
<mx:Application
  xmlns:mx="http://www.adobe.com/2006/mxml"
  layout="absolute">
```

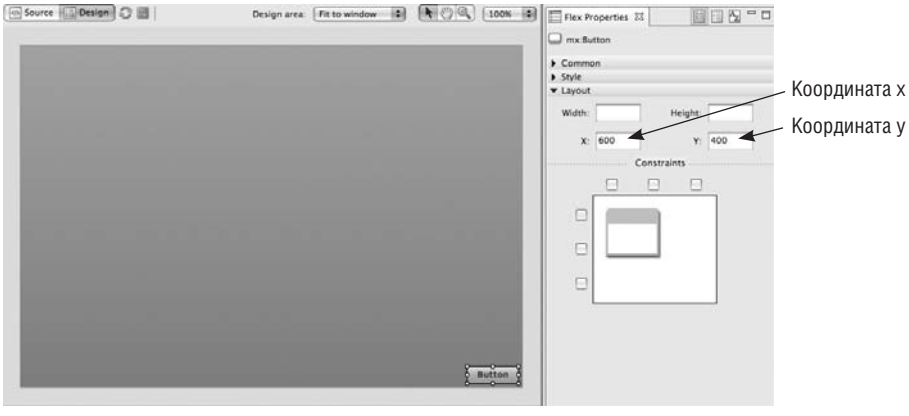

Рис. 8.16. *Раздел Layout панели Flex Properties, отображающий значения координат элемента*

```
    <mx:Button label="Button" x="600" y="400"/>
</mx:Application>
```

Раздел Layout панели Flex Properties поможет нам изменить способ расположения компонентов. Внизу раздела, после полей со значениями ширины и высоты, а также координат x и y находится инструмент для создания ограничителей. Установите правый верхний и нижний левый флажки, и Flex сгенерирует необходимый код с использованием ограничителей (рис. 8.17). При этом с помощью свойств `right` и `bottom` создаются ограничители для правого и нижнего края контейнера:

```
<mx:Application
  xmlns:mx="http://www.adobe.com/2006/mxml"
```

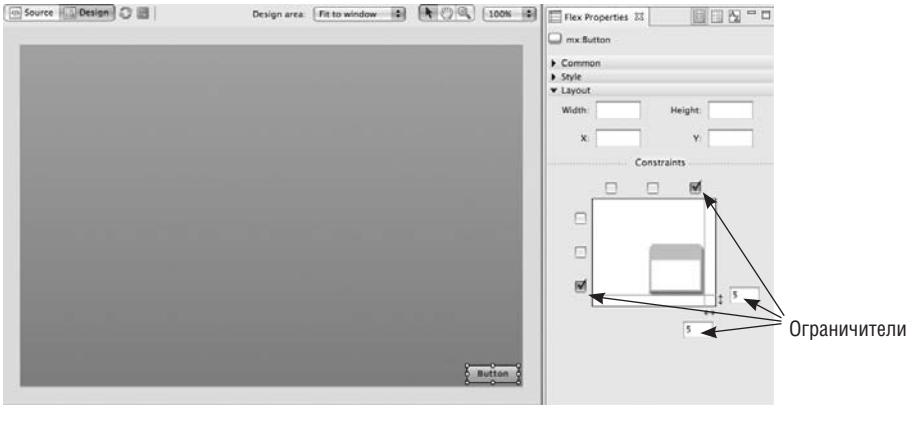

Рис. 8.17. *Раздел Layout при установке ограничителей*

```
    layout="absolute">
    <mx:Button label="Button" right="5" bottom="5" />
</mx:Application>
```

Обратите внимание, что данные атрибуты используются вместо атрибутов x и y.

Примечание

Раздел Layout в режиме Design определяет текущее расположение кнопки относительно родительского контейнера (в данном случае Application). В нашем примере кнопка расположена на расстоянии 5 пикселов от правой и нижней границ контейнера, поэтому создаются соответствующие ограничители при включении правого или нижнего флажков.

Теперь элемент Button расположен не абсолютно в соответствии со значениями координаты x, равной 600 пикселов и y, равной 400 пикселов, а привязан к точке, находящейся на расстоянии 5 пикселов от правого и нижнего краев контейнера. Внешний вид приложения при изменении его размеров показан на рис. 8.18.

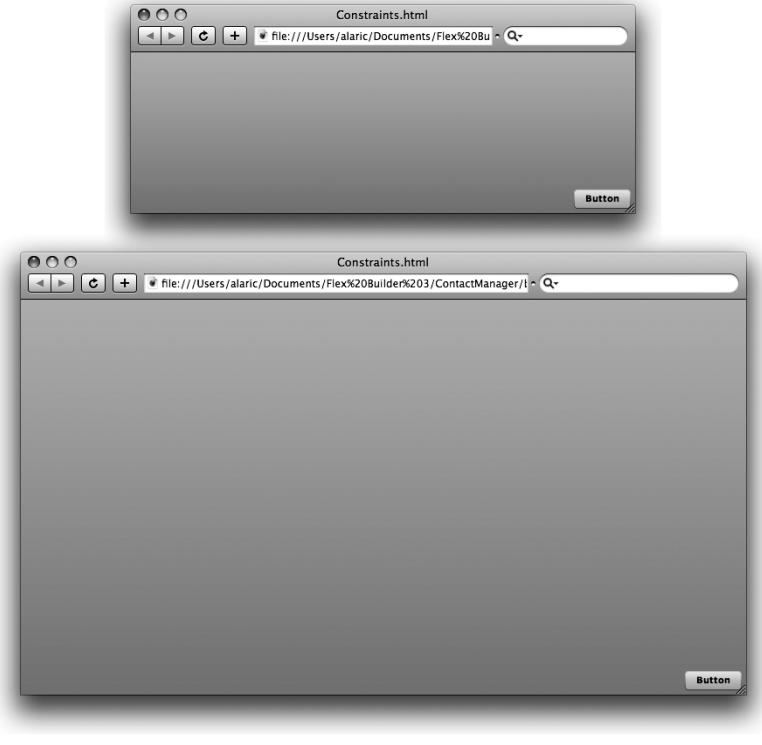

Рис. 8.18. *Кнопка привязана к правому нижнему углу приложения и остается в точке привязки при изменении его размеров*

Можно добавлять и несколько ограничителей, что может даже привести к изменениям размеров кнопки. Вновь обратимся в нашему примеру. Переключитесь в режим Design и установите размер кнопки так, чтобы она занимала контейнер Application целиком, не считая нескольких пикселов по краям (рис. 8.19). Свойства ширины и высоты имеют теперь большие значения.

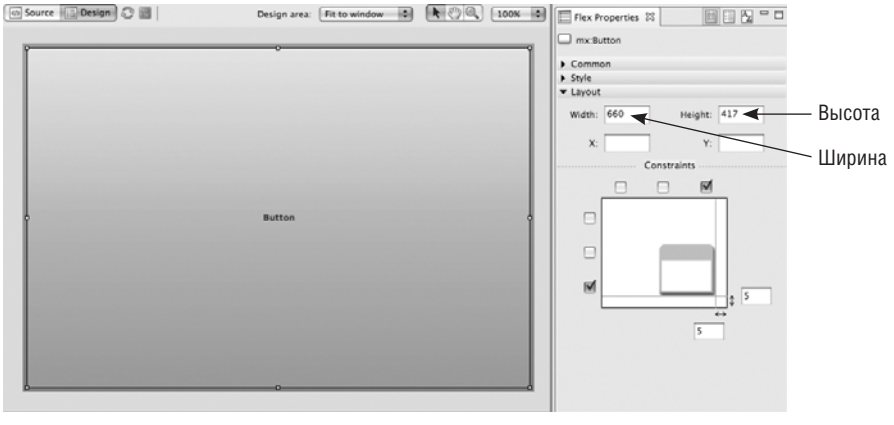

Рис. 8.19. *Размеры элемента Button совпадают с видимой частью Application (вид в режиме Design)*

Теперь добавим ограничители левого и верхнего края, включив флажки в разделе Layout. Вместо значений x и y в тег Button будут добавлены новые атрибуты (рис. 8.20). Код данного примера должен выглядеть примерно следующим образом:

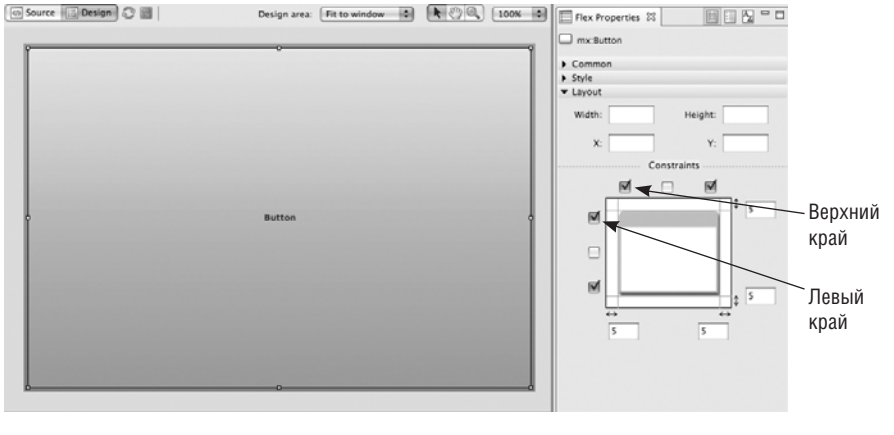

Рис. 8.20. *Размер кнопки теперь определяется значениями ограничителей*

```
<mx:Application
  xmlns:mx="http://www.adobe.com/2006/mxml"
  layout="absolute">

  <mx:Button label="Button" right="5" bottom="5" left="5" top="5"/>

</mx:Application>
```

А теперь кнопка изменяет свой размер, чтобы оставаться привязанной к углам тега Application, как показано на рис. 8.21.

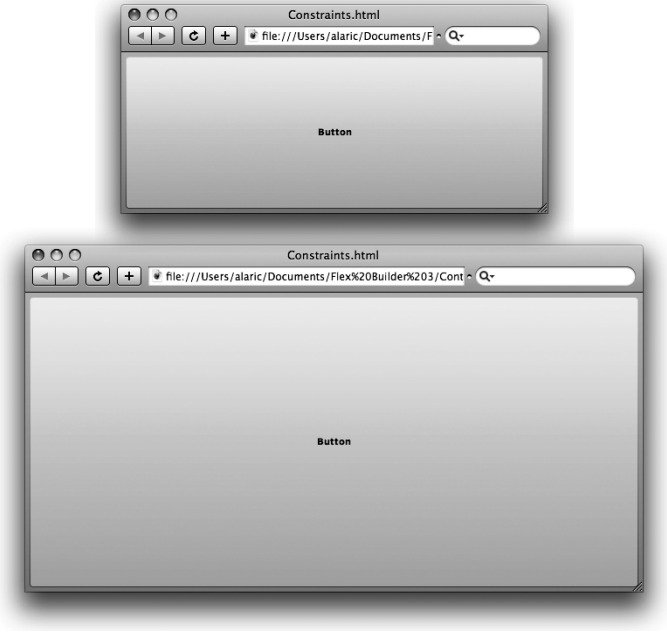

Рис. 8.21. *Элемент Button привязан к углам приложения и будет уменьшаться или увеличиваться соответствующим образом при изменении его размеров*

Ограничители также используются для центрирования элемента как по вертикали, так и по горизонтали. Для демонстрации этой возможности создадим небольшой пример. Поместите в любое место на сцене панель Panel в режиме Design. Вернитесь в раздел Layout панели Properties и на этот раз установите флажки, расположенные в центре слева и наверху. При этом координаты x и y будут заменены на центрирующие ограничители horizontalCenter и verticalCenter. Число, отображаемое в текстовых полях рядом с флажками, указывает на расстояние Panel от центра приложения по горизонтали и вертикали соответственно, измеряемое в пикселах. Панель можно расположить в самом центре приложения, установив нулевое значение обоих полей, как на рис. 8.22. При этом код приложения выглядит так:

```
<mx:Application
  xmlns:mx="http://www.adobe.com/2006/mxml"
  layout="absolute">

  <mx:Panel
    width="250"
    height="200"
    layout="absolute"
    horizontalCenter="0"
    verticalCenter="0">
  </mx:Panel>
</mx:Application>
```

Теперь вне зависимости от изменения размеров панели или самого приложения она будет расположена по центру.

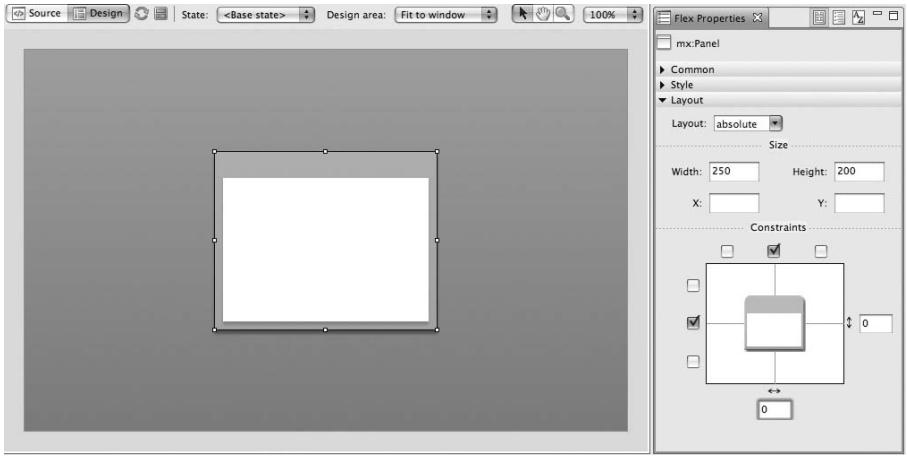

Рис. 8.22. *Элемент Panel, расположенный по центру с помощью ограничителей*

Примечание

Использование ограничителей `left` и `right` совместно с `horizontalCenter` может привести к непредсказуемым результатам. Похожим образом лучше избегать использования `top` и `bottom` вместе с `verticalCenter`. Их использование не исключает друг друга, но чаще всего не имеет смысла, к примеру, привязка элемента к левому краю контейнера и одновременное центрирование по горизонтали. Если в вашем коде используются как обычные, так и центрирующие ограничители, первые обычно определяют размер элемента, а вторые – его расположение. При работе в режиме Design Flex Builder поможет вам избежать ошибок, автоматически удаляя несовместимые ограничители.

Связывание данных при расположении элементов

Для создания гибкой схемы расположения элементов приложения возможно использовать связывание данных в различных атрибутах, определяющих размер и позиционирование элементов. Значения свойства `width` двух разных панелей можно связать так же, как ранее мы связывали свойства `text` элементов TextInput. При этом ширина обеих панелей всегда остается одинаковой.

Похожим образом можно привязать `width` панели к ее `height` – в результате она примет форму квадрата. Связывать можно также свойства `x`, `y` и ограничители. К примеру, при абсолютном позиционировании элементов при привязке свойства `x` одной из панелей к свойству `x` и `width` другой, их расположение станет относительным:

```
<mx:Panel id="onePanel"
  x="{anotherPanel.width + anotherPanel.x + 10}"
  width="{anotherPanel.width}"
  height="{anotherPanel.height}" />

<mx:Panel id="anotherPanel"
  width="250"
  height="200" />
```

Связывание данных можно осуществлять и для вычисления координат и размеров элементов. В этом смысле перед вами открываются практически безграничные возможности.

Единственный недостаток такого приема состоит в том, что связывание не отображается в режиме Design.

Использование ограничителей рядов и колонок

Flex предлагает расширенные возможности использования ограничителей с помощью невидимой решетки, состоящей из рядов и колонок, к которым можно привязывать компоненты. Благодаря им появляется возможность привязки к любой точке, а не только к краю контейнера.

Колонка создается при помощи тега `<mx:constraintColumn/>` и располагается верхнем левом углу приложения. Затем нужно задать в нем атрибут `width`. Следующая колонка располагается справа от только что созданной.

Ряды, создаваемые тегом `<mx:constraintRow/>`, располагаются по направлению сверху вниз. Первый созданный ряд находится в самом верху приложения. Необходимо определить размер занимаемого им

пространства, установив значение его атрибута height. Следующий ряд появляется прямо под предыдущим.

Примечание

Ряды и колонки появляются в приложении в соответствии с их порядком в MXML-коде. Высота ряда (height) определяет расположение ряда, следующего за ним. Аналогичным образом определяется позиция колонки – в зависимости от ширины предыдущей (width). Использование в данном случае координат x и y бессмысленно. Для определения положения следующей колонки или ряда всегда используйте свойства height и width.

Для привязки элементов к рядам используется практически такой же синтаксис, как и при обычной привязке. На то, что компонент привязывается к определенному ряду или колонке, указывает *предшествующий* значению расстояния в пикселах идентификатор колонки или ряда. К примеру, при обычной привязке компонента без использования рядов или колонок, чтобы разместить кнопку на расстоянии 5 пикселов от верхнего края содержащего его контейнера, можно использовать код <mx:Button top="5"/>. Для привязки элемента на расстоянии 5 пикселов от ряда с именем row2, необходимо перед числовым значением указать данное имя, за которым следует двоеточие: <mx:Button top="row2:5"/> (рис. 8.23).

Рис. 8.23. *Синтаксическая конструкция, используемая для привязки элемента к ряду или колонке*

Можно создавать множество рядов или колонок, привязывая элементы к любому из них. При этом возможно создание похожей на таблицу схемы расположения, позволяющей объединение рядов и колонок.

Посмотрите на следующий код, в котором несколько элементов HRule и VRule привязаны к рядам и колонкам. Поскольку ряды и колонки невидимы, привязанные к ним линии помогут вам увидеть их расположение визуально. Затем в сетке располагаются четыре кнопки, привязанные к рядам и колонкам (рис. 8.24).

```
<mx:Application
  xmlns:mx="http://www.adobe.com/2006/mxml"
  layout="absolute ">

  <!-- Создание двух колонок, первая из них занимает пространство
       в 200 пикселов по горизонтали, начиная от левого края контейнера.
       Вторая колонка начинается на расстоянии 200 пикселов от левого
```

```
        края контейнера и занимает все оставшееся пространство
        до правого края контейнера
    -->
    <mx:constraintColumns>
      <mx:ConstraintColumn id="col1" width="200" />
      <mx:ConstraintColumn id="col2" width="100%" />
    </mx:constraintColumns>

    <!-- Создание двух рядов, первый из которых занимает 100 пикселов вниз
        от верхнего края контейнера, второй начинается сразу под первым
        и занимает оставшееся пространство до нижнего края контейнера
    -->
    <mx:constraintRows>
      <mx:ConstraintRow id="row1" height="100"/>
      <mx:ConstraintRow id="row2" height="100%"/>
    </mx:constraintRows>

    <!-- HRule и VRule, используемые для визуального отображения
        местонахождения решетки рядов и колонок
    -->
    <mx:VRule height="100%" left="col1:0" />
    <mx:VRule height="100%" left="col2:0" />

    <mx:HRule width="100%" top="row1:0" />
    <mx:HRule width="100%" top="row2:0" />

    <!-- Кнопки, расположенные в сетке -->
    <mx:Button
      label="column 1, row 1"
      left="col1:5"
      right="col1:5"
      top="row1:5"
      bottom="row1:5"/>

    <mx:Button
      label="column 1, row 2"
      left="col1:5"
      right="col1:5"
      top="row2:5"
      bottom="row2:5"/>

    <mx:Button
      label="column 2, row 1"
      left="col2:5"
      right="col2:5"
      top="row1:5"
      bottom="row1:5"/>

    <mx:Button
      label="column 2, row 2"
      left="col2:5"
      right="col2:5"
      top="row2:5"
      bottom="row2:5"/>
</mx:Application>
```

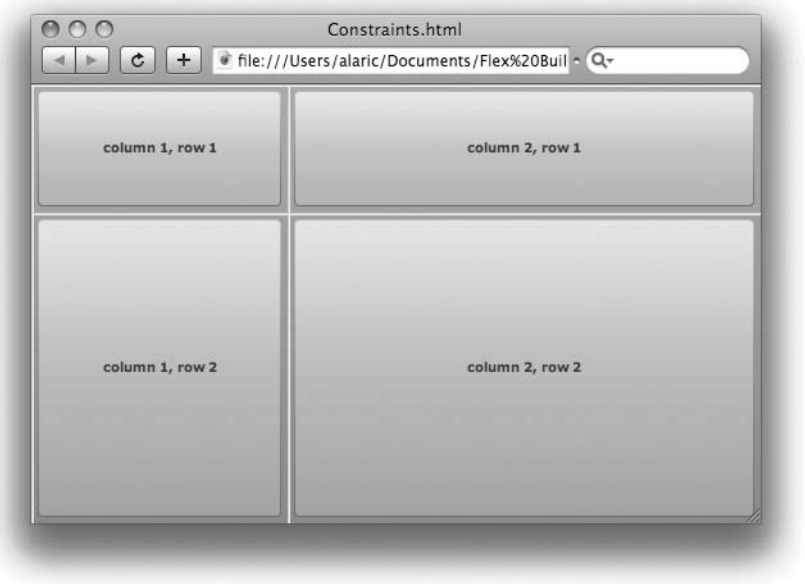

Рис. 8.24. *Четыре кнопки, расположенные в решетке, с привязкой к рядам и колонкам*

Примечание

В ссылках на ряды или колонки нельзя использовать фигурные скобки, т. е. следующая запись неверна: `<mx:Button top="{row1:0}"/>`.

Заключение

В этой главе вы узнали об особенностях построения схемы расположения элементов Flex-приложения. Вы имеете представление о возможных вариантах размещения элементов и списке отображения, благодаря которому представляется возможность перемещения компонентов, в том числе за пределы контейнера или внутрь другого контейнера. Интерфейс вашего приложения обладает еще большей гибкостью, поскольку вы умеете определять размеры компонентов как в пикселах, так и в процентах. Ваше приложение ContactManager теперь выглядит вполне солидно – благодаря использованию изученных в данной главе контейнеров. Вы также узнали о сложных приемах расположения элементов с помощью ограничителей. Возможно, данная глава показалась вам слишком объемной, но полученные в ней знания вы сразу можете применить на практике. В следующей главе мы подробнее поговорим о контейнере Form и создании с его помощью интерактивных, функциональных и привлекательных форм.

9

В этой главе:
- Подготовка приложения
- Возможности проверки данных
- Ограничение ввода данных
- Форматирование отображаемых данных

Формы «со всеми удобствами»

Случалось ли вам заполнять HTML-форму на веб-странице, отправлять ее и ждать затем результата – только для того, чтобы узнать, что некоторые из полей были заполнены неверно или не были заполнены вовсе? Легко ли было найти и исправить свою ошибку? Было ли это что-то нелепое, вроде того, что вы не заключили в скобки код страны в номере телефона (или, напротив, сделали это, когда это не требовалось)? Думаю, было бы лучше, если бы такие ситуации никогда не возникали.

Flex предлагает простой и естественный способ валидации и форматирования данных. Возможность проверки правильности введенных данных встроена в большинство элементов Flex. Пользователи вашего приложения без труда заметят свою ошибку и с легкостью внесут необходимые исправления. Благодаря прекрасным инструментам проверки и форматирования данных интерфейс вашего приложения станет максимально дружественным для пользователя.

Подготовка приложения

Прежде всего вернемся к проекту ContactManager, с которым мы работали в предыдущей главе, и займемся контейнером Form. Мы добавим несколько полей и внесем некоторые изменения, чтобы приложение стало похожим на настоящую адресную книгу. Пользователь сможет ввести номер телефона, e-mail, адрес и почтовый индекс, а с помощью нового элемента CheckBox можно будет определить, заполнены данные для физического лица или компании. Мы также уберем элемент NumericStepper, поскольку гораздо логичнее запросить данные о дате рождения (а рассчитывать возраст пользователя мы научимся чуть

позже в этой же главе), и научимся использовать элемент DateField для введения дат с помощью всплывающего календаря.

Для начала откройте приложение ContactManager, добавьте в форму Contact Editor Form поле DateField и измените значения его свойства `editable` на `true`. Это даст пользователю дополнительную возможность ввода даты с клавиатуры в дополнение к выбору мышью из всплывающего календаря.

Теперь добавьте элемент TextInput, предназначенный для ввода телефонного номера. Мы также предоставим пользователю возможность указать, является ли телефон мобильным, домашним или каким-либо иным. Поскольку в такой ситуации может быть выбран лишь один пункт из трех, использование элемента CheckBox нам не подойдет, ведь флажок может быть либо установлен, либо нет, т. е. элемент принимает одно из значений `true` или `false`. Для нашей цели идеально подойдет использование группы переключателей, что позволит пользователю выбрать один из предлагаемых вариантов.

Примечание

Не забывайте о режиме Design – с его помощью весьма просто заключить создаваемые вами поля формы в контейнеры FormItem. Просто перетащите необходимый элемент в нужное место, а Flex Builder создаст обрамляющий его контейнер автоматически.

Итак, переключитесь в режим Design и перетащите элемент RadioButtonGroup в контейнер FormItem, предназначенный для номера телефона. При этом откроется диалоговое окно, изображенное на рис. 9.1, в котором можно задать некоторые настройки создаваемый группы переключателей. Назовите ее `phoneRadioButtonGroup` и создайте три переключателя, соответствующие трем вариантам выбора – mobile (мобильный), home (домашний) и other (другое). Новый переключатель можно добавить с помощью кнопки Add. Как только вы подтвердите свой выбор, щелкнув по кнопке OK, Flex Builder автоматически создаст тег `<mx:RadioButtonGroup/>` и три тега `<mx:RadioButton/>` в коде приложения:

```
<mx:RadioButtonGroup id="phoneRadioButtonGroup" />

<mx:RadioButton label="mobile" groupName="phoneRadioButtonGroup"/>
<mx:RadioButton label="home" groupName="phoneRadioButtonGroup"/>
<mx:RadioButton label="other" groupName="phoneRadioButtonGroup"/>
```

Тег `<mx:RadioButtonGroup/>` не является контейнером для элементов RadioButton. Это невидимый элемент, к которому привязаны элементы RadioButton. Он позволяет получить доступ к их состоянию, например, определяет, который из них выбран. Связь между переключателями и данной группой осуществляется благодаря свойству `groupName`, значение которого соответствует имени группы.

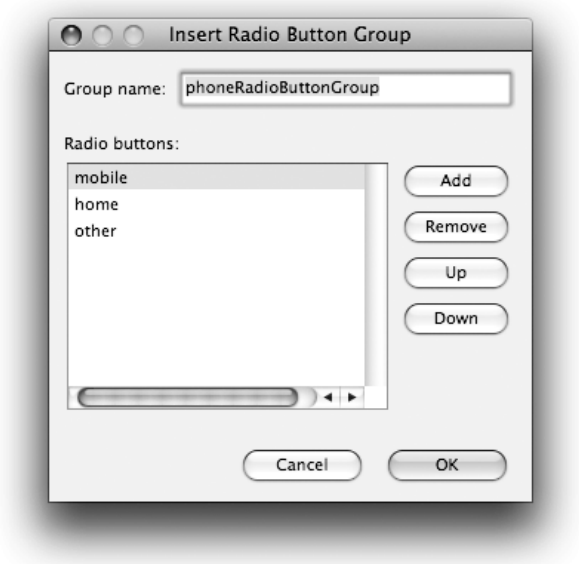

Рис. 9.1. *Диалоговое окно Insert Radio Button Group*

В режиме Design или Source (как вам удобнее) измените значение свойства `selected` первого переключателя (mobile) на `true`, так что он будет выбран по умолчанию.

Скорее всего будет необходимо проверить, ввел ли пользователь по крайней мере имя и адрес электронной почты. Для этого прежде всего нужно добавить соответствующим контейнерам FormItem свойство `required` со значением `true`. По сути дела это ничего не дает, разве что рядом с полем появится красная звездочка. Однако очень скоро вы сможете проверить, были ли введены какие-либо данные, с помощью валидатора.

Код формы Contact Editor Form должен выглядеть примерно так:

```
<mx:Form id="contactEditor">
  <mx:FormHeading
label="Contact Editor"/>
  <mx:FormItem
    label="First Name"
    required="true">
    <mx:TextInput id="firstNameTextInput"/>
  </mx:FormItem>
  <mx:FormItem
    label="Last Name">
    <mx:TextInput id="lastNameTextInput"/>
  </mx:FormItem>
  <mx:HRule
    height="22"
    width="100%"/>
```

```
    <mx:FormItem
      label="Email"
      required="true">
      <mx:TextInput id="emailTextInput"/>
    </mx:FormItem>
    <mx:FormItem
      label="Phone">
      <mx:TextInput id="phoneTextInput"/>

      <mx:RadioButtonGroup id="phoneRadioButtonGroup" />

      <mx:RadioButton
        label="mobile"
        groupName="phoneRadioButtonGroup"
        selected="true"/>
      <mx:RadioButton
        label="home"
        groupName="phoneRadioButtonGroup"/>
      <mx:RadioButton
        label="other"
        groupName="phoneRadioButtonGroup"/>
    </mx:FormItem>
    <mx:FormItem
      label="Address">
      <mx:TextArea id="addressTextArea"/>
    </mx:FormItem>
    <mx:FormItem label="Zip">
      <mx:TextInput id="zipCodeTextInput"/>
    </mx:FormItem>
    <mx:HRule
      height="22"
      width="100%"/>
    <mx:FormItem
      label="Birthday">
      <mx:DateField id="birthdayDateField"
        editable="true"/>
    </mx:FormItem>
    <mx:FormItem
      label="Favorite Color">
      <mx:ColorPicker id="favoriteColorPicker"/>
    </mx:FormItem>
    <mx:FormItem>
      <mx:CheckBox id="companyCheckBox"
        label="Company" />
    </mx:FormItem>
  </mx:Form>
```

Теперь полностью удалите раздел Contact Details: после только что внесенных изменений ссылки на элементы будут недействительны, поскольку последние уже не существуют. (Чуть позже мы создадим новую улучшенную версию этого раздела.) Затем запустите приложение – можете полюбоваться результатами проделанной работы (рис. 9.2).

Рис. 9.2. *Вид улучшенной формы Contact Editor*

Примечание

Обратитесь к статье об элементе DateField в документации Flex Language Refernce за более подробной информацией о возможных значениях свойства `formatString`.

Попробуйте ввести дату в поле DateField. Это можно сделать, выбрав число во всплывающем календаре или просто введя ее с клавиатуры. Данный элемент обладает встроенным инструментом анализа и проверки данных, что позволяет использовать различные форматы даты. К примеру, пользователь, родившийся 20 июля 1979 года может ввести `7/20/1979`, `7-20-79` или даже `7 20 1979`. Как вам такая возможность? Все должно прекрасно работать, если вы вводите сначала месяц, а потом число. И даже этот порядок можно изменить. Свойство `formatString` позволяет установить формат даты, для чего необходимо присвоить ему значение типа String. К примеру, если вам хочется, чтобы год предшествовал месяцу и числу, присвойте элементу свойство `formatString="YYYY-MM-DD"`. Это изменит не только способ отображения даты, но и ожидаемый порядок их ввода.

Примечание

Вы уже задавали себе вопрос, почему нет специальных элементов для ввода почтового индекса или, скажем, адреса электронной почты, раз существует осо-

бый элемент DateField? Вполне возможно, существуют подобные компоненты, созданные сторонними разработчиками; их можно скачать и использовать в своих приложениях, но и на данный момент в вашем распоряжении имеется полноценный набор инструментов для выполнения самых разных задач. Элементы TextInput прекрасно подходят для ввода различного типа данных и их проверки. Использование сторонних компонентов мы обсудим в следующей главе.

Несмотря на то, что визуально элемент DateField похож на поле для ввода обычного текста, он обладает свойством `selectedDate`, являющимся объектом типа Date. Это комплексный тип данных, используемый для хранения даты и времени. При вводе даты, проверенной валидатором элемента DateField, или при ее выборе с помощью всплывающего календаря значением данного свойства становится объект Date.

Элемент DateField – это отличный пример возможности ввода данных без необходимости задумываться об их формате. Теперь вы имеете представление о данном элементе, и настало время познакомиться со способами проверки введенных данных.

Возможности проверки данных

Увидев фантастические возможности элемента DateField, вы наверняка с ужасом вспоминаете встречавшиеся вам ранее формы на веб-страницах, где такая проверка данных отсутствовала. С помощью Flex можно с легкостью создавать удобные и привлекательные формы; для этого достаточно овладеть базовыми инструментами проверки данных.

Использование инструментов валидации

Чтобы научиться приемам проверки данных прямо на практике, мы установим проверку для каждого поля приложения Contact Editor. Начнем с первого поля, предназначенного для ввода имени, которое должно быть заполнено в обязательном порядке, что мы указали в атрибуте `required` его контейнера FormItem. Однако добавление этого атрибута лишь указывает на то, что поле требует заполнения, но не проверяет введенные данные на наличие ошибок. Для этого мы будем использовать невидимый компонент StringValidator.

StringValidator

StringValidator – основной инструмент проверки данных, подтверждающий наличие введенного текста. Для его использования необходимо разместить тег `<mx:StringValidator/>` в верхней части MXML-кода, не выходя за границы тега `<mx:Application/>`. Такое расположение необязательно, но традиционно принято размещать невизуальные компоненты именно таким образом; это облегчает их поиск и изменение. Тегу `<mx:StringValidator/>` можно присвоить атрибут `source`, которые ссылается на элемент, за которым должен осуществляться контроль.

Атрибут property указывает на свойство данного элемента, подлежащее проверке. Например, чтобы установить проверку для поля firstNameTextInput, нужно использовать данный тег следующим образом:

```
<mx:StringValidator id="firstNameValidator"
  source="{firstNameTextInput}"
  property="text"/>
```

Внимание

Свойство source тега StringValidator (равно как и других инструментов проверки данных) предназначается для осуществления привязки к элементу в целом, а не к его свойству, на что следует обратить внимание. Таким образом, следующая запись неверна: `<mx:StringValidator source="{firstNameTextInput.text}" />`.

Теперь запустите приложение и посмотрите, как все это работает. Если вы попытаетесь перейти к следующему полю формы (с помощью клавиши Tab или указателя мыши), не введя никаких данных, вокруг поля firstNameTextInput появится предупреждающая красная рамка. При наведении курсора мыши на данное поле появится ненавязчивое всплывающее сообщение, указывающее на характер ошибки. По умолчанию выводится текст «This field is required» («Поле обязательно для заполнения»).

Но вы использовали далеко не все возможности данного инструмента. С помощью свойства requiredFieldError можно изменить текст сообщения об ошибке. Можно выразиться чуть конкретнее и, что уж говорить, пожестче, присвоив следующее значение данному свойству: `requiredFieldError="I'm sure you've got a name.&#13;Why not enter it here?"`. («Не сомневаюсь, имя у вас есть. Что же вам мешает ввести его?».) Если наличие символов
 навело вас на мысль, что я решил вставить крепкое словцо, вынужден вас разочаровать. Это кодировка, используемая для вставки в XML специальных символов. Использование этого символа осуществляет перенос последующего текста на следующую строку, так что теперь сообщение об ошибке выглядит куда лучше.

Обязательно запустите приложение и посмотрите, как работает такая технология проверки данных. Примерный вид вашего приложения представлен на рис. 9.3.

Совет

В ActionScript для переноса текста на следующую строку используется последовательность символов \n. Таким образом, код будет выглядеть примерно так:

```
stringValidator.requiredFieldError = "I'm sure you've got a name.\n Why
not enter it here?";
```

Теперь добавим возможность проверки данных для следующего поля. Мы решили, что в данном приложении пользователю не обязательно вводить фамилию. В таком случае валидация и не нужна, не правда ли?

Рис. 9.3. *Дружелюбная форма*

Не совсем так. Поле не требует обязательного заполнения, но если данные тем не менее введены, необходимо осуществить их проверку.

Это можно сделать с помощью еще одного тега StringValidator. На этот раз присвоим его свойству `required` значение `false` (по умолчанию используется значение `true`). Таким образом, поле необязательно для заполнения, и несмотря на его связь с тегом валидатора, никакая проверка данных не осуществляется. При желании можно использовать простую проверку длины введенного текста с помощью свойства `minLength` (например, необходимо удостовериться, что введено больше двух символов). Это поможет предотвратить ввод пользователем инициала, поскольку вам нужна полная фамилия.

Внимание

Данный пример всего лишь демонстрирует возможности инструментов валидации. В реальности возможны случаи, когда имя пользователя действительно состоит из одного знака, и такой способ проверки не подойдет. При разработке способов проверки данных тщательно продумайте все возможные случаи.

Вы также можете изменить текст сообщения об ошибке, как и в предыдущем случае. Однако для этого следует использовать свойство `tooShortError`, а не `requiredFieldError`, поскольку в данном случае оповещение об ошибке происходит при вводе слишком короткого текста.

```
<mx:StringValidator id="lastNameValidator"
  source="{lastNameTextInput}"
  property="text"
  required="false"
  minLength="2"
  tooShortError="What kind of last name is that?[1]"/>
```

Запустите приложение, и вы увидите, как работает такой способ проверки данных. Внешний вид вашего приложения показан на рис. 9.4.

Рис. 9.4. *Чересчур требовательное приложение*

[1] «Это что еще за фамилия такая?» (англ.). – *Прим. перев.*

EmailValidator

Следующим пунктом в списке является поле для ввода адреса электронной почты. Для проверки правильности введенных данных удобно использовать компонент EmailValidator. Принцип его действия аналогичен StringValidator, так что просто добавьте данный тег в код вашего приложения и свяжите его с элементом `emailTextInput`. Чтобы разнообразить приложение, можете придумать собственный текст сообщения об ошибке, отображаемого с помощью свойства `requiredFieldError`. Однако данный компонент имеет множество подготовленных сообщений об ошибках и возможностей их настройки – для этого существуют свойства `missingAtSignError`, `invalidDomainError` и т. п. Совсем не обязательно изобретать свой собственный текст для каждого из этих случаев, – значения по умолчанию прекрасно подойдут в любой ситуации. Однако следует знать о наличии такой возможности.

```
<mx:EmailValidator id="emailValidator"
  source="{emailTextInput}"
  property="text"
  requiredFieldError="An at sign (@) is missing in your e-mail address.[1]"/>
```

Теперь при отсутствии введенного e-mail адреса ваше приложение выдаст дурацкое сообщение об ошибке. Если вы не станете изменять тексты сообщений, используемые по умолчанию, при вводе неправильного адреса электронной почты пользователь получит подробное оповещение о том, что именно следует исправить.

Запустите и протестируйте приложение. Его внешний вид показан на рис. 9.5.

Примечание

Естественно, проверка данных с помощью EmailValidator или PhoneNumberValidator не может гарантировать, что указанный адрес электронной почты или номер телефона действительно существуют – она лишь обеспечивает ввод данных в надлежащем формате. Чтобы удостовериться в реальном существовании введенного адреса или номера, необходимо подключение к базе данных или веб-сервису, предоставляющему соответствующие услуги. Об этом мы поговорим в следующей главе.

PhoneNumberValidator

Следующий пункт на повестке дня – проверка телефонного номера. Вы, наверное, уже догадались, что на этот раз мы будем использовать инструмент PhoneNumberValidator. Данное поле также не требует обязательного заполнения, но в том случае, когда данные введены, необходимо осуществить их проверку.

[1] «В вашем почтовом адресе не хватает знака собаки (@).» (англ.). – *Прим. перев.*

***Рис. 9.5.** Весьма конкретизированные сообщения об ошибке, выдаваемые EmailValidator*

```
<mx:PhoneNumberValidator id="phoneValidator"
  source="{phoneTextInput}"
  property="text"
  required="false" />
```

Основной особенностью данного компонента является (по умолчанию) принятие многочисленных возможных форматов номера телефона, т. е. пользователь может вводить `(415)555-8273`, `415-5558273` или даже `415 555 8273`. Однако введенная фраза `I don't have a phone` («У меня нет телефона») не пройдет проверку (рис. 9.6).

ZipCodeValidator

Поле для ввода адреса проживания не требует валидации, хотя при желании введенные данные можно проверить на предмет достаточной длины текста, как и в случае с фамилией. Итак, перейдем к почтовому индексу. Здесь нам на помощь придет ZipCodeValidator.

Рис. 9.6. *Мимо PhoneNumberValidator просто так не пройти*

Он предназначен для проверки почтового индекса в пределах США (по умолчанию), или в пределах как США, так и Канады. Эта настройка изменяется присваиванием его свойству `domain` одного из двух возможных значений типа String – «US Only» или «US or Canada». Давайте дадим пользователю возможность вводить и канадские индексы.

Примечание

Для большинства приложений с архитектурой «клиент-сервер» простой проверки данных на стороне клиента будет недостаточно. Как правило, такие приложения осуществляют валидацию данных на сервере в соответствии с определенной схемой во избежание сохранения неверной информации на сервере. Но на данный момент вам не стоит об этом задумываться – эту тему я оставлю для другой книги.

Это как раз тот случай, когда уместно вспомнить о константах. Это понятие было рассмотрено в главе 6 в разделе «Константы». Строку «US or Canada» можно подзабыть или допустить в ней опечатку, что приве-

дет к неправильному функционированию ZipCodeValidator. Вместо этого можно использовать константу, принадлежащую классу `mx.validators.ZipCodeValidatorDomainType`. С помощью технологии связывания данных можно присвоить в качестве значения свойства `domain` константу `ZipCodeValidatorDomainType.US_OR_CANADA`. Это позволит избежать опечаток, поскольку компилятор проверит данное значение. Благодаря функции автозаполнения ввести такое значение не составит труда.

Примечание

При использовании функции автозаполнения при вводе константы в Flex Builder импорт класса осуществляется автоматически.

Для получения доступа к классу его необходимо импортировать. Таким образом, код валидации будет состоять из двух частей. Вначале мы используем тег `<mx:Script/>` для импорта класса с помощью выражения `import`.

```
<mx:Script>
  <![CDATA[
    import mx.validators.ZipCodeValidatorDomainType;
  ]]>
</mx:Script>
```

Далее следует код самого валидатора:

```
<mx:ZipCodeValidator id="zipCodeValidator"
  source="{zipCodeTextInput}"
  property="text"
  domain="{ZipCodeValidatorDomainType.US_OR_CANADA}"
  required="false"/>
```

Теперь в вашем приложении осуществляется проверка почтовых индексов как США, так и Канады (рис. 9.7).

Другие валидаторы

Благодаря широким возможностям элемента DateField нет необходимости валидации вводимых данных. Однако при желании с этой целью можно использовать класс DateValidator. Кроме того, существует масса других валидаторов, таких как CreditCardValidator, CurrencyValidator, NumberValidator и SocialSecurityValidator. Теперь вы знаете основные принципы использования инструментов валидации, и разобраться с ними самостоятельно не составит никакого труда.

Примечание

Существует даже инструмент валидации RegExpValidator, работающий с регулярными выражениями. Он позволяет сопоставлять текст с шаблонами. С его помощью можно создать сложную систему проверки данных, вводимых в поле с адресом проживания, – можно удостовериться, что данные введены в соответствующем формате –название города, запятая, название штата.

Рис. 9.7. *ZipCodeValidator понимает и канадцев*

Ошибка без ошибки

Вы умеете использовать валидаторы для отображения сообщений об ошибке. Это прекрасный способ уведомления пользователя. Но знаете ли вы, что существует прекрасный способ вывода на экран сообщений об ошибке без использования валидаторов? Появления красной рамки и сообщения можно достичь с помощью свойства `errorString`.

Следующий код содержит пример использования данного свойства с элементом ColorPicker:

```
<mx:ColorPicker id="favoriteColorPicker"
errorString="Hmm. Not quite the color I was expecting...1"/>
```

[1] «Хм. Это не совсем тот цвет, которого я ожидал.» (англ.). – *Прим. перев.*

Это свойство удобно использовать в сценариях, что позволяет отображать сообщение в зависимости от каких-то событий или параметров. Чтобы убрать сообщение, достаточно присвоить значению свойства `errorString` пустую строку, как в следующем примере на ActionScript:

```
favoriteColorPicker.errorString = "";
```

Рис. 9.8. *Использование свойства errorString*

Основные способы валидации

Вы, конечно, заметили, что валидаторы выдают сообщение об ошибке только в том случае, когда некоторые данные введены (или не введены) и фокус смещается на следующее поле. Это происходит благодаря *триггерам*, т. е. событиям, вызывающим запуск процесса валидации.

Любой компонент валидации обладает свойством `trigger`, ссылающимся на элемент, за которым осуществляется контроль. По умолчанию

его значение совпадает со значением свойства source. Также можно использовать свойство triggerEvent, присваивая ему в качестве значения имя события, после которого должна быть осуществлена проверка данных. По умолчанию таким событием является valueCommit; оно происходит при окончании ввода данных, что как правило определяется смещением фокуса на другой элемент, но данное событие может быть вызвано и при изменении значения элемента из сценария.

К примеру, для запуска проверки данных в поле для ввода адреса электронной почты (emailTextInput), как только пользователь начнет ввод текста, нужно использовать свойство triggerEvent со значением change, т. е. событием, происходящим при внесении изменений в поле TextInput. (При этом значение свойства trigger валидатора оставлено по умолчанию и совпадает со значением его свойства source (в данном случае – emailTextInput).

```
<mx:EmailValidator id="emailValidator"
  source="{emailTextInput}"
  property="text"
  requiredFieldError="Please enter your email.&#13; I promise not
  to send spam.1"
  triggerEvent="change" />
```

Данный пример наглядно демонстрирует особенности функционирования триггеров, однако не стоит использовать такой код в реальном приложении, поскольку в течение всего процесса ввода данных пользователю будет выдаваться сообщение об ошибке до тех пор, пока адрес электронной почты не будет введен целиком. Однако триггеры можно применять по-другому. Добавьте в ваше приложение элемент Button, который будет выполнять функцию кнопки, подтверждающей ввод или отправку данных. (На самом деле пока мы не будем отправлять никакие данные, зато научимся создавать интерфейс, который будет обладать такой возможностью).

При добавлении элемента Button в форму с помощью режима редактирования Design он автоматически будет помещен в соответствующий контейнер FormItem, предназначенный для присвоения элементу метки и выравнивания полей формы. Что касается выравнивания, здесь FormItem весьма кстати, но в присваивании элементу Button метки нет никакого смысла, поскольку она у него уже имеется. Поэтому в качестве значения свойства label контейнера оставьте пустую строку. Затем задайте кнопке идентификатор (id) submitButton.

Теперь самое время указать валидатору EmailValidator на созданную кнопку с помощью его свойства trigger. Поскольку мы решили, что проверка данных будет запущена при нажатии на кнопку, присвойте свойству triggerEvent значение click:

---

[1] «Пожалуйста, введите ваш e-mail. Обещаю не присылать спам» (англ.). – *Прим. перев.*

```
<mx:EmailValidator id="emailValidator"
  source="{emailTextInput}"
  property="text"
  requiredFieldError="Please enter your email.&#13; I promise not
  to send spam."
  trigger="{submitButton}"
  triggerEvent="click" />
```

Теперь при нажатии пользователем на кнопку submitButton будет запущен процесс валидации данных, введенных в поле emailTextInput. Таким образом мы изменили поведение валидатора, используемое по умолчанию, и проверка данных при смене фокуса осуществляться не будет – это произойдет только при нажатии на кнопку.

Для более полноценного контроля за осуществлением проверки данных можно применять метод validate() собственно объекта валидатора, который позволяет запустить данный процесс в любое время. Чтобы увидеть данный метод в действии, для начала нужно убрать только что внесенные изменения значений свойств trigger и triggerEvent в вашем коде. Затем создадим функцию, вызывающую метод validate() объекта EmailValidator.

```
private function validateAndSubmit():void
{
  emailValidator.validate();
}
```

После того как для элемента submitButton будет установлен обработчик события click, ссылающийся на данную функцию, нажатие на кнопку послужит триггером валидации данных, введенных в поле для адреса электронной почты. Благодаря такому приему проверка данных будет осуществлена при подтверждении пользователем заполненных данных. При этом вы имеете возможность сделать проверку еще раз, прежде чем отправлять данные.

Примечание

Не забудьте, что код на ActionScript следует размещать внутри тега <mx:Script/>.

Данная техника валидации не предполагает вызова функции validate() для каждого отдельного валидатора. Существует вспомогательная функция validateAll(), являющаяся статическим методом, принадлежащим классу mx.validators.Validator (базового класса для всех валидаторов). Вместо создания объекта Validator и вызова данного метода для объекта метод может быть вызван для самого класса Validator. Метод validateAll() имеет единственный параметр, значением которого является массив валидаторов. Не забудьте импортировать класс mx.validators.Validator, а затем внесите в функцию следующие изменения:

```
private function validateAndSubmit():void
{
  var validators:Array = [firstNameValidator,
```

```
                        lastNameValidator, emailValidator,
                        phoneValidator, zipCodeValidator];

    Validator.validateAll(validators);
}
```

Теперь нажатие на кнопку submitButton послужит сигналом к началу процесса проверки данных для всех перечисленных валидаторов.

Данную форму можно сделать еще более удобной с точки зрения пользователя. Предполагается, что если пользователь нажимает на кнопку, он уверен, что форма заполнена верно. Если это не так, появление красных рамок вокруг неверно заполненных полей тем не менее может остаться незамеченным. Однако у вас есть возможность использования компонента Alert, позволяющего вывести сообщение об ошибке во всплывающем окне, которое невозможно не заметить.

Для открытия такого окна используется статический метод show() класса Alert (о нем я вкратце упоминал в главе 6). Этот метод имеет два параметра. Первый из них содержит отображаемый текст сообщения об ошибке, второй – строку заголовка окна Alert. Таким образом, импортируйте класс mx.controls.Alert и осуществите вызов всплывающего окна с помощью Alert.show("Please fix that stuff.", "There were problems with your form.[1]").

Для проверки наличия или отсутствия ошибок при заполнении формы можно использовать массив, возвращаемый функцией Validator.validateAll(). Значение его свойства length определяет длину массива, т. е. количество содержащихся в нем элементов. Если длина массива больше нуля, были допущены ошибки. С этой целью мы будем использовать *условный оператор*, позволяющий определить, истинно или ложно некоторое выражение. Мы воспользуемся выражением if. Оно осуществляет выполнение определенного фрагмента кода при выполнении определенных условий: если передаваемое в него значение истинно, будет выполняться код, заключенный в фигурные скобки. В противном случае он будет пропущен. Теперь посмотрим, как все это выглядит на практике. Нам необходимо осуществить доступ к массиву, возвращаемому методом Validator.validateAll(). Если его длина (length) превышает нулевое значение, необходимо вывести сообщение об ошибке во всплывающем окне.

```
private function validateAndSubmit():void
{
  var validators:Array = [firstNameValidator,
                        lastNameValidator, emailValidator,
                        phoneValidator, zipCodeValidator];

  var errors:Array = Validator.validateAll(validators);
```

[1] «Пожалуйста, исправьте ошибки». «С вашей формой что-то не так.» (англ.). – *Прим. перев.*

```
    if(errors.length > 0)
    {
      Alert.show("Please fix that stuff.",
      "There were problems with your form.");
    }
  }
```

Если все вышесказанное не вызывает у вас затруднений, можно усложнить задачу. Попробуем вывести всплывающее окно Alert, не только сообщающее об ошибке, но и указывающее на ее характер.

Примечание

Мы погружаемся в самые дебри написания сценариев, и поначалу вы можете чувствовать себя немного не в своей тарелке. Вы всегда можете вернуться в данному разделу позднее. Кроме того, возможностей MXML-тегов инструментов валидации вполне достаточно для создания полноценной системы проверки данных.

Для этого нам придется познакомиться с еще одним понятием ActionScript – *циклами*. Цикл позволяет осуществлять неоднократное выполнение определенного кода до тех пор, пока не будет выполнено некоторое условие. Одним из основных способов задания цикла является `for each…in`. Это выражение вызывает выполнение фрагмента кода для каждого элемента массива (или другого объекта) и передает этот элемент в код. Вы можете обработать в цикле элементы массива ошибок (`errors`), элементами которого являются результаты проверки ValidationResultsEvents, используя следующий код:

```
for each (var error:ValidationResultEvent in errors)
{
  Alert.show(error.message, "There were problems with your form.");
}
```

Теперь для каждой ошибки будет отображаться отдельное всплывающее окно Alert. Обратите внимание на свойство `message` объекта `ValidationResultEvent`, значением которого является текст сообщения об ошибке. Но у данного приема есть один недостаток – при большом количестве ошибок будет выведено слишком много всплывающих окон. Поэтому целесообразнее сохранить все сообщения об ошибках в одном месте и затем отобразить их все в одном всплывающем окне. Для этого можно создать массив с именем `errorMessages` и добавить сообщения в массив. Для добавления элементов в массив используется метод `push()` объекта Array (массив). После цикла `for each…in` добавьте вызов окна Alert. Конечный вид функции будет примерно следующим:

```
<mx:Script>
  <![CDATA[
    import mx.validators.Validator;
    import mx.validators.ZipCodeValidatorDomainType;
    import mx.events.ValidationResultEvent;
```

```
        import mx.controls.Alert;
        private function validateAndSubmit():void
        {
          var validators:Array = [firstNameValidator,
                        lastNameValidator, emailValidator,
                        phoneValidator, zipCodeValidator];
          var errors:Array = Validator.validateAll(validators);
          var errorMessages:Array = [];

          if(errors.length > 0)
          {

            for each (var error:ValidationResultEvent in errors)
            {
              var errorField:String = FormItem
                      (error.currentTarget.source.parent).label;
                      errorMessages.push(errorField + ": " +
                      error.message);
            }
            Alert.show(errorMessages.join("\n\n"),
            "There were problems with your form.");
          }

            //Сюда можно добавить другие инструменты валидации
            //или осуществить отправку данных
        }

      ]]>
    </mx:Script>
```

Теперь в вашем приложении имеется полноценная форма (рис. 9.9). Следующим шагом будет установка ограничений для данных, вводимых пользователем в поля TextInput.

Ограничение ввода данных

Я ни разу не встречал человека с именем 7*% 32$, а вы? Не нужно ли запретить ввод подобных имен? Это вполне возможно. Свойство `restrict` позволяет сузить диапазон допустимых символов, что позволит избежать ввода определенных знаков. Изменив тег поля с идентификатором `firstNameTextInput` следующим образом, вы запретите использование любых символов, кроме строчных и прописных букв английского алфавита от A до Z.

```
<mx:TextInput id="firstNameTextInput"
  restrict="a-z A-Z"/>
```

Однако мы не учли один фактор. Что делать пользователю, если его имя пишется через дефис, например Day-Lewis, или с апострофом – к примеру, O'Reilly? Необходимо добавить несколько символов в список допустимых. Единственная сложность заключается в том, что дефис используется для определения диапазона значений и необходимо

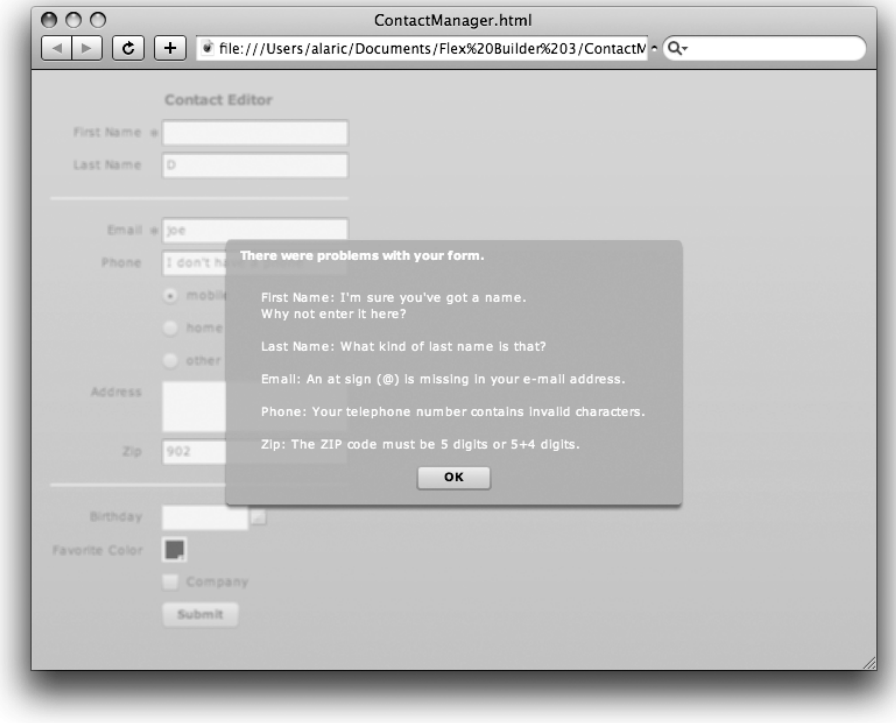

Рис. 9.9. *Сложная система проверки данных*

специальным образом указать, что нас интересует дефис в качестве самостоятельного символа. Для этого перед знаком дефиса поместите обратную косую черту (\).

```
<mx:TextInput id="firstNameTextInput"
  restrict="a-z A-Z ' \-"/>
```

Свойство `restrict` можно добавить к полям для ввода имени и фамилии, что позволит избежать ввода нежелательных символов.

Примечание

При установке ограничений на ввод символов в поля формы становится невозможным использование таких знаков, как символы национальных алфавитов и т. д. Во избежание такой ситуации можно использовать свойство `restrict` не для определения списка допустимых символов, а для указания *недопустимых*. Для этого перед последовательностью символов добавляется знак вставки (^). К примеру, чтобы запретить использование цифр, достаточно изменить рассматриваемое свойство следующим образом: `restrict="^0-9"`, при этом разрешается ввод любых символов кроме 0–9. В данном случае используется синтаксическая конструкция регулярных выражений, о которой вы можете узнать подробнее в документации Flex.

Аналогичным образом можно ограничить список допустимых символов для поля с номером телефона одними лишь цифрами, скобками и дефисами. Также можно ограничить и символы, используемые для ввода почтового индекса цифрами и дефисами. Однако стоит учесть, что в канадских почтовых индексах используются также буквы.

Примечание

Вам кажется удивительным, что я не рассказал о способах форматирования данных вроде телефонных номеров прямо при их вводе в соответствующее поле? Таким образом, чтобы при печати символов автоматически заключался в скобки код страны или добавлялся дефис после первых трех цифр телефонного номера? Как правило, это называется ввод по маске, и Flex не предоставляет готовых решений такой задачи. Однако такого эффекта можно с легкостью достичь при помощи компонентов, предлагаемых сторонними разработчиками.

Форматирование отображаемых данных

Flex предоставляет разработчику полноценный набор инструментов для форматирования всех основных типов данных. Благодаря таким инструментам у вас есть возможность хранить данные в «сыром» виде и при необходимости видоизменять их формат. Допустим, данные о ценах на продукты расположены в отдельной базе данных или XML-файле. Их удобнее всего хранить в виде цифр, что позволяет с легкостью проводить с ними различные операции (такие как расчет скидки, изменение вида валюты и т.п). Но вам бы безусловно хотелось, чтобы пользователи вашего приложения видели цены в привычном для них формате – со знаком доллара, запятой в качестве разделителя разряда тысяч и округлением до сотых, как обычно и бывают представлены цены в долларах США.

Приведение данных к определенному виду – задача инструментов форматирования. Такие инструменты позволяют форматировать наиболее часто встречающиеся типы данных – валюту, даты, числа, телефонные номера, почтовые индексы и т. д. Как и валидаторы, они являются невизуальными компонентами, поэтому для их использования необходимо создать соответствующий тег. Вместо указания в теге ссылки на источник форматирования применяется функция `format()`, возвращающая текстовую строку. К примеру, определим тег `CurrencyFormatter`:

```
<mx:CurrencyFormatter id="priceFormatter"/>
```

В дальнейшем потребуется осуществить вызов функции форматирования, передав ей числовое значение: `priceFormatter.format(10243)`. В результате выполнения функции мы получим строку «$10,243». Обратите внимание на появление запятой. По умолчанию она отделяет разряды тысяч, но если вы предпочитаете запись без запятой, эту настройку можно изменить с помощью свойства `useThousandsSeparator`, присвоив

ему значение false. При использовании инструментов форматирования широко применяется технология связывания данных. Во Flex можно осуществить связывание с возвращаемым функцией значением, и следующий код может поместить ту же строку, что и в предыдущем примере, в качестве значения свойства text элемента TextInput:

```
<mx:CurrencyFormatter id="priceFormatter"/>
<mx:TextInput text="{priceFormatter.format(10243)}"/>
```

Конечно, возможности данного инструмента этим вовсе не исчерпываются; вы можете использовать массу других полезных опций, например, округление числа в большую или меньшую сторону (rounding) или определение количества десятичных знаков (precision). Допустим, нам нужно указать цену в евро, и при этом округлить центы до целого евро. Справиться с такой задачей не составит труда, если вам известно, какие свойства нужно использовать. В данном случае необходимо изменить значение currencySymbol на евро (€) вместо используемого по умолчанию знака доллара ($). Кроме того, укажем, что необходимо округлить число до целого и показать два десятичных знака.

```
<mx:CurrencyFormatter id="priceFormatter"
  currencySymbol="€"
  rounding="up"
  precision="2"/>
```

В результате получим «€10,243.00.».

Конечно, существуют инструменты для форматирования дат, телефонных номеров, почтовых индексов и других данных, типично используемых при заполнении форм. Аналогично вышеприведенному примеру, можно использовать тег PhoneFormatter для приведения данных, введенных в качестве телефонного номера, в формат (###) ###-####:

```
<mx:PhoneFormatter id="phoneFormatter"/>
<mx:TextInput text="{phoneFormatter.format('4795558273')}"/>
```

Результатом отображения будет «(479) 555-8273.»

Теперь вернемся к приложению ContactManager и заново создадим панель Contact Details, добавив возможность форматирования данных. Разместите теги инструментов PhoneFormatter и DateFormatter прямо под тегами валидаторов в файле *ContactManager.mxml*. Повторюсь, такое расположение *не обязательно*, но желательно.

```
<mx:PhoneFormatter id="phoneFormatter"/>

<mx:DateFormatter id="dateFormatter"/>
```

Теперь добавим панель Panel с несколькими вложенными элементами Label, привязав их к соответствующим инструментам форматирования:

```
<mx:Panel id="contactDetails"
  x="346"
```

```
    y="10"
    paddingLeft="5"
    paddingRight="5"
    paddingTop="5"
    paddingBottom="5"
    title="Contact Details">
    <mx:Label
      text="Name:"
      fontWeight="bold"/>
    <mx:Label
      text="{firstNameTextInput.text} {lastNameTextInput.text}"/>
    <mx:Label
      text="Phone Number ( {phoneRadioButtonGroup.selectedValue} )"
      fontWeight="bold"/>
    <mx:Label
      text="{phoneFormatter.format(phoneTextInput.text)}"/>
    <mx:Label
      text="Birthday:"
      fontWeight="bold"/>
    <mx:Label
      text="{dateFormatter.format (birthdayDateField.selectedDate)}"/>
    <mx:Label
      text="Company:"
      fontWeight="bold"/>
    <mx:Label
      text="{companyCheckBox.selected}"/>
    <mx:Label
      text="Favorite Color:"
      fontWeight="bold"/>
    <mx:Canvas
      width="60"
      height="60"
      backgroundColor="{favoriteColorPicker.selectedColor}"/

  </mx:Panel>
```

В данном коде следует обратить внимание на несколько моментов. Во-первых, использование свойства `selectedValue` элемента RadioButtonGroup с идентификатором `phoneRadioButtonGroup`. Его значение соответствует имени выбранного переключателя (mobile, home или other). Кроме того, обратите внимание на осуществление привязки к свойству `selectedDate` элемента DateField. Данное значение передается в функцию `format()` инструмента DateFormatter. В результате ее выполнения возвращается отформатированное значение типа String. В противном случае значение `selectedDate` по умолчанию может напомнить таинственные письмена, поскольку будет содержать также и время, поэтому целесообразно воспользоваться инструментом для форматирования.

Вместо простого отображения даты рождения пользователя можно вывести его возраст, который можно рассчитать с помощью пары методов класса `Date` и капельки упорства. Посмотрите на следующую функ-

цию, которая принимает объект Date в качестве параметра и в результате выполнения выдает числовое значение возраста пользователя:

```
private function calculateAge(birthDate:Date):Number
{
  var today:Date = new Date();
  var ageDate:Date = new Date(today.time - birthDate.time);
  var age:Number = ageDate.fullYear - 1970;

  return age;
}
```

В данном примере создается объект Date, соответствующий текущей дате и времени. Для получения возраста пользователя необходимо вычислить разницу между текущей датой и датой его рождения. Однако нельзя произвести вычитание одного объекта Date из другого. Но объект Date обладает свойством time, представляющим числовое значение хранящихся в нем данных, с которым можно производить вычисления. Итак, для вычисления возраста пользователя с помощью данного свойства можно вычесть значение даты рождения из значения текущей даты. Далее полученный результат преобразовывается в количество лет с помощью свойства fullYear. (Поскольку стандартной точкой отсчета дат объекта Date является 1970, для получения конечного результата необходимо вычесть 1970.)

Примечание

Значение свойства time объекта Date – это количество миллисекунд, прошедших после полуночи 1 января 1970 года. Таков внутренний способ хранения данных данного объекта.

Вычисленное значение можно привязать к свойству text элемента Label. Для этого необходимо передать значение свойства selectedDate элемента DateField в функцию calculateAge() и добавить в конце строку «years old» («лет»). При изменении поля DateField обновленное значение отобразится элементом Label. Здесь важно отметить, что изначально значением данного элемента является пустая строка, т. е. до ввода данных в поле DateField не появится невразумительного выражения « years old», поскольку до тех пор, пока пользователь не введет дату рождения, связывание осуществлено не будет. Далее при каждом обновлении поля DateField метка будет изменена соответствующим образом.

```
<mx:Label
  text="{calculateAge(birthdayDateField.selectedDate)} years old"/>
```

При запуске приложения после добавления данного фрагмента кода вы наверняка обратите внимание на нечто странное. Поскольку вычисленный нами возраст пользователя может быть дробным числом (с десятичной частью), он может отображаться, к примеру, следующим образом: «28.662405 years old». Вряд ли это соответствует вашим

ожиданиям. Однако и здесь инструменты форматирования придут на помощь.

Итак, напоследок добавим NumberFormatter для форматирования данных о возрасте. Использование его свойства `rounding` позволяет округлить число до ближайшего целого значения, как в следующем примере:

```
<mx:NumberFormatter id="numberFormatter"
  rounding="down"/>
```

Далее достаточно лишь передать результат выполнения функции `calculateAge()` в качестве параметра функции `format()` данного объекта NumberFormatter:

```
<mx:Label
  text="{numberFormatter.format(calculateAge(birthdayDateField.
  selectedDate))} years old"/>
```

Теперь все отлично. На рис. 9.10 можно увидеть примерный вид вашего приложения.

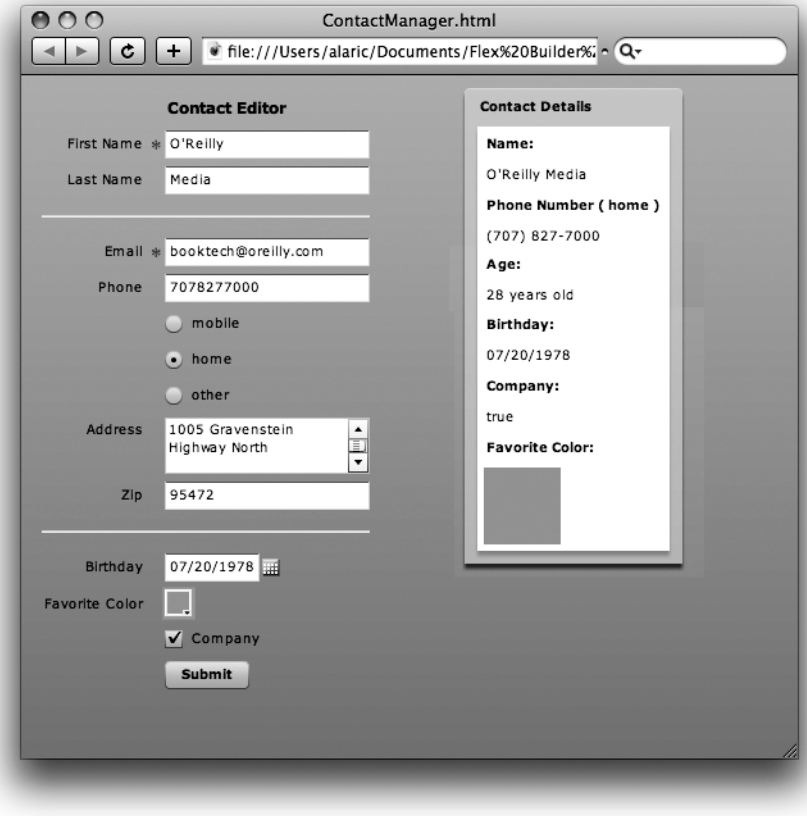

Рис. 9.10. *Применение инструментов форматирования данных на примере панели Contact Details*

Примечание

Класс Math обладает рядом методов для проведения вычислений и округления результатов. Например, метод Math.floor() позволяет округлить результат в меньшую сторону. Таким образом, вместо NumberFormatter для округления числа можно было использовать функцию calculateAge(). В нашем случае для этого достаточно заменить последнюю строку кода функции, содержащую `return age;`, на `return Math.floor(age);`.

Ниже приведен окончательный вариант кода панели Contact Details:

```
<mx:Panel id="contactDetails"
  layout="vertical"
  x="346"
  y="10"
  paddingLeft="5"
  paddingRight="5"
  paddingTop="5"
  paddingBottom="5"
  title="Contact Details">
  <mx:Label
    text="Name:"
    fontWeight="bold"/>
  <mx:Label
    text="{firstNameTextInput.text} {lastNameTextInput.text}"/>
  <mx:Label
    text="Phone Number ( {phoneRadioButtonGroup.selectedValue} ):"
    fontWeight="bold"/>
  <mx:Label
    text="{phoneFormatter.format(phoneTextInput.text)}"/>
  <mx:Label
    text="Birthday:"
    fontWeight="bold"/>
  <mx:Label
    text="{dateFormatter.format (birthdayDateField.selectedDate)}"/>
  <mx:Label
    text="Age:"
    fontWeight="bold"/>
  <mx:Label
    text="{numberFormatter.format
     (calculateAge(birthdayDateField.
    selectedDate))} years old"/>
  <mx:Label
    text="Company:"
    fontWeight="bold"/>
  <mx:Label
    text="{companyCheckBox.selected}"/>
  <mx:Label
    text="Favorite Color:"
    fontWeight="bold"/>
  <mx:Canvas
```

```
        width="60"
        height="60"
        backgroundColor="{favoriteColorPicker.selectedColor}"/>
    </mx:Panel>
```

Заключение

В этой главе вы узнали о технике валидации данных, начиная с самых азов и заканчивая сложными приемами. Вы научились пользоваться различными инструментами валидации и даже создали полноценную систему проверки данных с помощью ActionScript. Теперь вы также умеете ограничивать диапазон допустимых для ввода символов и использовать такие элементы, как RadioButton и DateField.

Я затронул такие важные понятия программирования, как условные операторы и циклы, и подробно рассказал о форматировании данных. Мы рассмотрели пример создания реального приложения с использованием различных методов форматирования. Теперь вы можете без труда проводить разного рода вычисления с датами и временем. Полученные в данной главе знания, а также умение красиво располагать элементы приложения помогут вам создавать по-настоящему удобные формы, обладающие широчайшими возможностями.

10

В этой главе:

- Использование элементов управления списком
- Использование данных в формате XML
- Выбор пункта из списка
- Подключение результатов поиска
- Перемещение элементов списка
- Использование встроенных представлений элементов
- Знакомство с другими сервисными компонентами

Сбор и отображение информации

Едва ли возможно создать полноценное приложение без использования каких-либо внешних данных. С наступлением эры веб-приложений данное утверждение становится еще актуальнее. Вспомните наиболее часто используемые вами веб-приложения. Будет ли смысл в их применении, если доступ к каким-либо дополнительным данным по каким-либо причинам станет невозможен? Ведь именно ради свободного получения информации мы имеем дело с таким явлением, как всемирная паутина.

Flex позволяет без труда подключать и отображать данные, получаемые из самых различных источников – XML-файлов, баз данных или специализированных веб-сервисов.

Использование элементов управления списком

Во Flex существует ряд элементов управления, предназначенных для отображения данных в виде списка. Все они работают с данными со структурой любой степени сложности и предоставляют разработчику широкие возможности их настройки и многократного использования. Если длина списка превышает размер элемента, появляются полосы прокрутки, позволяющие увидеть его содержимое целиком. Ниже представлен перечень таких элементов, наиболее часто используемых при разработке приложения:

List

Основа всех элементов управления для отображения списков. Располагает пункты списка по вертикали.

HorizontalList

Располагает пункты списка по горизонтали.

TileList

Элементы размещаются подобно плитке.

ComboBox

Данный элемент напоминает TextInput, но дает пользователю дополнительную возможность выбора вводимого значения с помощью выпадающего списка. Его можно условно сравнить с HTML-тегом `<select/>`.

DataGrid

Элемент с расширенными возможностями, предназначенный для табличной организации наборов данных. Можно сортировать ряды, а также изменять размер колонок и даже менять их местами, просто перетаскивая с помощью мыши.

Простые списки данных

Для начала в качестве примера приведем простой список названий цветов. Информация, представленная в виде списка элементов, как правило, хранится в *массиве*.

Массив можно создать с помощью MXML-тега `<mx:Array/>`. Элементы списка располагаются внутри вложенных тегов, например `<mx:String/>`. В нашем случае список с названиями цветов можно создать следующим образом:

```
<mx:Array>
  <mx:String>red</mx:String>
  <mx:String>green</mx:String>
  <mx:String>blue</mx:String>
</mx:Array>
```

Для создания массива в ActionScript перечень его элементов, разделенных запятой, заключается в квадратные скобки ([и]). Таким образом, точно такой же список можно создать и средствами ActionScript:

```
var colors:Array = [ "red","green","blue" ];
```

Теперь вы умеете создавать массивы; перейдем к способу их отображения с помощью специальных элементов. Для этого используется свойство `dataProvider`, передающее массив в элемент управления списком. К примеру, чтобы вывести на экран перечень цветов с помощью элемента управления List, можно привязать данное свойство к соответствующему массиву:

```
 <mx:List
width="150"
    dataProvider="{['Red', 'Orange', 'Yellow', 'Green', 'Blue', 'Indigo',
    'Violet']}" />
```

В результате мы получим список, изображенный на рис. 10.1.

***Рис. 10.1.** Список цветов*

Примечание

Любой Flex-компонент, предназначенный для работы со списками, обладает свойством `dataProvider`.

Вы уже знаете, что существуют два способа записи свойств элемента – в качестве XML-атрибута или вложенного тега. Поэтому аналогичный список элементов массива можно создать с помощью одних тегов, используя свойство `dataProvider` в виде дочернего тега `<mx:dataProvider/>`:

```
<mx:List
  width="150">

  <mx:dataProvider>

    <mx:Array>
      <mx:String>Red</mx:String>
      <mx:String>Orange</mx:String>
      <mx:String>Yellow</mx:String>
      <mx:String>Green</mx:String>
      <mx:String>Blue</mx:String>
      <mx:String>Indigo</mx:String>
      <mx:String>Violet</mx:String>
    </mx:Array>

  </mx:dataProvider>

</mx:List>
```

Дочерний тег `<mx:dataProvider/>` также имеет вложенные теги, с помощью которых создается массив названий цветов, как и в рассмотренном ранее примере MXML-кода.

Примечание

Обратите внимание на пространство имен mx в теге <mx:dataProvider/>.

Такое решение идеально при работе с небольшими списками, но при наличии большого количества элементов код существенно удлиняется. Как правило, в этом случае данные помещаются в отдельный файл, который затем можно загрузить, или же данные запрашиваются с помощью веб-сервиса.

Примечание

Списки также обладают свойством rowCount, позволяющим определить количество видимых пунктов.

Сложные списки данных

В предыдущих двух примерах источником данных для списка служил массив строк. Элемент управления списком может работать с массивами произвольных данных. Предположим, вам нужно составить список любимых песен. Конечно, можно ограничиться лишь названиями, но скорее всего вам захочется добавить и некоторую дополнительную информацию, например имя исполнителя и альбом, в который входит данная песня. В таком случае вместо массива текстовых строк гораздо удобнее воспользоваться массивом объектов. Объект является идеальным контейнером для хранения набора данных, поскольку ему можно присвоить какие угодно свойства. К примеру, нам понадобятся свойства artist, album и song:

```
<mx:List
  width="150">

  <mx:dataProvider>

    <mx:Array>

      <mx:Object
        song="In My Secret Life"
        album="Ten New Songs"
        artist="Leonard Cohen"/>
      <mx:Object
        song="Phantom Limb"
        album="Wincing the Night Away"
        artist="The Shins"/>
      <mx:Object
        song="Tinfoil"
        album="Live at Schuba's Tavern"
        artist="The Handsome Family"/>
      <mx:Object
        song="Highway 253"
        album="Extra Solar Sunrise"
        artist="The Saturn V"/>
```

```
        <mx:Object
          song="Junk Bond Trader"
          album="Figure 8"
          artist="Elliott Smith"/>
        <mx:Object
          song="Stalled"
          album="Through the Trees"
          artist="The Handsome Family"/>
        <mx:Object
          song="Every Dull Moment"
          album="Bring on the Snakes"
          artist="Crooked Fingers"/>
      </mx:Array>
    </mx:dataProvider>
  </mx:List>
```

В результате у нас получился список, пунктами которого являются объекты с заданными свойствами. Но посмотрите на рис. 10.2 – на нем отображается такой список.

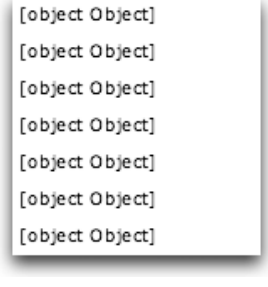

Рис. 10.2. *Список объектов, буквально*

Думаю, вы стремились создать нечто иное. Что же произошло? Дело в том, что поскольку вместо текстовых строк пункты представлены объектами и элементу списка не было указано, что именно нужно отобразить, на экран была выведена строка, по умолчанию используемая для представления объекта – `[object Object]`.

Выбор отображаемого свойства объекта

К счастью, у элементов списка есть свойство `labelField`, определяющее, какое именно из свойств объекта со сложной структурой данных должно быть отображено. В нашем случае это название песни, поэтому просто присвойте данному свойству значение `song`. После этого список будет выглядеть совсем по-другому (рис. 10.3).

По умолчанию значением свойства `labelField` является `label`. Благодаря такой настройке можно воспользоваться еще одним способом ука-

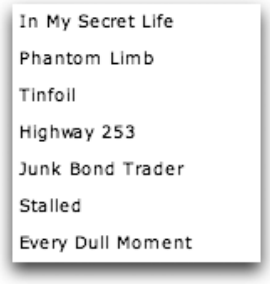

Рис. 10.3. *Список объектов, отображающий названия песен*

зания отображаемого свойства, изменив свойство song объектов массива на label. Однако я не рекомендую вам применять этот метод на практике: не нужно вносить никаких изменений в ваши данные, поскольку элемент управления списком для того и предназначен, чтобы обработать и отобразить их в соответствующем виде.

Списки с расширенными возможностями представления данных

Если вам хочется, чтобы одновременно отображалось несколько свойств объекта, на помощь придет элемент DataGrid, созданный специально для выполнения такого рода задач. Он позволяет создавать несколько колонок, каждая их которых соответствует свойству передаваемого объекта. По сути данный элемент похож на электронные или HTML-таблицы.

DataGrid проще всего создавать в режиме Design. При перенесении его на сцену с помощью указателя мыши Flex Builder автоматически пишет нужный код, который в дальнейшем при необходимости можно изменять. Пример такого кода приведен ниже:

```
<mx:DataGrid>
  <mx:columns>
    <mx:DataGridColumn headerText="Column 1" dataField="col1"/>
    <mx:DataGridColumn headerText="Column 2" dataField="col2"/>
    <mx:DataGridColumn headerText="Column 3" dataField="col3"/>
  </mx:columns>
</mx:DataGrid>
```

Количество колонок DataGrid определяется свойством columns, которому присваивается последовательность элементов DataGridColumn, в каждом из которых в свою очередь необходимо указать свойство для отображения с помощью атрибута dataField. По умолчанию заголовок колонки соответствует названию отображаемого в ней свойства, но в большинстве случаев потребуется задать свой собственный текст. Это можно сделать с помощью свойства headerText.

Для указания ссылки на источник данных для элемента DateField, как и для других элементов списка, служит свойство dataProvider. Итак, для создания таблицы можно использовать следующий код, во многом повторяющий содержимое тега <mx:dataProvider/> в предыдущем примере:

```
<mx:DataGrid>

  <mx:columns>
    <mx:DataGridColumn headerText="Track Name"
                       dataField="song"/>
    <mx:DataGridColumn headerText="Artist"
                       dataField="artist"/>
    <mx:DataGridColumn headerText="Album Name"
                       dataField="album"/>
  </mx:columns>

  <mx:dataProvider>
    <mx:Array>
      <mx:Object
        song="In My Secret Life"
        album="Ten New Songs"
        artist="Leonard Cohen"/>
      <mx:Object
        song="Phantom Limb"
        album="Wincing the Night Away"
        artist="The Shins"/>
      <mx:Object
        song="Tinfoil"
        album="Live at Schuba's Tavern"
        artist="The Handsome Family"/>
      <mx:Object
        song="Highway 253"
        album="Extra Solar Sunrise"
        artist="The Saturn V"/>
      <mx:Object
        song="Junk Bond Trader"
        album="Figure 8"
        artist="Elliott Smith"/>
      <mx:Object
        song="Stalled"
        album="Through the Trees"
        artist="The Handsome Family"/>
      <mx:Object
        song="Every Dull Moment"
        album="Bring on the Snakes"
        artist="Crooked Fingers"/>
    </mx:Array>
  </mx:dataProvider>

</mx:DataGrid>
```

В результате получим список, изображенный на рис. 10.4.

Track Name	Artist	Album Name
In My Secret Life	Leonard Cohen	Ten New Songs
Phantom Limb	The Shins	Wincing the Night Away
Tinfoil	The Handsome Family	Live at Schuba's Tavern
Highway 253	The Saturn V	Extra Solar Sunrise
Junk Bond Trader	Elliott Smith	Figure 8
Stalled	The Handsome Family	Through the Trees

Рис. 10.4. *Те же самые песни, представленные с помощью таблицы DataGrid*

Примечание

DataGridColumn обладает рядом других свойств для изменения настроек конкретной колонки. К примеру, ее размер можно установить с помощью свойства `width`.

Использование данных в формате XML

Мы рассмотрели несколько примеров составления списков с использованием массивов простых элементов и массивов объектов. Но для этого можно использовать и данные в формате XML.

Чтобы разобраться в том, что из себя представляют такие данные, вернемся к приложению ContactManager и создадим список контактов с помощью XML. Для этого необходимо продумать, какую именно информацию из формы необходимо сохранить. В данном случае это не составит труда, поскольку у вас уже есть готовый пользовательский интерфейс с некоторыми полями. Набросок нашего XML-файла будет примерно следующим:

```
<contacts>
  <contact id="">
    <firstName/>
    <lastName/>
    <email/>
    <phone/>
    <phoneType/>
    <address/>
    <zip/>
    <birthday/>
    <color/>
    <company/>
  </contact>
</contacts>
```

Начальный тег `<contacts/>` является корневым тегом, необходимым в обязательном порядке любому XML-документу. Он может включать

в себя произвольное количество вложенных тегов <contact/>, каждый из которых обладает атрибутом id, служащим в качестве числового идентификатора данного контакта. Каждый тег <contact/> может иметь сколько угодно дочерних элементов, отвечающих за его отдельные свойства. Эти свойства могут быть выражены и через атрибуты, но перспективнее использовать дочерние теги, поскольку в дальнейшем вы сможете при необходимости добавлять свойства самим дочерним элементам или включать в них содержимое с более сложной структурой.)

Примечание

Атрибут id XML-тегов не имеет никакого отношения к идентификатору id Flex-компонентов. При желании можно изменить название данного атрибута на свой вкус; скажем, name или number.

Итак, попробуйте создать свой собственный перечень контактных данных, используя только что приведенный шаблон или нижеследующий листинг.

Для использования структурированной с помощью XML информации в MXML-коде применяется тег <mx:XML/>. Поместите внутри данного тега следующие данные и добавьте данный код в приложение ContactManager.

```
<mx:XML id="contactsXML" xmlns="" >
  <contacts>
    <contact id="0">
      <firstName>Alaric</firstName>
      <lastName>Cole</lastName>
      <email>alaric@oreilly.com</email>
      <phone>4155558273</phone>
      <phoneType>mobile</phoneType>
      <address>555 Green St</address>
      <zip>94001</zip>
      <birthday>07/20/1979</birthday>
      <color>0x00FF00</color>
      <company>false</company>
    </contact>
    <contact id="1">
      <firstName>O'Reilly</firstName>
      <lastName>Media</lastName>
      <email>booktech@oreilly.com</email>
      <phone>7078277000</phone>
      <phoneType>home</phoneType>
      <address>1005 Gravenstein Highway North, Sebastopol, CA
      </address>
      <zip>95472</zip>
      <birthday>07/20/1978</birthday>
      <color>0x009999</color>
      <company>true</company>
    </contact>
```

```
    <contact id="2">
      <firstName>Crystal</firstName>
      <lastName>Clear</lastName>
      <email>crystal4354@hotmail.com</email>
      <phone>5015556492</phone>
      <phoneType>mobile</phoneType>
      <address>555 Lakeview Dr</address>
      <zip>94001</zip>
      <birthday>3/6/1975</birthday>
      <color>0xFF66CC</color>
      <company>false</company>
    </contact>
    <contact id="3">
      <firstName>Google</firstName>
      <email>google@gmail.com</email>
      <phone>6502530000</phone>
      <phoneType>home</phoneType>
      <address>1600 Amphitheatre Parkway, Mountain View, CA
      </address>
      <zip>94043</zip>
      <color>0xFF0000</color>
      <company>true</company>
    </contact>
    <contact id="4">
      <firstName>Yahoo!</firstName>
      <email>yahoo@yahoo.com</email>
      <phone>4083493300</phone>
      <phoneType>home</phoneType>
      <address>701 First Avenue, Sunnyvale CA</address>
      <zip>94089</zip>
      <color>0x9900FF</color>
      <company>true</company>
    </contact>
    <contact id="5">
      <firstName>Whatcha</firstName>
      <lastName>McCollum</lastName>
      <email>w.mccollum@fakeemail.com</email>
      <phone>3149884735</phone>
      <phoneType>other</phoneType>
      <color>0xFFFF00</color>
      <company>false</company>
    </contact>
  </contacts>
</mx:XML>
```

В приведенном примере атрибуту `id` тега `<mx:XML/>` присвоено значение `contactsXML`, и вдобавок определено пустое пространство имен. При работе в режиме Source последняя операция может быть осуществлена автоматически благодаря функции автозаполнения. В данном случае, поскольку мы имеем дело с простой структурой данных, это не обязательно, но и не помешает.

Примечание

В соответствии с правилами при составлении XML-файла необходимо использовать корневой тег, поэтому в нашем примере контактные данные заключены между тегами <contacts> и </contacts>. Корневому тегу можно дать любое название, поскольку на него не будет ссылок, а для обращения к корневому уровню можно просто использовать значение атрибута id тега <mx:XML/>.

Теперь вся необходимая информация имеется у вас в наличии, остается отобразить ее в приложении с помощью DataGrid. При желании можете изменить расположение ваших компонентов, чтобы освободить дополнительное пространство, а затем перетащить элемент DataGrid на сцену в режиме Design. Задайте ему идентификатор contactsDataGrid, переключитесь в режим Design и внесите следующие изменения в отношении свойства dataProvider и колонок таблицы:

```
<mx:DataGrid id="contactsDataGrid"
  dataProvider="{contactsXML.contact}">
  <mx:columns>
    <mx:DataGridColumn headerText="ID"
                       dataField="@id"/>
    <mx:DataGridColumn headerText="First"
                       dataField="firstName"/>
    <mx:DataGridColumn headerText="Last"
                       dataField="lastName"/>
  </mx:columns>
</mx:DataGrid>
```

Для привязки свойства dataProvider элемента ContactGrid к только что созданному XML-документу с именем contactsXML используются фигурные скобки. Обратите внимание, что привязка осуществляется не прямо к contactsXML, а к contactsXML.contact, иначе элемент DataGrid останется пустым:

```
<mx:DataGrid id="contactsDataGrid"
  dataProvider="{contactsXML}"/>
```

Это объясняется тем, что XML сам по себе не является списком данных, который передается элементу управления для отображения. В данном случае выражение contactsXML ссылается на узел верхнего уровня XML-структуры, но он является единым целым, а вовсе не списком элементов, и потому будет отображаться некорректно. Для доступа к собственно списку используется выражение contactsXML.contact, указывающее на перечень тегов <contact/>.

Обратите внимание на первую колонку в таблице DataGrid. Значением ее свойства dataField является не просто id, а @id, поскольку в данном случае это атрибут XML, а не дочерний тег. Напротив, firstName и lastName являются дочерними тегами в XML-коде, и в ссылках на них знак @ отсутствует. Его наличие указывает на обращение к атрибуту, ведь

в XML могут быть как атрибуты, так и дочерние теги, и нужно каким-то образом определить, что именно необходимо в данном случае.

ID	First	Last
0	Alaric	Cole
1	O'Reilly	Media
2	Crystal	Clear
3	Google	
4	Yahoo!	
5	Whatcha	McCollum

Рис. 10.5. *Список контактных данных, представленный с помощью XML, отображается в таблице DataGrid*

Примечание

Почему именно знак @? В английском языке он произносится «эт», что созвучно слову «этрибьют» (англ. атрибут).

Сами о том не подозревая, вы только что освоили новую технику, называемую E4X. Внешне ее использование напоминает точечную нотацию в ActionScript, но на самом деле это разные вещи. E4X обладает рядом полезных функций, таких как фильтрация и возможность работы с XML-данными через расширенный API.

Примечание

E4X расшифровывается «ECMAScript for XML». Этот язык лежит в основе ActionScript. Более подробная информация о его фунцкиях и принципах использования содержится в документации Flex.

Теперь вы получили общее представление о E4X, и можно удалить первую колонку DataGridColumn, ведь отображение `id` контактной информации будет излишним – я привел это лишь в качестве примера, объясняющего способ доступа к данному атрибуту с помощью E4X.

Загрузка внешних данных в процессе компиляции

Теперь в вашем Flex-приложении используются данные, структурированные с помощью XML, и вы можете наполнить ими приложение. Однако взгляните на исходный код; вам не кажется, что он немного громоздок? Уже сейчас дополнительный код на XML может привести к затруднениям при его чтении и разборе по частям. Представьте, что будет, если в нем будет представлена контактная информация на 200 человек? В этом случае удобно вынести данный код в отдельный файл и затем

подключить его к приложению напрямую. Для ссылки на файл тегу <mx:XML/> присваивается свойство source.

Для начала создайте чистый XML-файл, выбрав File→New→File. В качестве места сохранения файла выберите папку с исходным кодом текущего проекта ContactManager (т. е. папку src) и назовите файл **contacts.xml** (рис. 10.6). При этом будет создан новый файл с расширением .xml, в который мы и поместим содержимое тега <mx:XML/> (для этого достаточно скопировать всю информацию, заключенную между тегами <mx:XML> и </mx:XML> и вставить ее в файл). Затем присвоим свойству source тега <mx:XML/> значение, указывающее путь к только что созданному файлу:

```
<mx:XML id="contactsXML" xmlns="" source="contacts.xml" >
</mx:XML>
```

Данную запись можно сократить следующим образом. К тому же, такой код гораздо легче воспринимать.

```
<mx:XML id="contactsXML" xmlns="" source="contacts.xml" />
```

Такое расположение XML-данных – во внешнем файле – очень удобно. Однако несмотря на то, что теперь данные хранятся отдельно, они попрежнему являются частью приложения (т. к. включаются в него при

Рис. 10.6. *Диалоговое окно New File*

компиляции). Во многих случаях было бы еще удобнее, если бы внешний источник данных подключался при запуске приложения, т. е. после его полной загрузки. При использовании такого метода в XML-файл можно без труда вносить изменения, и в вашем приложении всегда будет использоваться только самая последняя версия. Для решения такой задачи нам впервые придется использовать компонент сервиса – HTTPService.

Примечание

При загрузке внешнего файла через броузер всегда есть вероятность, что он уже был сохранен в кэше. Более подробно такой случай рассматривается во врезке «Кэш броузера и загрузка данных» в главе 11.

Загрузка внешних данных при запуске приложения

С помощью тега `<mx:HTTPService/>` осуществляется доступ к текстовым или XML-данным через Интернет по стандартному протоколу HTTP (обратитесь к врезке «Секреты папки bin-debug».). Это означает, что вы сможете загружать информацию с HTML-сайтов (или даже с сайтов, использующих технологию PHP, ASP и иных веб-серверов, генерирующих HTML- данные).

Компонент HTTPService обладает свойством `url`, позволяющим указать путь к требуемому файлу для загрузки. Если вы уже работали с HTML или имели дело с HTTP иным способом, вы прекрасно знаете, что можно использовать как абсолютный URL-адрес (например, *http://www.oreilly.com*), так и относительный (по отношению к вашему серверу), например, *images/someimage.jpg*.

Поскольку в нашем случае XML-файл является частью нашего проекта, мы будем использовать относительный URL-адрес.

HTTP-протокол

Аббревиатура HTTP расшифровывается «как протокол передачи гипертекста» (Hypertext Transfer Protocol) и является стандартом в отношении передачи данных в сети Интернет. Именно этот протокол используется при открытии страницы броузером для передачи содержащейся в ней текстовой и графической информации.

Ввод адреса веб-страницы в окошке вашего броузера вы, возможно, начинаете с выражения *http://*. (В противном случае оно будет добавлено автоматически.) Это указывает на использование протокола HTTP.

Примечание

При запуске Flex-программы в качестве веб-приложения через броузер относительный URL-адрес вычисляется относительно текущей HTML-страницы, из которой загружено приложение.

Поместите следующий код туда, где ранее был расположен тег <mx:XML/>:

```
<mx:HTTPService id="contactsService"
  resultFormat="e4x"
  url="contacts.xml"/>
```

Секреты папки bin-debug

При запуске веб-приложений через Flex Builder открывается HTML-страница, отображающая скомпилированную версию приложения в качестве встроенного объекта. Но компиляция – нечто большее, чем простое создание SWF-файла. При запуске данного процесса все имеющиеся в папке src файлы автоматически копируются в папку с названием bin-debug. В нашем примере в указанную папку будет помещена и копия файла *contacts.xml*, как показано на рис. 10.7. При запуске приложения к нему подключается именно эта копия из папки bin-debug, а не оригинальный XML-файл. Такой принцип справедлив и для других видов данных, подключаемых к приложению извне, о чем я еще расскажу подробнее в этой книге.

При изменении файлов, расположенных в папке bin-debug, они будут перезаписаны, поэтому имеет смысл редактировать только их оригиналы, находящиеся в папке src.

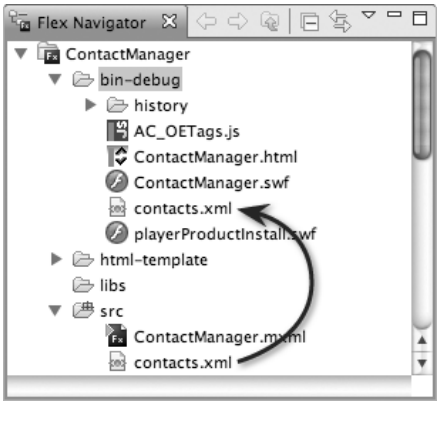

***Рис. 10.7.** Копирование дополнительных файлов в папку bin-debug*

Таким образом создается компонент HTTPService, ссылающийся на созданный нами ранее файл *contacts.xml*. Также определен XML-формат загружаемых данных. Необходимо указать свойство `resultFormat`, поскольку по умолчанию принимается его значение `object` и получаемые данные будут преобразованы компонентом в объекты, что приводит к невозможности использования E4X. Чтобы этого не произошло, измените значение данного свойства на `e4x`. Это позволит использовать данную технологию, а также укажет компоненту на то, что загружаемые данные находятся в формате XML.

На следующем этапе необходимо изменить значение свойства `dataProvider` элемента DataGrid, поскольку тег `<mx:XML/>` уже удален. Для указания ссылки на данные, получаемые из сервиса, используется свойство сервиса `lastResult`, значение которого содержит данные, полученные в результате последнего вызова данного сервиса:

```
<mx:DataGrid id="contactsDataGrid"
  dataProvider="{contactsService.lastResult.contact}">
  <mx:columns>
    <mx:DataGridColumn headerText="First"
                       dataField="firstName"/>
    <mx:DataGridColumn headerText="Last"
                       dataField="lastName"/>
  </mx:columns>
</mx:DataGrid>
```

Вам известно, что свойство `contactsService.lastResult` содержит возвращаемые сервисом данные в формате XML, однако в `dataProvider` нужно указать точную ссылку на список контактов. Для этого используется выражение `contactsService.lastResult.contact`.

Для решения поставленной задачи необходимо сделать еще один шаг. Дело в том, что запуск компонента сервиса не производится автоматически – для этого нужно использовать метод `send()`. Итак, установим в теге `<mx:Application/>` обработчик события `applicationComplete`. Данное событие происходит по окончании полной загрузки приложения, после чего будет вызван метод `send()`. Начальная часть вашего тега может выглядеть примерно так:

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

Теперь при запуске приложения будет инициирована работа сервиса и получены данные, которые благодаря технологии связывания данных будут отображены в таблице DataGrid. Как вам такое развитие событий?

Примечание

Метод, вызывающий работу сервиса, называется `send()` (англ. «посылать»), потому что он посылает соответствующий запрос.

В этом нет вашей вины

При использовании компонента HTTPService возможность получения данных, находящихся на удаленном сервере, зависит от качества его работы. Как и веб-броузер, он использует протокол HTTP. Вы наверняка сталкивались с такой ситуацией, когда запрашиваемая страница не может быть загружена? Подобные перебои возможны и при работе с компонентом HTTPService.

При возникновении проблем на сервере пользователь сталкивается с так называемой *ошибкой*. Она может быть вызвана отсутствием на сервере требуемой информации или его неисправностью. При этом значение свойства `lastResult` не сможет быть обновлено. По умолчанию в таких случаях Flex выведет всплывающее сообщение об ошибке. Если вы хотите изменить такое поведение по умолчанию, можно воспользоваться обработчиком события `fault`, которое происходит при возникновении этой ошибки.

Естественно, этот метод можно применять и для подключения удаленных данных, расположенных в сети Интернет. К примеру, можно использовать информацию из XML-файла, находящегося на удаленном сервере по адресу *http://greenlike.com/flex/learning/projects/contactmanager/contacts.xml* вместо локального файла *contacts.xml*. Для этого достаточно указать соответствующий путь в качестве значения свойства `url`:

```
<mx:HTTPService id="contactsService"
  resultFormat="e4x"
  url="http://greenlike.com/flex/learning/projects/contactmanager/
  contacts.xml" />
```

Примечание

Данные, расположенные удаленно, могут быть недоступны для Flex-приложения при отсутствии специального разрешения в файле, определяющем настройки веб-сайта. Подробнее об этом во врезке «Игры в песочнице».

Теперь приложение загружает необходимый XML-файл не с вашего локального компьютера, а через Интернет. При этом возможно небольшое замедление в работе программы, поскольку размер загружаемого файла больше, и вдобавок ему предстоит пройти некоторый путь по сети. В течение этого времени компоненты сервиса Flex выводят индикатор состояния загрузки в виде вращающегося колеса. Если такая настройка по умолчанию вам не нравится, ее можно без труда изменить, присвоив компоненту свойство `busyCursor` со значением `false`.

Примечание

При использовании абсолютных URL-адресов необходимо указать на использование HTTP-протокола. Не забывайте вводить *http://*.

Теперь вы умеете подключать к приложению данные в формате XML, расположенные удаленно, и приложение `ContactManager` содержит полную контактную информацию, предоставленную внешним источником. А теперь вернемся к созданию интерфейса нашего приложения.

Игры в песочнице

Когда речь заходит о Flash или компьютерной безопасности в целом, вы часто будете сталкиваться с термином **«песочница»**. Он употребляется по отношению к некоему безопасному пространству для запуска и использования программы (такой, как Flash Player, к примеру). Данное пространство ограничивает работу программы, а также доступ к нему других программ. Это напоминает ситуацию, когда родители приводят ребенка в парк, где он может свободно играть в песочнице, и мама с папой могут быть уверены в безопасности своего чада.

В отношении Flash Player при использовании «очень безопасной» песочницы для Flash-приложений, запускаемых через броузер, доступ к файловой системе пользователя или к информации, расположенной на удаленном веб-сайте, и манипулированию ей может быть полностью запрещен. (Стоит отметить, что ограничивающие рамки песочницы для AIR-приложений не столь жесткие). Первое и основное ограничение, налагаемое песочницей, относится к обращению с удаленной информацией.

Иными словами, интернет-приложению, созданному во Flex, не будут доступны данные, находящиеся вне домена, в котором используется приложение, за исключением случаев специального разрешения на это владельца сайта. Такое разрешение можно дать с помощью XML-файла с именем *crossdomain.xml*, расположенного в корне сайта.

В данном файле владелец сайта может указать определенные домены, для которых доступ к файлам будет разрешен, или установить возможность доступа для всех. Чтобы вы могли обращаться к файлу contacts.xml, расположенному на моем сайте, мне пришлось создать такой междоменный файл, обеспечивающий неограниченный доступ к моим ресурсам. Его можно посмотреть по адресу *http://greenlike.com/crossdomain.xml*. Выглядит он примерно следующим образом:

```
<?xml version="1.0"?>
<!DOCTYPE cross-domain-policy SYSTEM http://www.adobe.com/xml/
                                  dtds/cross-domain-policy.dtd">
<cross-domain-policy>
  <site-control permitted-cross-domain-policies="all"/>
  <allow-access-from domain="*" />
  <allow-http-request-headers-from domain="*" headers="*"/>
</cross-domain-policy>
```

Если бы данный файл отсутствовал или в нем не был бы разрешен неограниченный доступ к ресурсам сайта, попытка загрузить файл *contacts.xml* через Flex-приложение привела бы к возникновению ошибки и данные было бы невозможно получить.

Выбор пункта из списка

List, как и другие элементы списка, не только отображает перечень элементов, но и позволяет выбирать определенные пункты. С его свойством `selectedItem`, хранящим информацию о выбранном элементе списка, можно осуществлять связывание данных.

Рассмотрим это на примере нашего приложения `ContactManager`. Уберем текущее содержимое его раздела Contact Details, вместо этого разместим внутри него следующий код:

```
<mx:Panel id="contactDetails"
  layout="vertical"
  x="400"
  y="10"
  paddingLeft="5"
  paddingRight="5"
  paddingTop="5"
  paddingBottom="5"
  width="400"
  title="Contact Details">

  <mx:HBox>
    <mx:Label id="nameLabel"
    text="{contactsDataGrid.selectedItem.firstName}
          {contactsDataGrid.selectedItem.lastName}"
    fontWeight="bold"
    fontSize="14"/>

  <mx:Label id="emailLabel"
    text="Email: {contactsDataGrid.selectedItem.email}"/>

  <mx:Label id="phoneLabel"
    text="{contactsDataGrid.selectedItem.phoneType}:
{phoneFormatter.format(contactsDataGrid.selectedItem.phone)}" />
```

```
    <mx:Text id="addressText"
      text="Address: {contactsDataGrid.selectedItem.address}" />

    <mx:Label id="zipLabel"
      text="Zip Code: {contactsDataGrid.selectedItem.zip}" />

    <mx:Canvas
      width="60"
      height="60"
      backgroundColor="{contactsDataGrid.selectedItem.color}" />

</mx:Panel>
```

В данном коде осуществлена привязка нескольких элементов Label к свойству `selectedItem` таблицы DataGrid. (Ранее осуществлялась привязка полей к элементам управления для ввода данных, но теперь мы нацелены на создание режима отображения и режима редактирования в нашем приложении, поэтому пора их отделить). `selectedItem` – это конкретный элемент, выбранный из списка, расположенного в таблице DataGrid. В данном случае его значением является узел `<contact/>`, включающий свойства `firstName`, `lastName`, `phone`, `phoneType` и т. д. В нашем случае привязка элементов Label осуществляется именно к конкретным свойствам данного элемента (рис. 10.8).

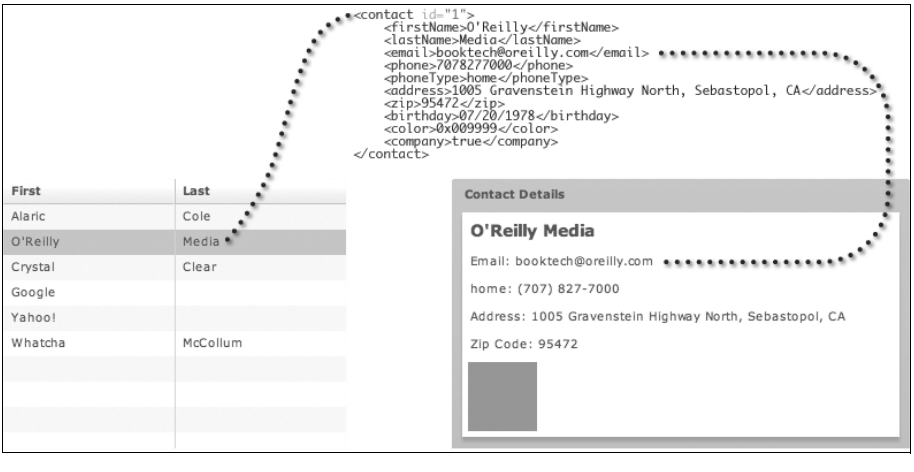

Рис. 10.8. *Использование свойства selectedItem*

Обратите внимание на использование элемента Text, обладающего свойством `text` и изменяющего свой размер в зависимости от величины его содержимого. Он подобен элементу Label, но содержащийся в нем текст может занимать несколько строк. Такой элемент оптимально использовать при необходимости вывести на экран небольшой фрагмент текста без полос прокрутки. Если объем текста велик и могут потребоваться полосы прокрутки, лучше воспользоваться элементом TextArea.

Примечание

При установке значения `true` свойства `allowMultipleSelection`, которым обладают все списки, появляется возможность выбора сразу нескольких пунктов. Для осуществления доступа к ним используется свойство `selectedItems`, значением которого будет массив выбранных элементов.

Подключение результатов поиска

Великолепным примером загрузки удаленных данных может послужить подключение к приложению результатов поиска, что существенно обогащает возможности приложения. В качестве примера создадим новый проект с названием Search (о том, как создавать новые проекты, подробно рассказано в главе 2).

Google предлагает общедоступный API, к которому могут обращаться Flash- и Flex-приложения, но для его использования разработчику необходимо уметь анализировать возвращаемые им данные в формате XML. Я создал простой компонент для доступа к поисковому сервису Yahoo!, позволяющий без труда размещать обработанные данные во Flex-приложении. Для создания описываемого ниже примера вам понадобится ActionScript 3 Search API. Откройте страницу *http://developer.yahoo.com/flash/astra-webapis/* и загрузите библиотекуYahoo! ASTRA Web APIs.

Распакуйте загруженный zip-архив, и вы увидите несколько вполне стандартных для Flex или ActionScript библиотек папок с названиями Build, Documentation, Examples и Source. Исходный код расположен в папке Source, но на данный момент нас интересует SWC-файл с названием AstraWenAPIs.swc, находящийся в папке Build.

Просто добавьте этот файл в папку libs проекта `Search`, и в вашем приложении сразу станет доступен компонент SearchService.

Теперь перейдем к вопросам использования данного компонента. Прежде всего необходимо вспомнить о понятии пространства имен. *Пространство имен* во Flex помогает различать группы взаимосвязанных компонентов. Безусловно, используемый нами компонент SearchService не входит в состав оболочки Flex по умолчанию и, соответственно, не принадлежит к пространству имен `mx`. Он является частью пространства имен `yahoo`. Как вы скорее всего помните, при использовании функции автозаполнения в режиме Source вводить название пространства имен не обязательно, поэтому для создания компонента `SearchService` достаточно начать вводить `<SearchService`, и оно будет вставлено автоматически. (Если вы не пользуетесь данной функцией, пространство имен можно ввести и вручную, как будет показано в следующем примере кода.)

Файлы SWC

SWC-файл (как правило, произносится «свик») – это компонент в скомпилированной форме. Удобство использования таких файлов можно подтвердить многочисленными доводами. Во-первых, отдельный файл распространять легче, чем исходный код. Во-вторых, если вы не хотите открыто распространять код созданных вами компонентов, использование формата SWC в некоторой степени гарантирует защиту вашего труда, поскольку для получения исходного кода такого компонента придется использовать инструменты декомпиляции, а это весьма затруднительно. В третьих, поскольку данные компоненты уже скомпилированы, процесс компиляции всего вашего приложения займет меньше времени.

Даже используемая вами оболочка Flex целиком содержится в нескольких отдельных SWC-файлах.

Для правильной работы компонента SearchService необходимо задать ему всего лишь одно свойство – текстовую строку, определяющую критерий поиска, – с помощью атрибута `query`. Значение данного свойства можно привязать к элементу TextIntput (давайте присвоим ему идентификатор `queryTextInput`). Пусть при нажатии на кнопку будет вызван метод `send()` компонента `SearchService`.

Компоненты SearchService и HTTPService не взаимосвязаны, однако их внутренний механизм обладает рядом схожих черт. Компонент SearchService использует метод `send()` и свойство `lastResult`, что позволяет Flex-разработчику легко начать им пользоваться.

При построении простого приложения Search можно использовать следующий код:

```
<mx:Application
  layout="absolute"
  xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:yahoo="http://www.yahoo.com/astra/2006/mxml">

  <yahoo:SearchService id="searchService"
                       applicationId="YahooDemo"
                       query="{queryTextInput.text}"/>

  <mx:HBox
    defaultButton="{searchButton}"
    y="10"
    x="10">

    <mx:FormItem
      label="Query:">
```

```
        <mx:TextInput id="queryTextInput"/>
      </mx:FormItem>

      <mx:Button id="searchButton"
        label="Search"
        click="searchService.send()"/>
    </mx:HBox>

    <!-Список результатов -->
    <mx:List id="resultsList"
      dataProvider="{searchService.lastResult}"
      left="10"
      right="10"
      top="75"
      bottom="10"
      showDataTips="true"
      labelField="name"/>

</mx:Application>
```

Вы, наверное, заметили, что в данном коде контейнер FormItem используется вне элемента Form. Лично я предпочитаю использовать этот контейнер для присваивания меток полям формы, но с тем же успехом можно воспользоваться и сочетанием элементов Label и HBox.

Обратите внимание на ранее не использовавшееся нами свойство `defaultButton` контейнера HBox (которое присуще контейнерам в целом). Оно указывает на кнопку, для которой произойдет событие `click` при нажатии пользователем на Enter (Return на Mac). Большинство пользователей привыкло к тому, что для отправки заполненных данных формы можно воспользоваться клавишей Enter (или Return), и потому наличие такого свойства как нельзя кстати. Кроме того, оно позволяет визуально выделить кнопку отправки.

Итак, после запуска приложения можно ввести запрос в окне поиска и нажать на кнопку (или клавишу Enter). Если все работает нормально, в приложении отобразится список данных, полученных в ответ на обработку запроса поисковым сервисом Yahoo!. По умолчанию данные выглядят как перечень заголовков страниц (этот способ можно слегка изменить; сейчас я покажу вам как). Запустите приложение (рис. 10.9). Некоторые из пунктов списка были обрезаны из-за излишней длины; наведите курсор мыши на любой из них, и появится всплывающая подсказка, отображающая текст целиком. Во Flex она называется *dataTip* и появляется при условии присваивания значения true свойству `showDataTips` списка.

Ну как вам, впечатляюще? Только что мы создали очень мощное приложение, использующее поисковый сервис Yahoo!. А теперь мы увидим, какие существуют дальнейшие возможности его совершенствования.

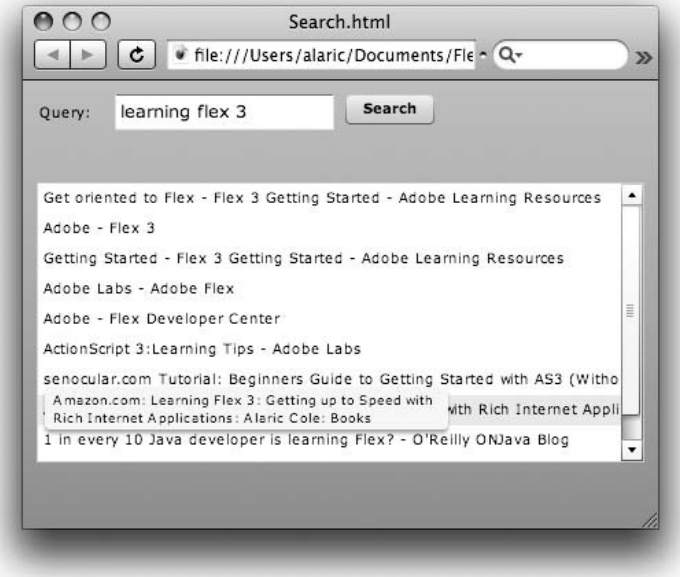

Рис. 10.9. *Простое, но очень функциональное приложение, осуществляющее поиск данных*

Перемещение элементов списка

Элементы списка Flex имеют встроенную поддержку перемещения расположенных внутри них данных. Чтобы перетащить пункт из одного списка в другой, не нужно мучиться и писать тонны дополнительного кода. Такая возможность является неотъемлемой частью элементов списка и реализуется с помощью свойств `dragEnabled` и `dropEnabled`.

Для примера добавим в наше приложение еще один элемент управления списком и зададим ему такое же значение свойства `labelField` – `resultList`. Затем присвоим его свойству `dropEnabled` значение `true`. Это позволит перемещать элементы в этот список из других списков. Далее установим значение `true` свойства `dragEnabled`. Теперь пользователи смогут перемещать элементы данного списка.

После запуска приложения можно просто взять и перетащить понравившийся вам результат поиска в соседний список! В следующем коде показано, как задать свойство `dragEnabled` списка `resultsList`, а также добавить еще одну панель и список, в который можно перемещать элементы. На рис. 10.10 изображен примерный процесс работы с таким приложением.

```
<mx:List id="resultsList"
  dataProvider="{searchService.lastResult}"
```

```
    left="10"
    right="318"
    top="74"
    bottom="47"
    showDataTips="true"
    labelField="name"
    dragEnabled="true" />
  <mx:Panel
    layout="vertical"
    title="Favorites"
    right="10"
    top="74"
    bottom="47"
    width="300">
    <mx:List
      dropEnabled="true"
      labelField="name"
      width="100%"
      height="100%"/>
  </mx:Panel>
```

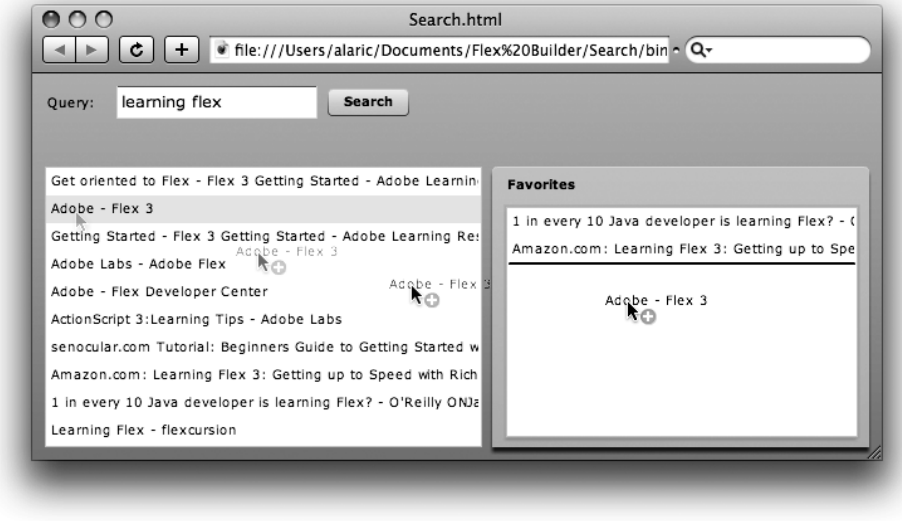

Рис. 10.10. *Перемещение элемента из одного списка в другой*

Для чего необходимо задавать свойство `labelField` новому элементу управления списком (в который можно перетаскивать пункты)? На самом деле в процессе перемещения соответствующий пункт копируется из источника информации одного списка в источник информации другого. Если в новом списке не задано свойство `labelField`, то в качестве метки было бы отображено `[object Object]`, т. к. не указано, что именно должно быть выведено.

Другие списки, такие как TileList или DataGrid, также позволяют перемещение пунктов. К примеру, вместо только что созданного нового списка можно было бы использовать DataGrid, причем при перемещении объекта в таблицу все его свойства были бы отображены в колонке.

Примечание

По умолчанию при перемещении элементов из одного списка в другой осуществляется их копирование. Но существует и свойство `dragMoveEnabled`, позволяющее именно переносить элементы, а не просто копировать их.

Использование встроенных представлений элементов

Внешний вид элементов списка можно легко настроить по своему вкусу. По умолчанию данный компонент создает метку для каждого элемента списка и, как вы уже знаете, отображает свойство, определяемое значением `labelField`. Создаваемая элементом списка метка называется `itemRenderer`, а вы легко можете создать свой собственный вариант.

Рассмотрим следующий код, в котором создается специальный объект `itemRenderer` для списка в нашем приложении `Search`:

```
<mx:List id="resultsList"
  dataProvider="{searchService.lastResult}"
  left="10"
  right="10"
  top="74"
  bottom="10">

  <mx:itemRenderer>
    <mx:Component>
      <mx:Label
        text="{data.name}"
        fontWeight="bold"/>
    </mx:Component>
  </mx:itemRenderer>

</mx:List>
```

В данном случае вместо стандартного использования свойства `labelField` мы попробуем видоизменить отображаемый список. Для этого необходимо более детально рассмотреть некоторые особенности данного кода. Прежде всего обратите внимание на тег `<mx:Component/>`, внутри которого заключено MXML-содержимое нашего `itemRenderer`; его использование указывает Flex на необходимость создать компонент и в дальнейшем применять его для каждого пункта списка. Это очень полезная функция, позволяющая разработчику создавать компоненты прямо внутри свойства другого компонента. К тому же с ее помощью можно создавать представление элементов, не прибегая к обязательно-

му созданию собственных компонентов. (Однако в создании собственных компонентов есть множество преимуществ, таких как возможность их многократного использования.).

Внутри тега <mx:Component/> находится собственно код, создающий itemRenderer. Обратите внимание, что его содержимым является один-единственный элемент Label. Он создаст метку для каждого пункта в списке данных, предоставляемых источником данных для элемента List.

Примечание

Содержимое тега <mx:itemRenderer/> не отображается в режиме Design.

Вместо свойства labelField мы используем элемент Label и с помощью привязки к свойству data указываем на элемент для отображения. Данное свойство ссылается на определенный элемент данных, содержащихся в источнике dataProvider. Таким образом, присваивание значения name свойству labelFiled и использование выражения data.name дает одинаковый результат – отображение соответствующего свойства (name) в списке.

Поскольку в нашем случае список отображает информацию, полученную в результате обработки поискового запроса, у нас есть возможность показать дополнительные сведения, например, URL-адрес страницы или краткий обзор ее содержания. В последнем случае можно осуществить привязку к свойству data.summary. Однако поскольку его значением может быть довольно внушительный объем текста, вместо элемента Label целесообразнее использовать элемент Text и осуществить привязку его свойства text. Такие текстовые элементы прекрасно подходят для отображения блоков текста, занимающих более одной строки. Для совместного использования элементов Label и Text заключите их в контейнер Vbox, как показано ниже.

```
<mx:List id="resultsList"
  dataProvider="{searchService.lastResult}"
  left="10"
  right="10"
  top="74"
  bottom="10">

  <mx:itemRenderer>
    <mx:Component>
      <mx:VBox
        width="100%" >
        <mx:Label
          text="{data.name}"
          fontWeight="bold"/>
        <mx:Text
          width="100%"
          text="{data.summary}"/>
      </mx:VBox>
```

```
    </mx:Component>
  </mx:itemRenderer>

</mx:List>
```

Примечание

Данные результаты являются объектами класса WebSearchResult, включенного в библиотеку ASTRA *Web* APIs. Он обладает такими свойствами, как `name`, `summary` и `clickURL`. Более подробная информация о данном классе содержится в документации, поставляемой с библиотекой Yahoo! ASTRA Web APIs.

При желании списку можно также присвоить свойство `variableRowHeight`. Его значение `true` позволяет списку варьировать высоту рядов, чтобы уместить максимально возможное количество информации, поскольку длина отображаемых элементов может существенно различаться, а по умолчанию все ряды имеют одинаковый размер.

Посмотрите, как будет выглядеть ваше приложение с использованием настроенного таким образом `itemRenderer`.

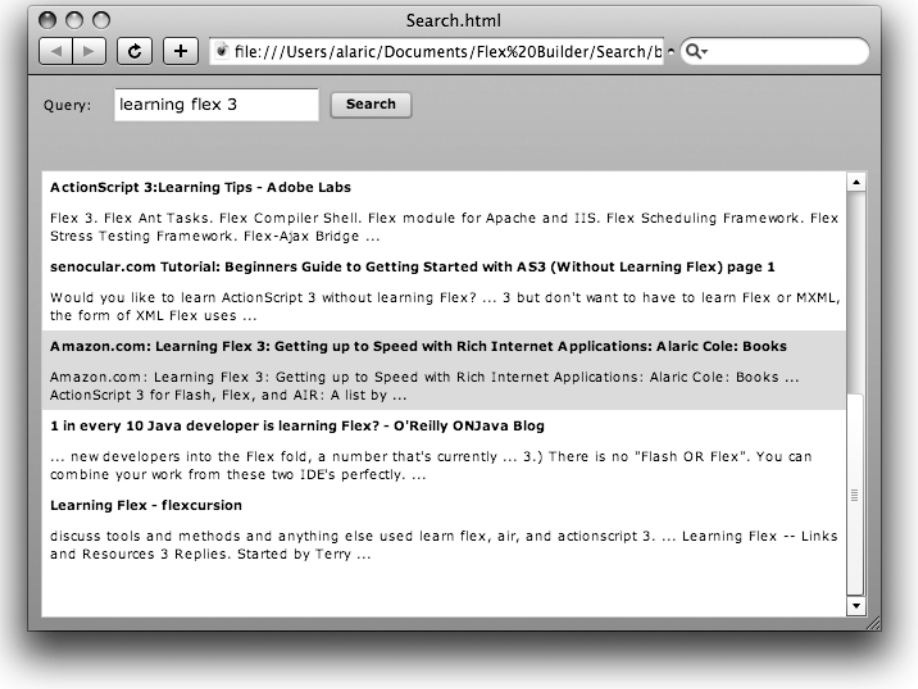

Рис. 10.11. *Приложение Search при использовании настроенного itemRenderer*

Знакомство с другими сервисными компонентами

Подробное рассмотрение такого рода сервисов выходит за рамки данной книги, но мне хотелось бы, чтобы у вас появилось представление о наличии тех или иных возможностей. Скорее всего, вам достаточно часто придется использовать компонент `HTTPService`, в отдельных случаях вы оцените преимущества компонента `RemoteObject` или же вам может потребоваться доступ к каким-либо веб-сервисам с помощью компонента `WebService`.

WebService

Компонент HTTPService позволяет получить доступ к сервисам, имеющим URL-адрес, т. е. данным, отображаемым в броузере, например в формате HTML или XML. Однако существует масса веб-сервисов, доступных только с помощью XML-стандарта SOAP. Как правило, их использование особенно актуально в приложениях для бизнеса, и несмотря на их «тяжеловесность», вам может потребоваться возможность подключения веб-сервиса, использующего протокол SOAP, к вашему Flex-приложению.

Внимание

Ввиду сложности использования веб-сервисов SOAP рекомендуется обращаться к ним только в случае крайней необходимости и при отсутствии альтернативных путей решения задачи.

Компонент `WebService` существенно упрощает доступ к веб-сервисам SOAP. Вам также наверняка пригодится новая команда, введенная во Flex Builder 3 – Import Web Services. Она позволяет сослаться на URL-адрес описания веб-сервиса (WDSL) и автоматически пишет для вас большую часть кода.

Примечание

WDSL расшифровывается как Web Services Description Language (язык описания веб-сервисов), и является стандартной моделью языка, описывающего веб-сервисы. Он определяет возможные операции, проводимые сервисом, и формат полученных в их результате данных.

Эта команда доступна через меню Data→Import Web Service (WSDL). При этом откроется диалоговое окно, в котором необходимо указать место для расположения генерируемого кода и URL-адрес описания веб-сервиса на WSDL. Flex Builder автоматически создаст необходимый код, облегчив вам задачу доступа к веб-сервису.

Примечание

Перечень доступных для использования веб-сервисов расположен по адресу *www.xmethods.net*.

RemoteObject

Мощнейшим компонентом для доступа к данным является `remoteObject`, обеспечивающий простоту соединения с серверами, использующими технологии Java (Java EE или J2EE) или ColdFusion. Использование *удаленных объектов* дает разработчику массу преимуществ, таких как установление соответствия объектов Java и ActionScript. Это означает, что такие типы данных, как Number и Date, будут переданы без искажения. Еще одно преимущество удаленных объектов заключается в компрессии (сжатии) данных, что ускоряет доступ к данным больших объемов. Среди прочего, вы можете использовать метод «*проталкивания*» данных (data push), позволяющий отслеживать изменения данных на удаленном сервере и получать обновления автоматически. (В этом случае можно провести аналогию с почтовым клиентом, отображающим новые сообщения электронной почты сразу по их получении, не требуя от вас запроса на доставку почты.)

Создание приложения из базы данных

Если на вашем сервере используется технология ColdFusion, PHP, J2EE или ASP.NET, то вам очень повезло, поскольку Flex Builder автоматически сгенерирует для вас необходимый серверный код, что упростит процесс подключения к базе данных и даже позволит изменять содержащиеся в ней данные. Достаточно лишь предоставить саму базу данных, а весь необходимый код для создания, чтения, обновления и удаления записей будет написан автоматически, включая даже код пользовательского интерфейса!

Для приложений PHP, J2EE или ASP.NET необходимы собственно база данных и сервер, настроенный для использования одной из перечисленных технологий. Вам достаточно лишь выбрать пункт меню Data→Create Application from Database и следовать инструкциям в открывающихся диалоговых окнах. При этом будет создано Flex-приложение с необходимыми компонентами пользовательского интерфейса и вспомогательными методами для доступа к данным. Его можно использовать как основу для создания своего собственного приложения.

Для разработки приложений, использующих технологию ColdFusion, потребуется установить ColdFusion-сервер и расширения ColdFusion для Flex Builder. Их можно добавить прямо при установке Flex Builder или в любое время позже. Расположенная по адресу *http://livedocs.adobe.com/coldfusion/8/htmldocs/othertechnologies_11.html* статья содержит инструкции по установке упомянутых расширений.

Примечание

Java EE означает Java Enterprise Edition. Это версия Java-платформы, предназначенная для обработки данных на сервере. ColdFusion является приложением Java EE, позволяющим создавать серверный код на языке разметки.

Чтобы использовать компонент RemoteObject, необходимо предварительно установить соответствующие настройки вашего приложения, позволяющие осуществлять доступ к удаленным объектам, расположенным на определенных серверах. Как правило, эти настройки устанавливаются при создании нового Flex-проекта. В нижней части диалогового окна New Flex Project (рис. 10.12) последний раздел называется ServerTechnology. Здесь можно выбрать тип сервера – ColdFusion или J2EE. Убедитесь, что флажок «Use remote object access service» установлен.

Рис. 10.12. *Настройка нового проекта Flex для работы с ColdFusion Flash Remoting*

При использовании технологии ColdFusion вы имеете возможность выбора между сервисами Flash Remoting и LifeCycle DataServices, а в случае с приложениями J2EE можно использовать только последний.

Для использования в полной мере функций данного компонента требуется также настроить сервер, использующий одну из перечисленных технологий. Это требует более детального объяснения, чем может быть дано в этой книге. Рекомендую вам обратиться за дополнительной информацией к статье, расположенной по адресу *http://livedocs.adobe.com/flex/3/html/data_intro_2.html*.

Заключение

Итак, вы поднялись еще на несколько ступеней к вершинам создания насыщенных интернет-приложений. В этой главе вы научились не только использовать данные, расположенные прямо в вашем MXML-коде, но и осуществлять доступ к информации из внешних источников в сети Интернет. Теперь вам известны основы работы с XML-данными, а также принципы использования элементов для отображения информации в виде списка. Вы на практике освоили способы применения компонента DataGrid, обладающего широкими возможностями.

Кроме того, вы смогли воочию убедиться, что компоненты, созданные сторонними разработчиками, нередко можно с успехом применять в качестве дополнения к стандартным Flex-компонентам, а разнообразие таких компонентов еще больше расширяет ваши возможности как разработчика. Вы создали несложное приложение с использованием поискового сервиса Yahoo!, встроенных представлений элементов itemRenderer и нового элемента Text для полноценного отображения результатов поиска.

Теперь вы можете без труда использовать неограниченное количество информации из сети Интернет в своих приложениях. А представленные в следующих главах приемы позволят отшлифовать до блеска внешний вид ваших приложений.

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.

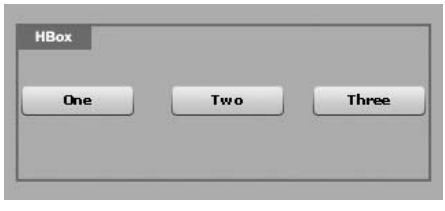

Рис. 11.1. *Три кнопки внутри контейнера HBox, все видимы*

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)*

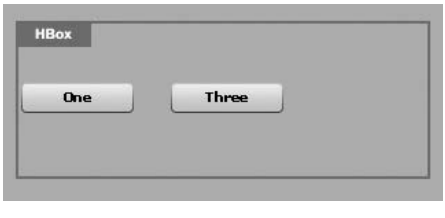

Рис. 11.4. *Три кнопки внутри контейнера HBox; для второй кнопки установлены свойства includeInLayout="false", visible="false" (третья кнопка занимает ее место)*

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.

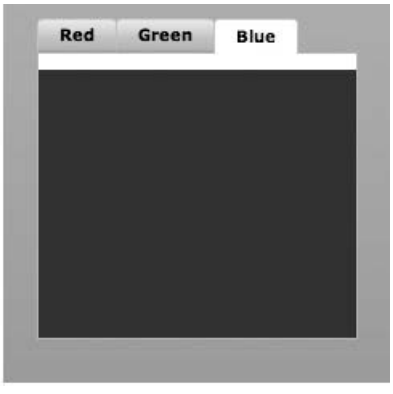

Рис. 11.5. *Три контейнера Canvas внутри TabNavigator; переключение между ними реализовано с помощью вкладок*

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

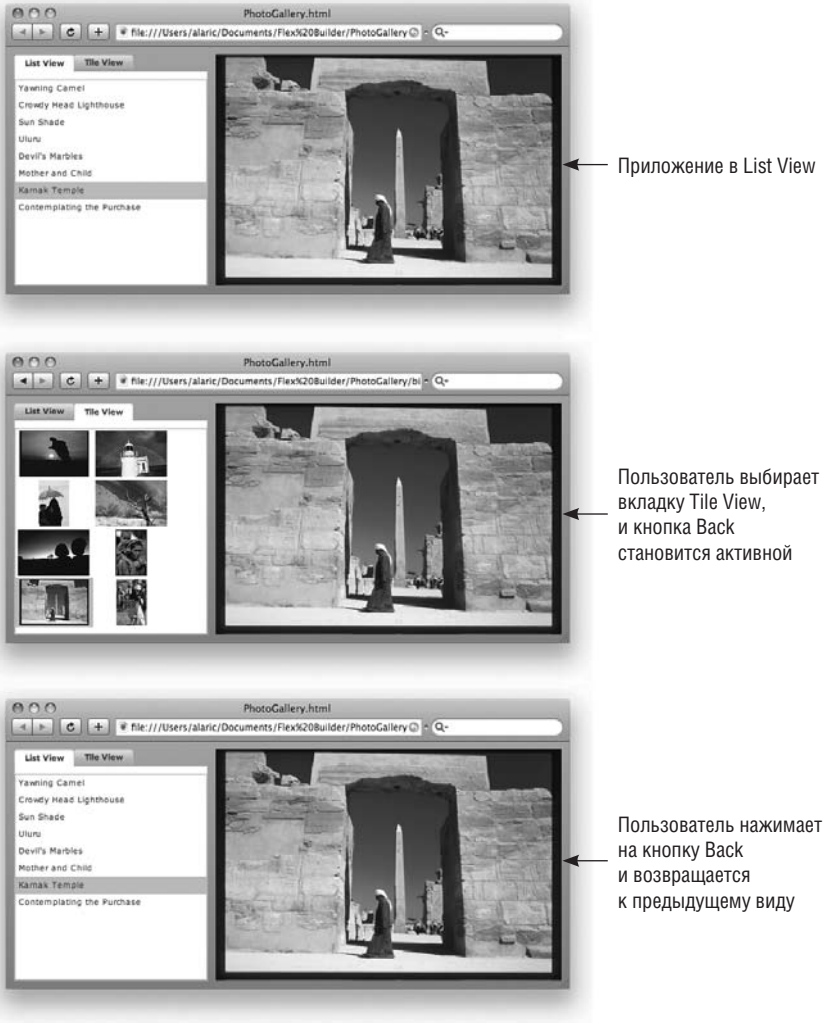

***Рис. 11.8.** Изменение текущего вида с помощью кнопки Back*

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.

В этой главе:

- Типичные случаи использования состояний
- Создание нового состояния
- Изменение свойств, стилей и событий состояния
- Добавление компонентов
- Использование состояний на практике

12

Состояния приложения

Отличным инструментом для построения динамического и гибкого пользовательского интерфейса являются состояния приложения. Благодаря различным состояниям интерфейс может выглядеть по-разному в конкретный момент времени и в зависимости от определенных целей. Представьте себе веб-приложение, включающее в себя как страницу для авторизации пользователя, так и страницу настройки пользовательских предпочтений – эти страницы можно считать HTML-эквивалентами состояний в приложении Flex.

Состояние – это набор логически сгруппированных изменений пользовательского интерфейса. По сути, *состояние* – это совокупность изменений свойств, стилей и типов поведения компонента. Аналогичного эффекта – внесения ряда единовременных изменений в интерфейс приложения – можно добиться и с помощью набора функций, но состояния Flex, реализуемые с помощью MXML, гораздо проще создавать и модифицировать. А использование среды разработки Flex Builder еще больше упростит этот процесс.

Типичные случаи использования состояний

Вспомните созданное нами в главе 8 приложение для поиска по заданным критериям. При загрузке данного приложения пользователь увидит поле для ввода запроса и список результатов поиска. Это, конечно, замечательно, но немного странно показывать список результатов до их получения после отправки запроса.

Конечно, можно сделать список результатов невидимым, а при поступлении данных отобразить его, изменив настройку видимости. Но в идеа-

ле, конечно, хотелось бы внести больше изменений в данный интерфейс. Не будет ли лучше, если при запуске приложения будет отображаться только поле для ввода запроса, расположенное посередине крупным планом, так что оно сразу бросалось бы в глаза (рис. 12.1)? При отправке поискового запроса пользователем данное поле можно переместить в верхнюю часть приложения, чтобы освободить место для отображения результатов поиска (рис. 12.2). Это не только выглядит стильно и живо, но дает пользователю дополнительные удобства при работе с приложением, поскольку в каждый момент времени отображается именно то, что ему сейчас нужно.

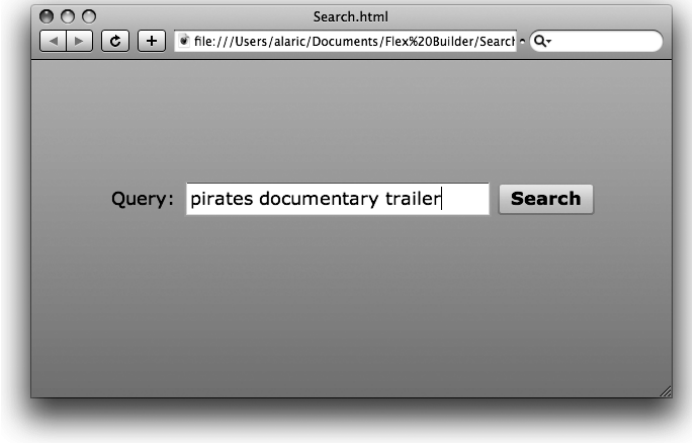

Рис. 12.1. *Первоначальное состояние приложения Search: отображается только поле для ввода запроса*

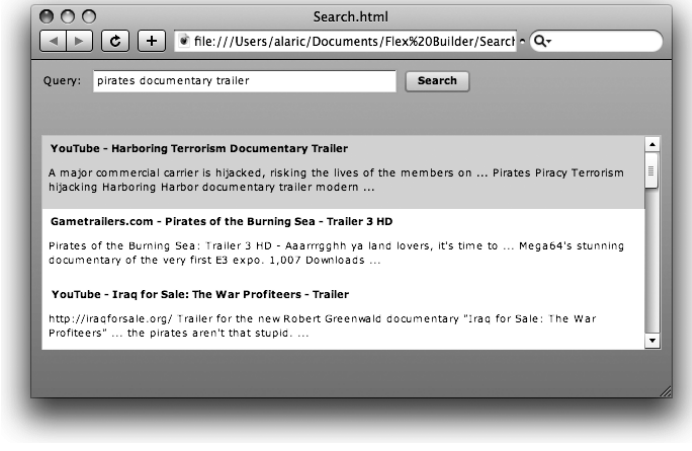

Рис. 12.2. *Дополнительное состояние приложения Search: поле для ввода запроса уменьшено и теперь отображается в самом верху, ниже расположены результаты поиска*

Изучить механизм создания состояний приложения проще всего прямо на практике, с использованием режима Design среды Flex Builder. Я покажу вам, как применять их широчайшие возможности в собственных приложениях. Если вы хотите попробовать свои силы по ходу прочтения данной главы, создайте новый проект. Мы построим новое приложение с кнопкой, расположение которой будет меняться при переключении между двумя состояниями.

Состояния или контейнеры навигации?

В большинстве случаев использование состояний дает разработчику большую гибкость при создании интерфейса приложения в сравнении с применением контейнеров навигации. Это объясняется возможностью использования одних и тех же компонентов в разных состояниях. Например, кнопка, существующая в одном состоянии, может оставаться и в другом состоянии (возможно, с измененными свойствами). Между прочим, одна и та же кнопка может даже появляться в разных частях приложения, к примеру, перемещаясь из контейнера Panel в контейнер Form. Для достижения подобного эффекта с помощью контейнера навигации для отображения кнопки при переключении в другой вид вам придется создать новую кнопку.

Состояния можно создавать, основывая их друг на друге, что невозможно при использовании контейнеров навигации. Благодаря такому способу разработчик может создавать каскадный набор взаимозависимых видов. Элемент ViewStack и другие контейнеры навигации всегда независимы.

Однако вышесказанное не означает, что использование состояний подходит для решения любой задачи. Вы уже могли убедиться, что контейнеры ViewStack, TabNavigator и другие элементы навигации с успехом можно использовать во многих случаях – например, когда приложение состоит из нескольких отдельных независимых частей. Кроме того, элементы навигации и механизм вкладок контейнера TabNavigator позволяют пользователям без труда переключать виды; создание такого удобного инструмента навигации с помощью состояний более затруднительно.

Создание нового состояния

Прежде всего, поместите кнопку на сцене ближе к левому ее краю. Для создания нового состояния мы будем использовать панель States. Ее можно вывести на экран через меню Window→States. Открыв данную панель, вы увидите *основное* состояние, которое имеет любое прило-

жение. Новое состояние создается на его базе путем нажатия первой кнопки на панели инструментов States, как показано на рис. 12.3.

Рис. 12.3. *Создание нового состояния с помощью панели States*

При этом откроется диалоговое окно, запрашивающее параметры нового состояния (рис. 12.4). Состоянию можно дать любое имя, но, как всегда, я рекомендую придумать «говорящее» название. В нашем случае давайте назовем его stageRight, поскольку его единственной функцией будет перемещение кнопки в правую часть приложения.

Рис. 12.4. *Диалоговое окно New State*

Изменение свойств, стилей и событий состояния

После создания нового состояния оно будет активировано на панели States. Это означает, что производимые вами изменения в режиме Design будут применены именно к этому состоянию. В нашем примере достаточно всего лишь перетащить кнопку к правому краю сцены. А теперь переключитесь между двумя разными состояниями (основным состоянием и только что созданным состоянием stageRight) с помощью панели States. Вы увидите, что в первоначальном состоянии кнопка находится в левой части приложения, но при переходе в состояние stageRight

она перемещается к его правому краю. Переход между состояниями можно осуществлять и с помощью выпадающего списка, расположенного на панели инструментов в режиме Design (рис. 12.5).

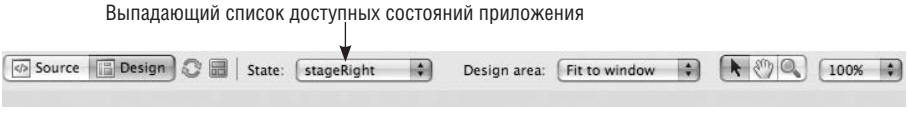

Рис. 12.5. *Выпадающий список состояний в режиме Design*

При выполнении вышеописанных операций в режиме Design генерируется необходимый MXML-код. Любое приложение Flex (а также, в чем вы вскоре убедитесь, любой компонент) обладает свойством states. Оно содержит список его возможных состояний, которые могут быть определены с помощью тега <mx:State/>. У данного тега есть свойство name, соответствующее названию состояния. Он может включать различные вложенные теги, соотносящиеся с различными видами изменений. В нашем примере с созданием двух состояний приложения сгенерированный код может выглядеть примерно следующим образом:

```
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                      layout="absolute">

  <mx:states>

    <mx:State name="stageRight">
      <mx:SetProperty
        target="{button1}"
        name="x"
        value="570"/>
    </mx:State>

  </mx:states>

  <mx:Button id="button1"
    x="10"
    y="10"
    label="Click to move"
    click="currentState='stageRight'"/>

</mx:Application>
```

Обратите внимание на стандартный компонент Button в нижней части кода – его можно назвать компонентом в основном состоянии. Между открывающим и закрывающим тегами <mx:states> и </mx:states> расположен единственный тег с именем stageRight. Его содержимое описывает изменения, которые произойдут при выборе этого состояния. В нашем примере изменения коснутся свойства x элемента Button, которое должно принять значение 570 вместо 10. Процесс изменения осуществляется с помощью тега <mx:setProperty/>, а точнее, трех его

свойств. Свойство target принимает имя id модифицируемого компонента, name указывает на имя изменяемого свойства, а value отвечает за его новое значение в данном состоянии.

Для переключения состояний используется свойство приложения currentState. В нашем примере переход к состоянию stageRight произойдет при нажатии на кнопку.

Примечание

По умолчанию значением свойства currentState является пустая строка (??), поэтому для переключения в основное состояния можно использовать выражение currentState=??.

Безусловно, при переключении состояний можно реализовать гораздо более сложную модификацию пользовательского интерфейса, чем простое изменение свойств компонента. Можно также изменять стили и даже обработчики событий компонентов. Вспомните пример с тремя разноцветными контейнерами из главы 8. Следующий код создает контейнер Canvas с красным цветом фона по умолчанию. Затем создаются два состояния, названные green и blue, изменяющие фон Canvas на зеленый и синий, соответственно, с помощью свойства backgroundColor:

```
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                      layout="absolute">

  <mx:states>

    <mx:State name="green">
      <mx:SetStyle
        target="{canvas}"
        name="backgroundColor"
        value="#00FF00"/>

    </mx:State>

    <mx:State name="blue">
      <mx:SetStyle
        target="{canvas}"
        name="backgroundColor"
        value="#0000FF"/>
    </mx:State>

  </mx:states>

  <mx:Canvas id="canvas"
    width="200"
    height="200"
    backgroundColor="#FF0000"/>

</mx:Application>
```

Поскольку категория стилей отличается от обычных свойств, их изменение требует использования тега <mx:SetStyle/>, а не <mx:SetProperty/>, в остальном же механизм их изменений ничем не отличается.

Для установки обработчиков событий существует специальный тег, названный, как нетрудно догадаться, <mx:SetEventHandler/>. Принцип его действия аналогичен перечисленным выше тегам, за тем лишь исключением, что вместо свойства value используется handler. В предыдущий пример, в котором происходило перемещение кнопки, можно добавить возможность возвращения к основному состоянию, изменив обработчик события click в состоянии stageRight:

```
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                      layout="absolute">

  <mx:states>
    <mx:State name="stageRight">

      <mx:SetProperty
        target="{button1}"
        name="x"
        value="570"/>

      <mx:SetEventHandler
        target="{button1}"
        name="click"
        handler=" currentState='' "/>

    </mx:State>
  </mx:states>

  <mx:Button id="button1"
    x="10"
    y="10"
    label="Click to move"
    click="currentState='stageRight'"/>

</mx:Application>
```

В этом коде возвращение к основному состоянию реализовано путем добавления второго дочернего тега в состоянии stageRight, устанавливающего обработчик события click для кнопки Button.

Примечание

При работе с состояниями в режиме Design вам, возможно, не придется писать ни строчки кода вручную. Благодаря наличию панели States вся рутинная работа выполняется автоматически. Однако понимание механизма создаваемого кода совершенно необходимо, если вам придется в дальнейшем заняться его доводкой.

Добавление компонентов

При использовании состояний можно также добавлять и удалять компоненты. Это означает возможность модифицирования целых частей вашего приложения с помощью несложного MXML-кода. Представьте себе обычно предлагаемые на веб-страницах формы для авторизации

пользователя. Как правило, в них запрашиваются имя пользователя и пароль, но если вы еще не регистрировались на данном сайте, придется перейти по указанной ссылке на страницу регистрации, где потребуется заполнить свое настоящее имя, логин и пароль. Согласитесь, было бы гораздо удобнее, если бы поля, необходимые для регистрации пользователя, появлялись при необходимости на той же странице. В следующем примере мы создадим такую функцию (рис. 12.6).

В начале кода создается панель Panel и вложенный в нее контейнер Form, содержащий поля для ввода имени пользователя и пароля. В этом для вас нет ничего нового. На рис. 12.6 показан результат выполнения данного кода.

```
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                      verticalAlign="middle">

  <mx:Panel id="loginPanel"
    title="Returning Users Sign In"
    horizontalAlign="right"
    paddingLeft="5"
    paddingRight="5"
    paddingTop="5"
    paddingBottom="5">

    <mx:Form id="loginForm">
      <mx:FormItem id="usernameFormItem"
        label="Username:" >
        <mx:TextInput/>
      </mx:FormItem>

      <mx:FormItem id="passwordFormItem"
        label="Password:" >
        <mx:TextInput
          displayAsPassword="true"/>
      </mx:FormItem>
      </mx:Form>

    <mx:Button id="submitButton"
      label="Sign in"/>

    <mx:ControlBar
      horizontalAlign="right">

      <!-- Элемент LinkButton осуществляет переход
      к другому состоянию при щелчке по нему мышью -->
      <mx:LinkButton id="registerLink"
      label="Don't have an account yet?"
      color="#1B337B"
      click="currentState='registration'" />

    </mx:ControlBar>

  </mx:Panel>

</mx:Application>
```

Рис. 12.6. *Форма для авторизации пользователя в первоначальном состоянии*

Затем можно создать состояние «регистрации», добавив в форму несколько полей и изменив некоторые свойства уже существующих, приспособив их для этого состояния. Это позволит использовать уже существующие компоненты с некоторыми «надстройками» для соответствующих состояний. В приложение можно добавить следующий код для создания нового состояния с дополнительными полями, как показано на рис. 12.7.

```
<mx:states>

  <mx:State name="registration">
    <!-- Установка свойств контейнера Panel -->
    <mx:SetProperty
      target="{loginPanel}"
      name="title"
      value="New User Registration"/>

    <!-- Установка свойств элемента Button -->
    <mx:SetProperty
      target="{submitButton}"
      name="label"
      value="Register"/>

    <!-- Изменение свойства label элемента LinkButton -->
    <mx:SetProperty
      target="{registerLink}"
      name="label"
      value="Already have an account with us?"/>

    <!-- Изменения для LinkButton, возвращающие
```

```
      к состоянию формы авторизации -->
      <mx:SetEventHandler
        target="{registerLink}"
        name="click"
        handler="currentState=''"/>

      <!-- Добавление в форму поля Full Name -->
      <mx:AddChild
        relativeTo="{loginForm}"
        position="firstChild">
        <mx:FormItem id="fullNameFormItem"
          label="Full Name:">
        <mx:TextInput/>
        </mx:FormItem>
      </mx:AddChild>

      <!-- Добавление в форму поля для подтверждения введенного пароля -->
      <mx:AddChild
        relativeTo="{loginForm}"
        position="lastChild">
        <mx:FormItem id="confirmPasswordFormItem"
          label="Confirm Password:">
        <mx:TextInput
          displayAsPassword="true"/>
        </mx:FormItem>
      </mx:AddChild>

    </mx:State>

  </mx:states>
```

Рис. 12.7. *Форма для авторизации пользователя в новом состоянии с двумя дополнительными полями, новым заголовком панели и текстом кнопки*

Примечание

Дочерний элемент, добавленный в одном состоянии, можно убрать при переходе в другое состояние. Для этого существует соответствующий тег <mx:RemoveChild/>, удаляющий дочерние элементы.

Первые четыре дочерних тега <mx:State/> в данном примере изменяют свойства приложения, но два последних тега <mx:AddChild/> выполняют иную функцию – они добавляют новые компоненты.

Внутри <mx:AddChild/> расположены дочерние MXML-теги, которые вы уже не раз использовали. Ему также присущ ряд свойств, определяющих положение нового компонента. Свойство relativeTo указывает на другой компонент, служащий в качестве точки отсчета, – в данном примере им является компонент loginForm, что определено с помощью фигурных скобок. Свойство position определяет местонахождение компонента относительно элемента, указанного в свойстве relativeTo.

Примечание

Вы наверняка уже оценили преимущества режима Design как инструмента для изменения свойств компонента, но при работе с несколькими состояниями приложения это просто находка. В данном режиме можно выбрать компонент в определенном состоянии, а при переключении в другое состояние тот же компонент останется выделен, но уже в соответствующем состоянии. Затем можно изменять свойства, стили или события данного компонента, используя панель Flex Properties, а Flex Builder автоматически напишет весь необходимый код. Вам остается лишь решить, какие именно изменения нужно внести.

В нашем примере значением свойства position элемента fullNameFormItem является firstChild. Это означает, что добавленный компонент станет первым по порядку дочерним элементом формы loginForm, т. е., хотя он и расположен в нижней части ее списка отображения, он будет создан первым из вложенных элементов. Элемент confirmPasswordFormItem, напротив, станет последним дочерним элементом контейнера loginForm, поскольку его свойство position принимает значение lastChild. (Свойству position также можно присваивать значения before и after, позволяющие определять местонахождение компонента относительно соседних компонентов, а не родительского элемента. Свойство before помещает его *перед* соответствующим элементом в списке отображения, а after – *после*.)

Примечание

В следующей главе мы рассмотрим способы создания различных переходов и эффектов, которые можно применять при изменении состояния приложения.

Использование состояний на практике

Теперь у вас должно сложиться представление о состояниях, и настало самое время использовать их в собственных проектах.

Приложение Search

Вернемся к приложению Search, созданному в главе 10. Откройте этот проект, чтобы внести некоторые изменения в отношении его интерфейса. Безусловно, рациональнее было бы заранее продумать, каким образом будет строиться интерфейс – с использованием различных состояний или нет – и изначально ориентироваться на те или иные средства, но тем не менее нам не придется полностью перестраивать приложение с нуля. Начальное состояние нашего приложения будет содержать только поле для введения запроса (т. е. на начальном этапе результаты не должны отображаться). Но основное состояние уже содержит результаты. Ничего страшного; я покажу вам, как добавлять и удалять компоненты, используя различные состояния.

Удаление компонентов

Откройте основной файл приложения (*Search.mxml*) в режиме Design и добавьте новое состояние с названием search. В некотором смысле мы будем работать в обратном направлении, поскольку состояние с результатами – это основное состояние вашего приложения, которое уже создано. Выберите состояние search и удалите элемент управления списком с именем resultsList. Не сомневайтесь, ведь он останется в основном состоянии в целости и невредимости – переключитесь в него, и сами в этом убедитесь. Его удаление в состоянии search освободит место, занимаемое пустым списком. Теперь в вашем приложении гораздо больше свободного пространства, и можно без труда переместить поля для ввода поискового запроса в самый центр. Это можно сделать с помощью ограничителей, задав свойства horizontalCenter и verticalCenter элемента HBox. Можно также изменить параметры шрифта, чтобы текст был более привлекательным и читаемым. Код, созданный автоматически режимом Design, будет выглядеть примерно следующим образом:

```
<mx:states>
  <mx:State name="search">
    <mx:RemoveChild
      target="{resultsList}"/>
    <mx:SetProperty
      target="{hbox1}"
      name="x"/>
    <mx:SetProperty
      target="{hbox1}"
      name="y"/>
    <mx:SetStyle
```

```
        target="{hbox1}"
        name="horizontalCenter"
        value="0"/>
      <mx:SetStyle
        target="{hbox1}"
        name="verticalCenter"
        value="-30"/>
      <mx:SetStyle
        target="{hbox1}"
        name="fontSize"
        value="18"/>
  </mx:State>
```

Обратите внимание на тег `<mx:RemoveChild/>`. Он предназначен для удаления указанного в свойстве `target` элемента (в данном случае – список с результатами поиска). В предыдущих примерах вам приходилось добавлять элементы, но в нашем случае в приложении имеются ненужные элементы, удалить которые очень просто именно с помощью этого тега. Также обратите внимание на свойства и стили, присвоенные элементу `hbox1`. На самом деле этим элементом является контейнер HBox, содержащий поле для поиска и кнопку для подтверждения запроса. Поскольку вы не задали идентификатор данного контейнера самостоятельно, Flex Builder присвоил ему произвольное имя для дальнейших ссылок. (Довольно безликое имя, согласитесь. Вот поэтому все-таки стоит самостоятельно задавать компонентам описательные имена.) Для центрирования HBox имеет смысл использовать стили `horizontalCenter` и `verticalCenter`. Вы наверняка заметили, что два первых тега `<mx:SetProperty/>` имеют свойства `target` и `name`, но свойство `value` отсутствует. Дело в том, что в данном случае они не присваивают свойству новое значение, а, напротив, удаляют данное свойство компонента. (В соответствующем коде на ActionScript свойствам было бы присвоено значение `null`.)

Примечание

Чтобы изменить идентификатор `hbox1` и все ссылки на него, имеющиеся в вашем коде, воспользуйтесь функцией Find and Replace. Выберите пункт меню Edit→Find/Replace во Flex Builder. При этом откроется диалоговое окно Find/Replace, позволяющее найти фрагмент текста и заменить его на определенное значение. Очень удобно, что если выделить фрагмент текста в коде перед переходом в это диалоговое окно, то он автоматически попадает в поле Find данного инструмента.

Установка начального состояния

Новое состояние создано, но при запуске приложения оно, конечно, отобразится в основном состоянии с прежней структурой, включающей и список результатов. Поэтому нужно изменить первоначальное состояние приложения на отличное от основного. Для этого дважды

щелкните по состоянию search на панели States, чтобы открыть диалоговое окно Edit State Properties, напоминающее уже знакомое вам окно New State. В нем вы увидите флажок Set as start state. При включении данного флажка при запуске приложения оно будет отображено в состоянии search. Этого можно добиться и с помощью атрибута currentState тега <mx:Application/>.

Нам остается добавить последний штрих – задать возможность возвращения приложения в основное состояние при получении результатов для отображения. Для этого можно установить обработчик события click для кнопки Button или, что еще лучше, обработчик события result компонента searchService.

```
<yahoo:SearchService id="searchService"
  query="{queryTextInput.text}"
  result="currentState = '' " />
```

Теперь при получении данных компонентом searchService происходит событие result, возвращающее приложение в основное состояние. Полный код приложения Search представлен ниже, а его внешний вид показан на рис. 12.1 и 12.2.

```
<mx:Application
  xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:yahoo="http://www.yahoo.com/astra/2006/mxml"
  layout="absolute"
  currentState="search">

  <mx:states>
    <mx:State name="search">
      <mx:RemoveChild
        target="{resultsList}"/>
      <mx:SetProperty
        target="{hbox1}"
        name="x"/>
      <mx:SetProperty
        target="{hbox1}"
        name="y"/>
      <mx:SetStyle
        target="{hbox1}"
        name="horizontalCenter"
        value="0"/>
      <mx:SetStyle
        target="{hbox1}"
        name="verticalCenter"
        value="-30"/>
      <mx:SetStyle
        target="{hbox1}"
        name="fontSize"
        value="18"/>
      </mx:State>
    </mx:states>
```

```
    <yahoo:SearchService id="searchService"
      query="{queryTextInput.text}"
      result="currentState = '' " />

    <mx:HBox id="hbox1"
      verticalAlign="middle"
      defaultButton="{searchButton}"
      y="10"
      x="10" >

      <mx:FormItem
        label="Query:">
        <mx:TextInput id="queryTextInput"/>
      </mx:FormItem>

      <mx:Button id="searchButton"
        label="Search"
        click="searchService.send()"/>

    </mx:HBox>

    <!-Результаты, представленные в виде списка List -->
    <mx:List id="resultsList"
      dataProvider="{searchService.lastResult}"
      left="10"
      right="10"
      top="74"
      bottom="10"
      variableRowHeight="true">

      <mx:itemRenderer>
        <mx:Component>
          <mx:VBox
            width="100%" >
            <mx:Label
            text="{data.name}"
            fontWeight="bold"/>
          <mx:Text
            width="100%"
            text="{data.summary}"/>
        </mx:VBox>
      </mx:Component>
    </mx:itemRenderer>

  </mx:List>
</mx:Application>
```

Примечание

При планировании приложения состояния могут весьма эффективно использоваться для реализации поэтапного сценария выполнения приложения. Так как состояния могут быть каскадными (т. е. одно состояние основано на другом), их можно создавать в любое время, даже в самом начале разработки. В дальнейшем изменения родительского состоянии автоматически распространяются на дочерние состояния.

Приложение ContactManager

Убедиться на практике во всемогуществе состояний нам поможет приложение ContactManager, созданное в главе 10. На данный момент оно содержит список контактов в двух видах – видом только для чтения данных (панель Contact Details) и для их редактирования (панель Contact Editor). Такое положение вещей вполне можно усовершенствовать с помощью состояний.

В данном приложении возможны два состояния:

- Вид, отображающий контактную информацию при выборе контакта из списка
- Вид для редактирования данных, предназначенный для внесения изменений в контактную информацию

Для решения такой задачи можно создать два состояния в приложении ContactManager. Однако в этот раз процесс их создания будет осуществлен не в самом приложении, а в отдельном компоненте. Сейчас мы узнаем, как добавить модульность в приложение, создав собственный компонент. Это совсем несложно: потребуется добавить немного MXML.

Создание модульного приложения

Создать новый компонент – все равно что перенести ваш текущий MXML-код в отдельный файл. Во Flex можно без труда создавать *составные компоненты* путем смешивания различных компонентов. Для этого, как правило, создается отдельный файл с расширением *.mxml*, в который помещается контейнер, включающий необходимые составные части (другие компоненты). На созданный компонент можно ссылаться в основном коде через MXML-тег, подобно обращению к стандартным компонентам Flex или компоненту Yahoo! Search, использованному нами для создания приложения Search.

В некотором смысле панель Contact Details и форма Contact Editor очень похожи, различие между ними состоит лишь в том, что в первой используются элементы Label для отображения информации, а во второй – элементы TextInput для редактирования той же самой информации. Вполне очевидно, что такого рода интерфейс можно построить с помощью различного представления одних и тех же данных. Итак, создадим компонент ContactViewer, включающий два состояния – основное и состояние редактирования.

Примечание

Вы уже привыкли видеть в папке с исходным кодом основной MXML-файл с кодом приложения. В состав проекта может входить и несколько приложений, т. е. несколько файлов с расширением *.mxml*, содержащих код, заключенный между тегами <mx:Application/>. В папке src могут находиться и другие файлы с расширением *.mxml*, являющиеся компонентами, не входящими в стандартный набор Flex.

Для создания нового компонента во Flex Builder можно воспользоваться диалоговым окном New MXML Component. Итак, выделите ваш проект на панели Flex Navigator и выберите меню File→New→MXML Component. При этом откроется диалоговое окно, показанное на рис. 12.8.

Рис. 12.8. *Диалоговое окно New MXML Component*

В нем можно задать имя и выбрать тип компонента, служащего основой для вновь создаваемого. Имя файла совпадает с именем самого компонента, а значит, и соответствующего ему MXML-тега, поэтому он не может содержать пробелов и может начинаться исключительно с буквы. Компоненты удобно называть в стиле camel case[1], т. е. начинать смысловые части заглавными буквами. В начале имени также рекомендуется использовать заглавную букву.

[1] CamelCase (*ВерблюжийРегистр*) – стиль написания составных слов, при котором несколько слов пишутся слитно без пробелов, при этом каждое слово пишется с большой буквы. Стиль получил название *CamelCase*, поскольку заглавные буквы внутри слова напоминают горбы верблюда (англ. *Camel*). Широко используется в языках программирования. – *Прим. перев.*

Когда говорят, что один компонент создан на базе другого, это означает наследование им свойств и методов базового компонента и их надстройку. Создание нового MXML-компонента по сути является расширением возможностей другого визуального компонента. Как правило, такими компонентами являются контейнеры, позволяющие заключить в себя другие компоненты, что создает новые возможности для разработчика. Итак, назовем наш новый компонент ContactViewer, а в качестве основы для него выберем контейнер Panel.

Созданный файл откроется в режиме Design или Source. Его код будет следующим:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Panel xmlns:mx="http://www.adobe.com/2006/mxml"
  layout="verical"
  width="400"
  height="300">
</mx:Panel>
```

Обратите внимание на указание пространства имен, как и в теге `<mx:Application/>` – это обязательное условие создания компонента (указывающее на то, что это именно компонент). Все, что содержится внутри данного тега, является частью компонента. Поэтому мы переместим туда фрагмент кода из приложения `ContactManager`, который и станет содержимым компонента.

Перенос кода

Прежде всего откройте приложение `ContactManager` в режиме Source и выделите содержимое панели Contact Details. Иными словами, выберите код, расположенный между тегами `<mx:Panel>` и `</mx:Panel>`. Далее выберите команду Edit→Cut и вставьте в режиме Source выделенный код в новый компонент ContactViewer (между тегами `<mx:Panel>` и `</mx:Panel>`) с помощью команды Edit→Paste.

Примечание

При работе в Flex Builder можно вырезать и копировать целые контейнеры и другие компоненты и в режиме Design. Чтобы выбрать несколько компонентов одновременно, щелкните мышью по сцене и проведите указателем по необходимым компонентам, удерживая нажатой кнопку мыши.

Также нужно скопировать атрибуты, присущие панели Contact Details, за исключением атрибута `id`, и поместить их в новый компонент ContactViewer. Что касается атрибута `id`, в определении компонента его задавать нельзя, поскольку идентификатором может обладать конкретный объект, но не его определение.

Итак, интерфейс базового компонента ContactViewer создан. Теперь создадим в нем состояние редактирования с именем edit. Переключи-

тесь в режим Design, находясь в этом состоянии, и удалите содержимое компонента – при этом исчезнут все его элементы. Таким образом, компоненты, предназначенные только для чтения, будут удалены в состоянии edit, но в основном состоянии они присутствуют по-прежнему. На их месте (в пустой панели ContactViewer) в состоянии edit поместите контейнер Form из панели Contact Details в приложении ContactManager. Проще всего выделить контейнер Form из основного файла вашего приложения в режиме Design и вырезать его. Или же выделите и вставьте соответствующий код в режиме Source, не забыв заключить его в тег <mx:AddChild/>.

Примечание

Вам необязательно в точности следовать приведенным здесь инструкциям. Готовый исходный код примеров, предлагаемых в данной главе, можно получить по адресу *www.greenlike.com/flex/learning* (как и код остальных примеров, представленных в других главах). Главное – понять логику происходящего.

Затем замените заголовок панели Contact Details на Contact Editor в состоянии edit. Теперь оба необходимых вам режима присутствуют в одном компоненте.

Примечание

В главе 11 было подробно рассмотрено использование элемента Image. Его можно использовать и в данном приложении – добавьте соответствующие узлы к каждому контакту в файле *contacts.mxml* – это позволит показать для каждого контакта изображение. В самом приложении для этого нужно добавить элемент Image, привязанный к соответствующему полю в XML-файле.

Однако для нормального функционирования компонента этого недостаточно. Вы, возможно, уже заметили, что он содержит ссылки на свойства contactsDataGrid, а также ссылки на функции, в данном компоненте отсутствующие. Хотя наш компонент и является частью того же проекта, что и приложение ContactManager, ему недоступна содержащаяся в последнем информация. В данном случае нужно сделать две вещи: добавить требующийся для работы компонента код и передать в него необходимую информацию для отображения. Я начну с первого, а затем, когда вы узнаете один полезный прием, можно будет вернуться и ко второму.

Прежде всего переместите тег <mx:Script/> со всем содержащимся в нем кодом ActionScript из приложения ContactManager в компонент ContactViewer сразу после открывающего тега <mx:Panel/>. В компонент, как и в приложение, может входить данный тег и сценарии на ActionScript. Поскольку данный фрагмент кода предназначен для проверки и подгонки данных, его рациональнее поместить внутри созданного нами компонента, а не в основном приложении.

Примечание

В отдельный файл с исходным кодом компонента можно поместить теги <mx:Script/>, что невозможно для компонентов, расположенных прямо в приложении. Таким образом, возможности компонента могут быть расширены с помощью сценариев.

Создание свойств компонента

В компоненте ContactViewer создадим переменную contact типа Object средствами ActionScript или с помощью тега <mx:Object/>. В коде на ActionScript не забудьте добавить тег метаданных [Bindable] перед объявлением переменной, что позволит использовать ее при связывании данных (об этом подробнее говорилось в главе 7). Таким образом, мы создали свойство contact компонента ContactViewer. В нем будут содержаться данные, переданные из свойства selectedItem элемента contactsDataGrid в основном приложении. Однако сначала нужно заменить все ссылки на contactsDataGrid.selectedItem ссылками на новое свойство contact. Например, следующий код

```
<mx:Label id="emailLabel"
  text="Email: {contactsDataGrid.selectedItem.email}"/>
```

следует заменить на

```
<mx:Label id="emailLabel"
  text="Email: {contact.email}"/>
```

Проще всего это сделать с помощью команды Find and Replace (меню Edit →Find/Replace). Заменив ссылки, сохраните компонент и проверьте его на наличие ошибок. Если таковые имеются, их нетрудно найти и исправить благодаря специальному механизму Flex Builder, как правило, выводящему подробное описание ошибки.

Примечание

У вас может вызвать недоумение тот факт, что при сохранении компонента ContactViewer может обнаружиться множество ошибок. Их появления можно избежать, убрав все ссылки на конкретное приложение, – потом данный компонент можно будет свободно использовать в других приложениях.

Использование созданных компонентов

Итак, новый компонент сохранен – пора применить его на практике. Удалите все ссылки на панели Contact Editor или Contact Details, имеющиеся в приложении ContactManager. Потом добавьте ваш новый компонент ContactViewer с помощью панели Components в режиме Design. В самом верху списка компонентов располагается раздел Custom, в котором находятся все созданные компоненты, не входящие в стандартный набор Flex (рис. 12.9). Здесь вы увидите компонент ContactViewer, который можно перетащить на сцену, как и любой другой.

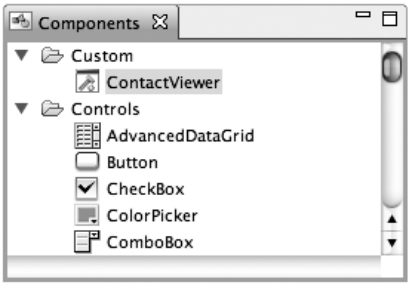

Рис. 12.9. *Созданный нами компонент на панели Components*

Пакетная организация компонентов

Разместив созданный компонент в приложении с помощью режима Design, переключитесь в Source и посмотрите на добавленный код. Скорее всего, он будет выглядеть примерно следующим образом:

```
<ns1:ContactViewer
  x="318"
  y="10">
</ns1:ContactViewer>
```

Это и есть ваш компонент. Поскольку он не входит в состав набора Flex SDK, а лишь создан на его основе и расширяет его, он принадлежит к иному пространству имен – ns1 – для отличия от других компонентов. Flex автоматически пишет за вас необходимый код, в чем вы можете убедиться, взглянув на тег <mx:Application/>, в который добавлено определение пространства имен:

```
xmlns:ns1="*"
```

В переводе на обычный язык это значит: «Нужно создать пространство имен ns1, указывающее на папку с исходным кодом приложения». Звездочка (*) означает, что пространство имен ссылается на папку текущего проекта.

Возможно, такое название пространства имен покажется вам слегка невнятным. Как правило, Flex автоматически создает названия с нумерацией, например ns1, ns2, ns3 и т. д. Их можно с легкостью изменить (о чем уже упоминалось в главе 4), и именно этим мы сейчас займемся.

Как правило, компоненты размещают в папках и подпапках. До сих пор мы не организовывали расположение компонентов специальным образом в целях экономии времени, но обычно этот момент не стоит упускать из виду. Если вы станете разрабатывать Flex-приложения, вы наверняка будете создавать свои собственные компоненты, и роль хорошей организации файлов будет возрастать прямо пропорционально увеличению вашей библиотеки. Прекрасный способ организации

компонентов состоит в их размещении в различных *пакетах*, т. е. папках и подпапках, в зависимости от выполняемой ими функции. К примеру, созданные элементы управления логично поместить в папку с названием controls (элементы управления – англ.), а контейнеры расположения могут находиться в папке containers. Это позволяет различать компоненты по их назначению.

Традиционный способ наименования пакетов использует метод *обратных доменов*. Домен – это, например, *adobe.com* или *oreilly.com*, а, соответственно, обратный домен будет *com.adobe* или *com.oreilly*. Допустим, вы работаете в компании О'Рейли (O'Reilly). В таком случае разработанный вами компонент можно разместить в пакете com.oreilly.controls, при этом ваш компонент должен находиться в каталоге со структурой com/oreilly/controls. Если у вас есть собственный веб-сайт, можете использовать собственное доменное имя для создания структуры пакетов. В противном случае можно просто использовать свое настоящее имя или псевдоним.

Размещение компонентов в специальные папки называется *пакетированием*, поскольку при этом создаются группы схожих по функциям компонентов. Это очень напоминает укладывание вещей одного типа по пакетам или коробкам при переезде в новый дом. Чтобы с первого взгляда было понятно, что находится в коробках, на них можно приклеить ярлыки «посуда» или, скажем, «игрушки». То же самое и с компонентами – название папки помогает тотчас определить предназначение содержащихся в ней компонентов. Кроме того, как уже было сказано, подобная организация поможет избежать путаницы с наименованием компонентов – можно давать компонентам произвольные имена, не думая о том, что это имя может быть уже занято другим компонентом, поскольку пакет компонента может стать его пространством имен. Можно, например, создать компонент с именем Button и поместить его в определенный пакет – это позволит отличить его от стандартного Flex-компонента Button при его использовании в коде.

В качестве примера создадим пакет для нашего компонента ContactViewer. Это совсем нетрудно – для начала нужно полностью удалить тег `<ns1:ContactViewer/>` из кода приложения, а затем убрать объявление пространства имен из тега `<mx:Application/>`.

Примечание

Поскольку данный компонент был только что добавлен в режиме Design, можно удалить как его тег, так и объявление пространства имен с помощью команды Edit→Undo.

Теперь перейдем к созданию пакета – добавим набор пустых папок в текущий проект. Для этого выделите папку с исходным кодом проекта ContactManager (ContactManager/src) на панели Flex Navigator. Затем выберите команду File→New→Folder, чтобы создать новую папку. Вы можете создать любую структуру, которая вам по душе, но в данном при-

мере мы будем придерживаться com/oreilly/view, поскольку рассматриваемый нами компонент используется для создания определенного вида части приложения. С помощью диалогового окна New Folder можно создать как одну-единственную папку, так и целую иерархию, для чего необходимо ввести полный путь к папке, используя каталоги, разделенные левой косой чертой (/). Поэтому проще всего будет просто ввести `com/oreilly/view` в поле Folder name и подтвердить выполнение операции, нажав на кнопку Finish (рис. 12.10).

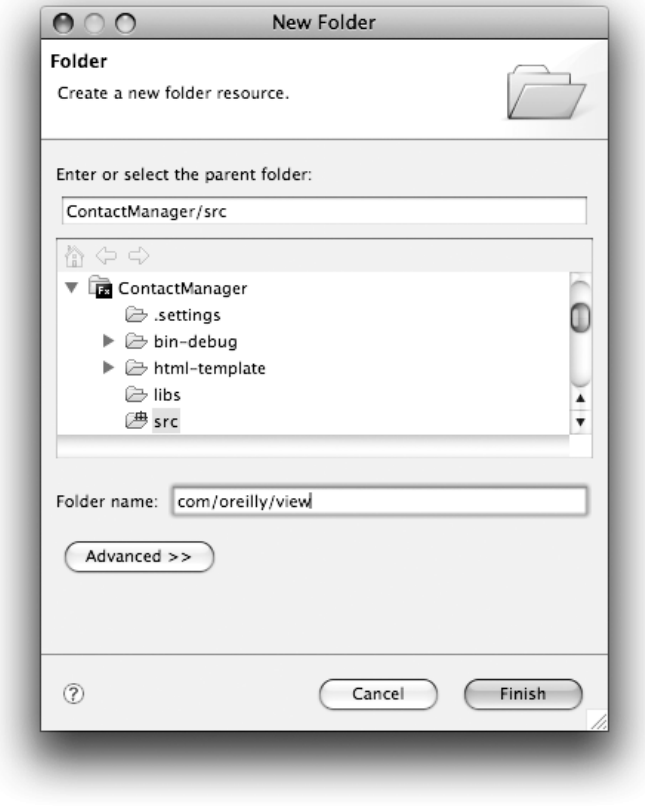

Рис. 12.10. *Диалоговое окно New Folder*

Примечание

Имена пакетов принято вводить в нижнем регистре. Это также помогает отличить их от имен классов, в которых, как правило, используются заглавные буквы. Кроме того, имена пакетов не могут содержать спецсимволы.

Итак, папки созданы. Теперь сделать наш компонент ContactViewer частью пакета com/oreilly/view не составит никакого труда – для этого достаточно лишь перетащить его мышью в папку с именем view.

Папки можно создать и с помощью стандартных инструментов, предлагаемых вашей операционной системой, например Windows Explorer или Mac Finder. Перейдите в папку, содержащую файлы проекта, которая должна находиться по адресу /Users/имя_пользователя/Documents/Flex Builder 3/ для пользователей Mac или C:\Documents and Settings\имя_пользователя\My Documents\Flex Builder 3\ для пользователей Windows. Здесь можно создать новые папки, которые автоматически будут отображены в Flex Builder. Если вы их не видите, выделите проект на панели Flex Navigator и выберите команду File→Refresh для обновления списка папок.

На рис. 12.11 показаны получившаяся структура папок и компонент ContactViewer, расположенный в новом пакете.

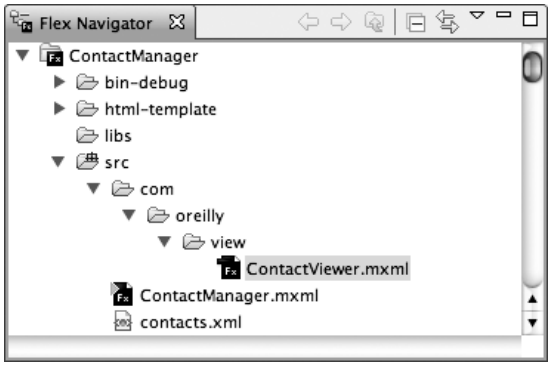

Рис. 12.11. *Пакетная структура созданного компонента*

Компонент, расположенный в новом месте, можно добавить в приложение в режиме Design привычным для вас методом – перетащив его мышью на сцену из панели Components. При переключении в режим Source вы увидите, что в теге `<mx:Application/>` создано новое пространство имен:

```
xmlns:ns1="com.oreilly.view.*"
```

Ранее компонент ContactViewer находился в основной директории (*), теперь же он принадлежит этому пространству имен. Название пространства имен можно изменить на более понятное, например:

```
xmlns:view="com.oreilly.view.*"
```

Естественно, при изменении имени пространства имен его нужно соответствующим образом обновить и в теге компонента:

```
<view:ContactViewer
  x="318"
  y="10">
</view:ContactViewer>
```

И еще об исходниках

Вас, может быть, удивляет, почему созданные вами компоненты помещаются в пространство имен типа *com.oreily.view*, а стандартные компоненты Flex принадлежат пространству имен *http://www.adobe.com/2006/mxml*. Это компания Adobe распространяет Flex-компоненты запакованными в файл .swc специальным методом.

На самом деле все компоненты Flex организованы по пакетам. В этом можно убедиться, взглянув на исходный код компонентов, расположенный в специальном каталоге файловой системы. При установке Flex Builder исходные файлы большей части среды Flex находятся в следующем каталоге (если настройки по умолчанию не были изменены):

- Mac: *Applications/Adobe Flex Builder 3/sdks/3.0.0/frameworks/projects/framework/src/mx*
- Windows: *C:\Program Files\Adobe\Flex Builder 3\sdks\3.0.0\frameworks\projects\framework\src\mx /*

Там вы можете увидеть набор папок, таких как controls, core, utils и т. д. Упомянутый метод, применяемый Adobe, создает сложное пространство имен, в котором размещаются все эти пакеты, поскольку пространство имен в целом не ограничивается одним пакетом, а может использоваться несколькими пакетами.

Как правило, каждому пакету достаточно отдельного пространства имен. При разработке обширной библиотеки компонентов может оказаться предпочтительнее иметь собственное уникальное пространство имен для всех ваших компонентов. С этой целью вы можете создать собственный SWC-файл, используя проект Flex Library, специально предназначенный для создания скомпилированной библиотеки пользовательских компонентов. Этот метод использует файл *manifest.xml*, содержащий определение каждого вашего компонента и указание на его связь с определенным пространством имен. Для более глубокого ознакомления с этом методом обратитесь к разделу о библиотеке компонентов (*component libraries*) в документации Flex.

Использование свойств созданных компонентов

Созданный вами компонент помещен в соответствующий пакет; остается связать его свойство `contact` со свойством `selectedItem` компонента `contactsDataGrid`, чтобы передать в него контактные данные. Это можно реализовать следующим образом:

```
<view:ContactViewer
  contact="{contactsDataGrid.selectedItem}"
```

```
    x="318"
    y="10">
</view:ContactViewer>
```

Теперь информация об изменении `selectedItem` будет передана созданному нами объекту `contact` и вид приложения будет обновлен соответствующим образом.

С помощью панели States можно свободно переключаться между различными состояниями приложения, однако в самом приложении такой возможности пока нет. Чтобы после запуска приложения пользователь смог переходить от основного состояния к состоянию редактирования, нужно добавить соответствующие элементы управления – оптимальным вариантом будет кнопка.

Однако для начала добавим контейнер ControlBar в нижнюю часть компонента ContactViewer в его основном состоянии. Данный контейнер используется только с элементом Panel (и его расширением TileWindow). Он располагается в нижней части панели, тем самым изменяя ее внешний вид. По сути он похож на контейнер HBox, т. к. располагает дочерние элементы по горизонтали. Его удобно использовать для добавления в панель кнопок и подобных элементов управления, чтобы закрепить их в нижней части панели и придать ей завершенный вид. Итак, поместите в контейнер ControlBar кнопку с меткой «Edit», а также задайте ей идентификатор `editAndSaveButton` (кнопка для редактирования и сохранения данных). (Из этого названия явно видно, что данная кнопка будет выполнять различные функции в зависимости от состояния приложения.) И, наконец, установите обработчик события `click` кнопки для изменения текущего состояния (`currentState`) на состояние редактирования (edit).

Примечание

Будьте предельно внимательны при работе с состояниями в режиме Design среды разработки Flex Builder, чтобы нечаянно не перепутать, в каком именно состоянии нужно изменить свойства компонентов – потом будет довольно затруднительно искать и исправлять свою ошибку. К примеру, если вы добавили кнопку, находясь в состоянии редактирования, в основном состоянии она не появится, – и поскольку кнопка была предназначена для перехода в режим редактирования, она не появится при запуске приложения (т. е. в основном состоянии), как предполагалось.

Затем переключитесь в состояние edit в режиме Design и выделите эту же самую кнопку. Замените ее метку на «Save» и измените поведение обработчика ее события `click` так, чтобы при нажатии на кнопку осуществлялся возврат к основному состоянию (это можно сделать с помощью следующего ActionScript-выражения: `currentState=""`). Теперь пользователь, открывший ваше приложение, вначале увидит кнопку Edit, при нажатии на которую открывается панель для редактирования вместо панели отображения данных. В свою очередь кнопка Edit пре-

вратится в кнопку Save, возвращающую приложение в основное состояние (обратите внимание, что данная кнопка, несмотря на ее название, не выполняет функцию сохранения данных).

И в завершение, нужно установить выделение первого пункта в contactsDataGrid, поскольку компонент ContactViewer отображается вне зависимости от того, содержит ли он какие-либо данные, и в случае их отсутствия он будет выглядеть как пустая панель, что нежелательно. Для решения этой проблемы достаточно лишь присвоить свойству selectedIndex таблицы DataGrid значение 0. Теперь при загрузке контактных данных в таблицу первому пункту будет передано выделение, и именно он будет отображен компонентом ContactViewer.

Примечание

Свойство contactsDataGrid.selectedItem, является типом данных объект (Object), но при связывании данных может интерпретироваться как Boolean, т. е. принимать одно из значений true или false. Правило таково: если объект равен null, возвращается значение false, в противном случае – значение true. Таким образом, если значение свойства selectedItem не установлено, оно принимает значение null, что интерпретируется как false при связывании со свойством visible.

Вместо выделения пункта в элементе DataGrid можно просто-напросто скрыть компонент ContactViewer до тех пор, пока пользователь не выберет определенный пункт самостоятельно. Этого можно достичь, присвоив свойству visible компонента ContactViewer значение свойства selectedItem контейнера contactsDataGrid:

```
visible="{ contactsDataGrid.selectedItem }"
```

Данное выражение означает, что при наличии выбранного пункта в таблице contactsDataGrid свойство visible компонента ContactViewer принимает значение true, а при его отсутствии – false. Таким образом, при запуске приложения будет отображен только элемент DataGrid со списком контактов, а контактные данные будут показаны при выборе пользователем определенного пункта этого списка. (Затем после щелчка по кнопке Edit появится панель Contact Editor).

Итак, мы рассмотрели основные возможности приложения ContactManager (рис. 12.12 и 12.13). Ввиду большого объема кода компонента ContactsViewer он не представлен в данной книге, но его можно скачать со справочного сайта. Поскольку основной механизм функционирования приложения сосредоточен в отдельном компоненте ContactViewer, код основного приложения весьма краток:

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

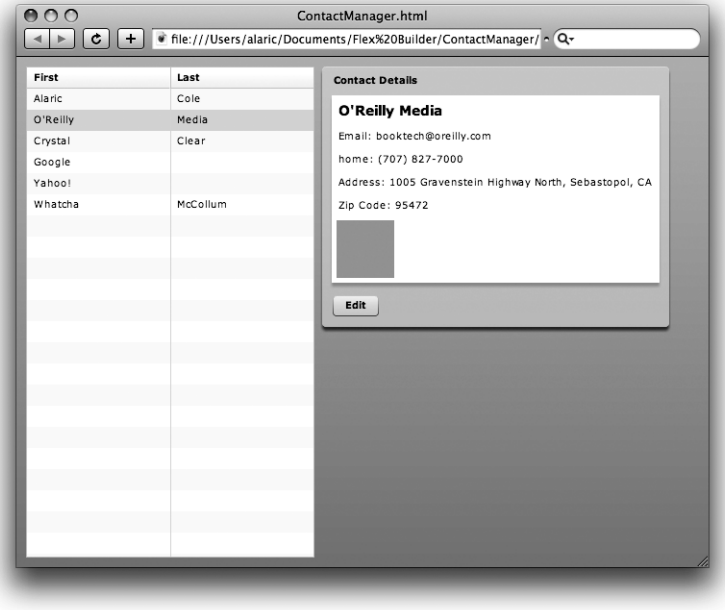

Рис. 12.12. *Основное состояние компонента ContactViewer в приложении ContactManager*

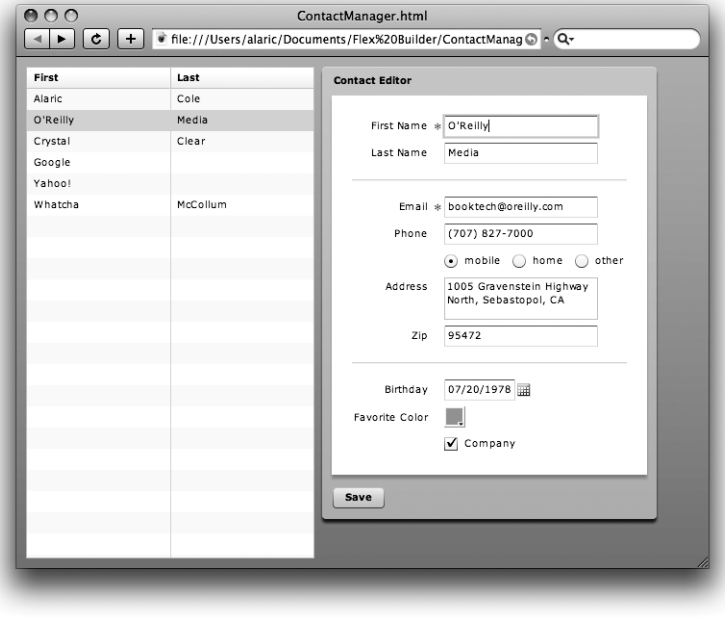

Рис. 12.13. *Состояние редактирования (edit) компонента ContactViewer в приложении ContactManager*

```
  <mx:HTTPService id="contactsService"
    resultFormat="e4x"
    url="contacts.xml" />

  <mx:DataGrid id="contactsDataGrid"
    dataProvider="{contactsService.lastResult.contact}"
    selectedIndex="0"
    left="10"
    top="10"
    bottom="10"
    width="300">
    <mx:columns>
      <mx:DataGridColumn headerText="First"
          dataField="firstName"/>
      <mx:DataGridColumn headerText="Last"
          dataField="lastName"/>
    </mx:columns>
  </mx:DataGrid>

  <view:ContactViewer
    contact="{contactsDataGrid.selectedItem}"
    x="318" y="10">
  </view:ContactViewer>

</mx:Application>
```

Механизм внутренних ссылок

Использование состояний просмотра или контейнеров навигации дает разработчику возможность воспользоваться еще одной полезной функцией – созданием *внутренних ссылок* (ее использование потребует более глубоких знаний ActionScript и понимания основ JavaScript). Такой механизм позволяет программисту сохранять текущее состояние или выбранную в данный момент вкладку TabNavigator в URL-адресе Flex-приложения. Благодаря такой возможности пользователь может добавить закладку на определенный вид вашего приложения и при переходе по данной ссылке откроется соответствующее состояние вашего приложения.

Механизм внутренних ссылок позволяет определить специальный параметр URL, указывающий на определенное состояние. Вы, без сомнения, встречали такие параметры в адресной строке вашего броузера при посещении веб-страниц, хотя, возможно, не задумывались об этом. То, что следует за основным адресом сайта, и является таким параметром, сообщающим серверу дополнительную информацию. Например, при использовании поискового сервиса Yahoo! Search в адресной строке вашего броузера может отобразиться примерно следующее:

http://search.yahoo.com/search?p=flex

В данном примере *http://search.yahoo.com/search* – основной URL, а *p=flex* – единственный параметр *p* со значением *flex*. (Знак вопроса [?] указывает серверу на то, что за ним следует параметр.)

На некоторых веб-сайтах вы наверняка неоднократно встречали гиперссылки на различные разделы одного и того же HTML-документа. Такая ссылка создается с использованием *якоря* при помощи знака решетка (#), за которым следует название соответствующего раздела, например:

http://en.wikipedia.org/wiki/Anchor_tag#Links_and_anchors

Выражение *#Links_and_anchors* является якорем, указывающим на определенный раздел страницы *http://en.wikipedia.org/wiki/Anchor_tag*.

Механизм использования внутренних ссылок во Flex сильно напоминает вышеописанный. Он состоит в создании специальных якорей или параметров URL, позволяющих выйти непосредственно на определенные места вашего приложения.

Более подробную информацию об этой возможности можно найти в документации Flex.

Заключение

Перед вами были раскрыты основные возможности Flex, позволяющие начать разработку собственных насыщенных интернет-приложений. В этой главе вы узнали о понятии состояний просмотра, позволяющих управлять отображением вашего приложения. Вместе с контейнерами и элементами навигации они входят в набор инструментов, необходимых для построения удобного и привлекательного интерфейса приложений любого типа. В самом деле, вы же уже использовали их при создании приложений Search, ContactManager и PhotoGallery.

Вы также приобрели незаменимые навыки создания собственных MXML-компонентов, которые позволили перестроить приложение ContactManager; теперь оно разбито на модули и обладает более логичной структурой. Этот простой пример демонстрирует возможности создания и распространения своих собственных MXML-компонентов, которые могут быть многократно использованы в различных приложениях.

Вас можно поздравить с завершением прочтения самой «технической» части книги. И в ознаменование этого события в следующей главе я расскажу о самой увлекательной части процесса создания Flex-приложения – использовании фильтров и эффектов.

В этой главе:
- Поведения
- Типичные эффекты и их свойства
- Звуковые эффекты
- Новые возможности состояний
- Фильтры

13

Поведения, переходы, фильтры

Теперь, когда вы изучили наиболее важные приемы создания Flex-приложений, пора немного отдохнуть и заняться чем-то увлекательным. Все визуальные компоненты Flex обладают стандартным набором графических фильтров и эффектов, позволяющих без особых усилий создавать выразительные приложения. *Эффекты* – это звуковые или визуальные изменения компонентов, чаще всего из разряда анимации. *Фильтры* – статические визуальные изменения, такие как размывание границ или создание теней. В этой главе вы узнаете о том, как применять визуальные эффекты по отношению к компонентам вашего приложения с помощью несложного MXML-кода.

Поведения

Любой Flex-компонент имеет встроенный набор поведений, позволяющих добавлять выразительные эффекты, буквально оживляющие приложения. Их разнообразие безгранично, но чаще всего используются эффекты, создающие иллюзию постепенного исчезновения и появления компонента, или анимация, прослеживающая путь компонента при его перемещении. Также существуют аудиоэффекты, т. е. сопровождение определенных событий или действий пользователя определенными звуками.

Поведение – это эффект, вызываемый *триггером*, т. е. определенным действием, происходящим в приложении. Типичными примерами таких событий являются щелчок мышью по элементу, перемещение курсора или особые события самого компонента, такие как его создание, отображение или сокрытие.

Использование типичных эффектов

Чтобы понять, что такое поведение, на практике, вернемся к приложению PhotoGallery, над которым мы работали в главе 11. Откройте основной файл приложения: сейчас мы будем добавлять в него различные эффекты.

Данное приложение загружает фотографии при выборе пользователем соответствующего пункта из списка. Кроме того, ход загрузки изображения показан с помощью элемента ProgressBar, исчезающего по окончании загрузки. Визуальное представление этих действий – загрузку фотографий, появление и исчезновение индикатора ProgressBar – можно сделать более плавным и изящным.

Поскольку поведение – это эффект, вызываемый триггером, а, как вы знаете, компоненты вызывают определенные события, то можно ожидать, что события компонента могут быть триггером для вызова эффекта. В табл. 13.1 представлен перечень типичных поведений и соответствующих им событий.

Таблица 13.1. Типичные поведения и соответствующие им события

Название поведения	Соответствующий триггер события	Описание триггера
addedEffect	add	Добавление компонента в список отображения.
removedEffect	remove	Удаление компонента из списка отображения.
creationCompleteEffect	creationComplete	Создание компонента.
focusInEffect	focusIn	Компонент в фокусе.
focusOutEffect	focusOut	Компонент теряет фокус.
showEffect	show	Значение свойства visible компонента изменено с false на true.
hideEffect	hide	Значение свойства visible компонента изменено с true на false.
moveEffect	move	Изменение координаты x и/или y компонента.
resizeEffect	resize	Изменение свойства width или height компонента.
mouseDownEffect	show	Нажатие кнопки мыши, указатель которой расположен над компонентом.
mouseUpEffect	mouseUp	Отпускание кнопки мыши, указатель которой расположен над компонентом.

Название поведения	Соответствующий триггер события	Описание триггера
rollOverEffect	rollOver	Указатель мыши наводится на компонент.
rollOutEffect	rollOut	Указатель мыши покидает область компонента.

Рассмотрим элемент Image в приложении PhotoGallery. По окончании загрузки изображения происходит событие complete, которое уже использовалось нами, чтобы скрыть индикатор ProgressBar. Данный элемент также обладает свойством из категории стилей completeEffect, которое позволяет разработчику определить, какой эффект следует отобразить, когда произойдет событие complete.

В качестве примера добавим эффект Fade для элемента Image, предназначенного для загрузки полноформатного изображения. Это можно сделать, присвоив его свойству completeEffect значение Fade, и тогда Flex создаст стандартный эффект постепенного появления изображения:

```
<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"
  completeEffect="Fade"/>
```

Примечание

Список, представленный в табл. 13.1, содержит наиболее общие и часто используемые поведения визуальных компонентов, но этот перечень далеко не полный. Существуют также события, характерные для определенных компонентов, например completeEffect элемента Image. Для получения полного списка возможных эффектов для конкретного компонента обратитесь к документации Flex. Их также можно увидеть на панели Flex Properties в Category View.

Теперь при запуске приложения и выборе изображения оно загружается постепенно. В тот момент, когда событие complete произошло, но изображение еще не отображено, эффект Fade устанавливает значение 0 для свойства alpha элемента Image, которое затем постепенно увеличивается до полного появления фотографии. В технологии Flash это называется *твин*[1]; анимации подвергается *переход* от одного значения свойства к другому (или от одного состояния к другому).

[1] Термин твин (tween) образован от предлога *between* (между): анимируется переход *между* состояниями или свойствами. – *Прим. ред.*

Примечание

Установка эффекта никак не влияет на соответствующие ему события. К примеру, триггером для `completeEffect` служит событие `complete`, но для запуска эффекта неважно, установлен ли обработчик данного события, – их механизмы абсолютно независимы.

При использовании вышеописанного эффекта изображение появляется не внезапно, а постепенно, что гораздо приятнее с эстетической точки зрения. Резкое изменение изображений заменяется на плавный эффект их появления, что выглядит весьма привлекательно.

Индикатор хода процесса загрузки ProgressBar также появляется и исчезает быстро; на мой взгляд, можно добиться куда лучшего поведения с помощью эффектов. К счастью, все визуальные компоненты обладают поведением `showEffect` и `hideEffect`, благодаря чему можно использовать различные эффекты, видоизменяющие процесс их появления и исчезания. Мы будем использовать оба поведения, чтобы добиться плавного появления и исчезновения элемента ProgressBar:

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

Примечание

В разделе «Типичные эффекты и их свойства», расположенном ниже в этой главе, представлен список самых распространенных эффектов.

Теперь индикатор загрузки появляется постепенно и также плавно исчезает. Согласитесь, это выглядит куда привлекательнее, чем раньше.

Примечание

При присваивании атрибутам `showEffect` и `hideEffect` значения `Fade` будет создан соответствующий анимационный эффект плавного появления или исчезновения.

Однако вы, должно быть, заметили, что при запуске приложения эффект плавного появления и исчезновения не затронул текстовую метку (заданную свойством `label`) элемента ProgressBar, в отличие от его графических составляющих – она появляется и исчезает так же резко, как и прежде. Причина в том, что по умолчанию при установке Flex-приложения для отображения текстовой информации используются *шрифты устройства*, установленные в системе пользователя. На данный момент Flash Player не позволяет регулировать прозрачность не встроенных в ролик шрифтов, поэтому соответствующий эффект не

может быть отображен. Такие шрифты также нельзя поворачивать под различными углами и производить с ними некоторые иные операции, что накладывает определенные ограничения на типы используемых с ними эффектов.

Решить данную проблему можно двумя путями. Во-первых, можно *встроить* в приложение свои собственные шрифты, которые будут в него включены при компиляции (об этом подробнее говорится в следующей главе). При этом вы сможете регулировать прозрачность букв и, соответственно, использовать эффект Fade. Можно пойти более простым путем и просто-напросто удалить текстовую метку, не несущую в данном случае особой смысловой нагрузки, поскольку ProgressBar появляется лишь временно и благодаря специфическому внешнему виду наглядно демонстрирует свое предназначение. Для этого достаточно просто изменить значение соответствующего свойства на пустую строку:

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

Примечание

Несмотря на то, что к шрифтам устройства нельзя применять эффекты, связанные с прозрачностью (Fade) или углом расположения (Rotate), другие эффекты можно использовать безо всяких проблем.

Применение эффектов по отношению к контейнерам навигации

Контейнеры навигации могут регулировать видимость дочерних элементов, поэтому для них можно использовать поведение `hideEffect` и `showEffect`. В приложении `PhotoGallery` расположен контейнер TabNavigator, содержащий два дочерних контейнера Canvas, включающие два разных списка. При переключении видов они сменяются довольно резко, поэтому имеет смысл это скорректировать с помощью эффектов. Просто определите эффект `Fade` для поведения `showEffect` для обоих контейнеров для смягчения визуализации переходов между видами – кроме отображения первого вида.

Вы, конечно, сразу заметили, что данный эффект прекрасно работает с видом Tile, представляющим собой список TileList, состоящий из миниатюр изображений, однако вид с простым списком List не обрабатывается данным эффектом. Проблема все в тех же шрифтах устройства, используемых для отображения текстовых пунктов списка. В данном случае ее несложно решить, используя вместо Fade эффект Dissolve.

При его использовании белый прямоугольник накладывается поверх списка List, и эффект плавного исчезновения применяется к этому прямоугольнику. В результате можно наблюдать постепенное появление списка List. Просто замените значение showEffect контейнера Canvas с видом List View на Dissolve и можно любоваться результатом:

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

  <mx:Canvas
    label="Tile View"
    width="100%"
    height="100%"
    showEffect="Fade">
    <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>
```

Теперь приложение PhotoGallery стало намного приятнее с виду и в обращении, и для создания такой красоты потребовалось всего несколько строчек MXML-кода, написанных буквально за пару минут!

Примечание

В этом приложении можно использовать еще одно поведение – creationCompleteEffect, реализующее определенный эффект при первоначальном создании компонента.

Еще эффектов?

При разработке приложений довольно часто используется эффект Resize, создающий анимацию при изменении размера компонента. Его было бы весьма уместно применить в нашем приложении ContactManager, в котором в зависимости от состояния или от размера его содержимого изменяется размер компонента ContactViewer. С помощью эффекта Resize данный процесс можно сделать более плавным. Кроме того, что еще более существенно, в нижней части компонента находится кнопка, выполняющая функцию либо перехода в режим редактирования, либо сохранения введенных данных. При изменении размера компонента она моментально меняет свое расположение. С помощью упомянутого эффекта можно немного замедлить этот процесс.

Откройте приложение ContactManager и добавьте атрибут поведения resizeEffect с эффектом Resize компоненту ContactViewer:

```
<view:ContactViewer id="contactViewer"
  contact="{contactsDataGrid.selectedItem}"
  x="318"
  y="10"
  resizeEffect="Resize">
</view:ContactViewer>
```

Теперь при запуске приложения изменение размера компонента будет сопровождаться приятным анимационным эффектом. При нажатии на кнопку Edit компонент ContactViewer переходит в состояние редактирования, при этом благодаря применению эффекта Resize становится заметно, как его размер регулируется, чтобы разместить дополнительные элементы, появляющиеся в данном состоянии.

Однако здесь возникает еще одна помеха – при изменении размеров компонента появляются (или исчезают) полосы прокрутки, что сразу бросается в глаза. Это происходит из-за того, что изменение размеров может повлечь за собой обрезание части содержимого, во избежание чего добавляются полосы прокрутки. На самом деле изменить такое поведение не составляет никакого труда, поскольку каждый компонент обладает свойствами для его регулирования на усмотрение разработчика – horizontalScrollPolicy и verticalScrollPolicy. Они определяют, есть ли нужда в появлении горизонтальных и вертикальных полос

прокрутки соответственно. По умолчанию принимается значение auto, что означает появление прокрутки по мере необходимости. Полностью запретить их появление можно, установив значение off, а значение on, напротив, указывает компоненту на необходимость отображения полос прокрутки в обязательном порядке.

```
<view:ContactViewer id="contactViewer"
  contact="{contactsDataGrid.selectedItem}"
  x="318" y="10"
  resizeEffect="Resize"
  horizontalScrollPolicy="off"
  verticalScrollPolicy="off">
</view:ContactViewer>
```

Примечание

Учтите, что данный способ – установка значения off свойств horizontalScrollPolicy и verticalScrollPolicy – решает проблему визуального отображения эффекта Resize, но теперь полосы прокрутки не появятся даже тогда, когда они действительно нужны.

До сих пор эффект Fade использовался при изменении видимости компонента, а Resize – при изменении размеров (resizeEffect). Однако стоит отметить, что поведение и эффект взаимозаменяемы. Это означает, что поведению resizeEffect не обязательно соответствует эффект Resize – можно использовать и любой другой эффект, например Fade или Wipe. Но чаще всего конкретное поведение использует один наиболее подходящий эффект.

Примечание

Мы только что рассмотрели несколько вполне жизнеспособных примеров использования распространенных эффектов. Попробуйте самостоятельно создать новый проект и на практике ознакомиться с особенностями применения других возможных эффектов.

Настройка эффектов

Пока мы использовали поведения с эффектом, настроенным по умолчанию, т. е. в качестве значения поведения указывалось просто имя класса эффекта, например, Fade или Resize. Это очень удобно, но в определенных случаях необходим более тонкий контроль за свойствами применяемого эффекта. С этой целью создается объект эффекта с помощью соответствующего MXML-тега. Это позволит создавать различные виды одного и того же эффекта для применения в разных случаях.

Примечание

Теги эффектов, как и любых других невизуальных компонентов, нельзя размещать внутри контейнеров, за исключением тега верхнего уровня <mx:Applica-

tion/> (или корневого тега нестандартного компонента). Предпочтительнее располагать такие теги в верхней части кода приложения или компонента.

Чтобы создать настраиваемый объект эффекта Resize, добавьте тег <mx:Resize/> в приложение ContactManager и задайте ему идентификатор fastResize. Как и любой другой эффект, он обладает свойством duration, определяющим временной промежуток, измеряемый в миллисекундах, в течение которого должен отображаться эффект. Задайте ему значение 300. Поскольку триста миллисекунд составляет примерно треть секунды, эффект должен быть весьма живым:

```
<mx:Resize id="fastResize" duration="300"/>
```

На объявленный эффект Resize можно в дальнейшем ссылаться в коде приложения по его id. Можно заменить текущее поведение resizeEffect компонента ContactViewer этим конкретным экземпляром. Вместо стандартного эффекта Resize, достигаемого присваиванием соответствующего значения поведению, можно передать ему специальным образом настроенный эффект fastResize:

```
<view:ContactViewer id="contactViewer"
  contact="{contactsDataGrid.selectedItem}"
  x="318"
  y="10"
  resizeEffect="{fastResize}"
  horizontalScrollPolicy="off"
  verticalScrollPolicy="off">
</view:ContactViewer>
```

Запустите приложение, чтобы увидеть полученный результат. Обратите внимание на то, как меняется впечатление от приложения, т. к. эффект Resize происходит гораздо быстрее.

Примечание

Применение настроенных эффектов предполагает использование фигурных скобок, указывающих на связывание данных.

Совместное использование нескольких эффектов

Resize прекрасно подходит для добавления эффекта анимации при увеличении размеров ContactViewer, когда осуществляется переход из режима отображения данных в режим редактирования. Однако возможности Flex этим не ограничиваются. Было бы неплохо добавить еще и эффект Dissolve, осуществляющий плавный переход из одного режима в другой. Но как этого достичь, если уже установлен эффект Resize?

Parallel

Для решения этой задачи вовсе не обязательно заменять эффект Resize эффектом Dissolve – последний можно просто добавить. Благодаря на-

личию специального тега Parallel можно накладывать несколько эффектов одновременно. Для этого все они должны быть заключены между тегами <mx:Parallel/>. Это дает разработчику возможность использования и настройки комплексных эффектов.

В нашем случае должно произойти одновременное применение эффектов Dissolve и Resize, поэтому необходимо заменить предыдущий код объекта Resize на следующий:

```
<mx:Parallel id="fadeAndResize">
  <mx:Dissolve/>
  <mx:Resize id="fastResize" duration="300"/>
</mx:Parallel>
```

Следующий шаг – связывание resizeEffect компонента ContactViewer с набором эффектов fadeAndResize, после чего при изменении размеров компонента будут отображены оба эффекта: Dissolve и Resize. Это поможет избежать обрезания содержимого, происходящего при использовании одного только эффекта Resize. Такое комплексное преобразование выглядит очень привлекательно и сразу дает пользователю понять, что происходит смена режимов.

Примечание

Свойство duration тега Parallel устанавливает длительность внутренних эффектов, если данное свойство не определено для каждого из них отдельно.

Sequence

Существует особый тег <mx:Sequence/>, позволяющий определить последовательность эффектов. Подобно тегу Parallel он включает в себя набор эффектов, однако отображаться они будут по порядку (а не одновременно, как в предыдущем примере). Вы можете сами в этом убедиться, заменив тег <mx:Parallel/> на <mx:Sequence/> в только что рассмотренном коде.

Примечание

При использовании последовательности эффектов полезно использовать тег <mx:Pause/>. Его свойство duration позволяет определить длительность промежутков между отображением различных эффектов.

Типичные эффекты и их свойства

Помимо duration все эффекты обладают специальными свойствами, регулирующими их отображение. Такие свойства определяют начальное и конечное значение объекта трансформации, позволяя разработчику получать различные вариации одного и того же эффекта. К примеру,

для эффекта Resize можно задать начальное и конечное значения свойств `width` и `height`, а для эффекта Move – свойств `x` и `y`.

Данным свойствам не обязательно присваивать какие-либо значения, поскольку Flex автоматически определяет их в зависимости от изменений соответствующих свойств компонента. Например, если вы измените значение свойства `x` панели Panel с `0` на `100` и добавите к ней эффект Move, свойства `xFrom` и `xTo` примут соответствующие значения `0` и `100` автоматически.

Если значения свойств эффекта не заданы явно и Flex не может определить их значения из свойств компонента, то используются настройки эффекта по умолчанию.

Ниже представлена информация о типичных эффектах и их наиболее важных свойствах.

Примечание

Живые и наглядные примеры использования всех описанных здесь эффектов расположены по адресу *http://examples.adobe.com/flex3/componentexplorer/explorer.html*. В разделе «Effects, View States and Transitions» вы найдете отличные примеры вместе с исходным кодом, который можно использовать в собственных приложениях.

Blur

Эффект Blur размывает очертания изображения, как при отсутствии фокусировки.

Его действие можно настроить с помощью следующих свойств:

`blurXFrom`

Определяет начальную степень размытия по горизонтали.

`blurXTo`

Определяет конечную степень размытия по горизонтали.

`blurYFrom`

Определяет начальную степень размытия по вертикали.

`blurYTo`

Определяет конечную степень размытия по вертикали.

Данный эффект можно очень удачно использовать для имитации движения. Его можно применять совместно с эффектом Move. С помощью очень сильного размытия (со значением более 20) можно добиться впечатления некоей трансформации.

Dissolve

Эффект Dissolve регулирует свойство `alpha` прямоугольного наложения, что создает впечатление постепенного проявления или исчезновения лежащего под ним компонента.

Его действие можно настроить с помощью следующих свойств:

`alphaFrom`

Определяет первоначальное значение свойства `alpha`.

`alphaTo`

Определяет конечное значение свойства `alpha`.

`color`

Определяет цвет прямоугольника, наложенного над целевым компонентом. По умолчанию оно совпадает с цветом фона данного компонента (установленного с помощью свойства `backgroundColor`), поэтому, как правило, не требует никаких изменений. Если значение `backgroundColor` компонента не задано, используется белый цвет (`0xFFFFFF`).

Dissolve с успехом замещает эффект Fade при использовании шрифтов устройства.

Примечание

Если эффект Dissolve применяется по отношению к контейнеру, он действует только на его дочерние элементы. К примеру, если речь идет о панели Panel, ее рамка и строка заголовка не будут затронуты данным эффектом.

Fade

Данный эффект воздействует на свойство `alpha` компонента, изменяя его прозрачность.

Его действие можно настроить с помощью следующих свойств:

`alphaFrom`

Определяет первоначальное значение свойства `alpha`.

`alphaTo`

Определяет конечное значение свойства `alpha`.

Glow

Данный эффект используется для подсветки компонента, т. е. создает впечатление исходящего от него света.

Его действие можно настроить с помощью следующих свойств:

`alphaFrom`

Определяет первоначальное значение свойства `alpha`.

`alphaTo`

Определяет конечное значение свойства `alpha`.

`blurXFrom`

Определяет первоначальную степень горизонтального размытия подсветки.

`blurXTo`

Определяет конечную степень горизонтального размытия подсветки.

`blurYFrom`

Определяет первоначальную степень вертикального размытия подсветки.

`blurYTo`

Определяет конечную степень вертикального размытия подсветки.

`color`

Определяет цвет подсветки.

`inner`

Выбирает одну из двух возможностей – внутреннюю или внешнюю подсветку. По умолчанию используется значение `false`, соответствующее внешней подсветке.

`knockout`

Задает режим «прозрачного окна», при котором цвет самого объекта станет более прозрачным и сквозь него будет проступать цвет фона компонента. Значение по умолчанию – `false`.

Iris

Данный эффект осуществляет анимацию в виде расширяющейся или сужающейся прямоугольной рамки, подобной диафрагме фотоаппарата, отцентрированной на объекте. Данная рамка раскрывает или, наоборот, скрывает содержимое компонента, на который она наложена. Вспомните старые фильмы – такой эффект раньше часто применялся в их конце.

Действие данного эффекта можно настроить с помощью следующего свойства:

`showTarget`

Позволяет определить, должен ли компонент быть отображен (`true`) или скрыт (`false`, настройка по умолчанию).

Move

Данный эффект позволяет осуществить постепенное перемещение компонента.

Его действие можно настроить с помощью следующих свойств:

`xFrom`

Определяет начальное положение компонента по горизонтальной оси.

`xTo`

Определяет конечное положение компонента по горизонтальной оси.

`yFrom`

Определяет начальное положение компонента по вертикальной оси.

`yTo`

Определяет конечное положение компонента по вертикальной оси.

`xBy`

Задает количество пикселов, на которое нужно сдвинуть компонент по горизонтали. Это свойство можно использовать вместо `xFrom` или `xTo`, чтобы определить *расстояние*, на которое смещается компонент относительно начального или конечного значения координаты `x`. Если заданы значения обоих свойств `xFrom` и `xTo`, значение данного свойства игнорируется.

`yBy`

Задает количество пикселов, на которое нужно сдвинуть компонент по вертикали. Это свойство можно использовать вместо `yFrom` или `yTo`, чтобы определить *расстояние*, на которое смещается компонент относительно начального или конечного значения координаты `y`. Если заданы значения обоих свойств `yFrom` и `yTo`, значение данного свойства игнорируется.

Примечание

Данный эффект не будет отображаться должным образом при его применении по отношению к элементу, находящемуся внутри контейнера с относительным размещением. Дело в том, что сразу после перемещения компонента с помощью Move контейнер тотчас вернет его на прежнее место.

Resize

Изменяет значения ширины и высоты компонента в отведенный временной промежуток, анимируя изменение размеров.

Его действие можно настроить с помощью следующих свойств:

`widthFrom`

Определяет начальную ширину компонента.

`widthTo`

Определяет конечную ширину компонента.

`heightFrom`

Определяет начальную высоту компонента.

`heightTo`

Определяет конечную высоту компонента.

`widthBy`

Устанавливает требуемое изменение ширины компонента в пикселах. Данное свойство можно использовать вместо свойства `widthFrom` или `widthTo`, чтобы определить, *на сколько* нужно изменить задан-

ный начальный или конечный параметр размера. Если заданы значения обоих свойств `widthFrom` и `widthTo`, значение данного свойства игнорируется.

`heightBy`

Устанавливает требуемое изменение высоты компонента в пикселах. Данное свойство можно использовать вместо свойства `heightFrom` или `heightTo`, чтобы определить, *на сколько* нужно изменить заданный начальный или конечный параметр размера. Если заданы значения обоих свойств `heightFrom` и `heightTo`, значение данного свойства игнорируется.

`hideChildrenTargets`

Данное свойство можно использовать при применении эффекта Resize по отношению к контейнеру Panel. Оно позволяет скрыть его содержимое в процессе изменения размеров. Принимает значения массива панелей Panel.

Когда от изменения размера компонента, к которому применяется эффект Resize, зависят размеры других компонентов, данный эффект распространяется и на них. С такой ситуацией можно столкнуться при использовании компонентами ограничителей, зависящих от размера основного компонента, или при относительном расположении компонентов приложения.

Rotate

Данный эффект изменяет положение компонента путем его поворота относительно определенной точки (по умолчанию, левого верхнего угла приложения). Точку можно установить самостоятельно; ею может быть, к примеру, центр компонента. Кроме того, пользователем задаются значения начального и конечного углов поворота в пределах 360 градусов. Если данное значение превысит допустимое, оно будет приравнено к максимально возможному – 360 градусам.

Действие эффекта можно настроить с помощью следующих свойств:

`angleFrom`

Определяет начальный угол поворота.

`angleTo`

Определяет конечный угол поворота.

`originX`

Используется для определения точки, относительно которой осуществляется поворот, устанавливая ее положение по горизонтали относительно целевого компонента. По умолчанию значение равно 0.

`originY`

Используется для определения точки, относительно которой осуществляется поворот, устанавливая ее положение по вертикали относительно целевого компонента. По умолчанию значение равно 0.

WipeLeft, WipeRight, WipeUp и WipeDown

Эти эффекты регулируют видимость компонента, отображая или скрывая их содержимое. Их использование подобно невидимому прямоугольнику, перемещающемуся над компонентом.

Действие данных эффектов можно настроить с помощью следующего свойства:

`showTarget`

Позволяет определить, должно ли содержимое компонента быть отображено (`true`) или скрыто (`false`, настройка по умолчанию).

Примечание

Принцип действия эффектов Wipe и Iris состоит в наложении маски, или сокрытии видимой части элемента другим графическим элементом. Маска в эффекте Iris масштабируется от центра или к центру элемента, в то время как в эффекте Wipe – перемещается над ним.

Zoom

Увеличивает или уменьшает компонент подобно объективу фотоаппарата. Данный эффект позволяет осуществлять масштабирование, создавая впечатление, что компонент находится очень далеко или, наоборот, очень близко.

Действие эффекта можно настроить с помощью следующих свойств:

`zoomHeightFrom`

Определяет изначальный масштаб компонента по вертикали. По умолчанию принимается значение 1. Значение 2 удвоит размер компонента, масштабируя его соответствующим образом, 3 – утроит и т. п.

`zoomHeightTo`

Определяет конечный масштаб компонента по вертикали.

`zoomWidthFrom`

Определяет изначальный масштаб компонента по горизонтали.

`zoomWidthTo`

Определяет конечный масштаб компонента по горизонтали.

`originX`

Используется для определения точки, относительно которой осуществляется масштабирование, устанавливая ее положение по горизонтали относительно целевого компонента. По умолчанию ею является центр компонента.

`originY`

Используется для определения точки, относительно которой осуществляется масштабирование, устанавливая ее положение по вер-

тикали относительно целевого компонента. По умолчанию ею также является центр компонента.

AnimateProperty

Этот эффект полностью определяется пользовательскими настройками. Он позволяет анимировать, т. е. осуществлять переход от одного числового значения конкретного свойства компонента к другому.

В распоряжении у разработчика имеются следующие свойства для настройки:

`property`

Определяет имя изменяемого свойства (принимает значения типа String).

`fromValue`

Определяет начальное значение свойства.

`toValue`

Определяет конечное значение свойства.

К примеру, для создания эффекта, напоминающего Resize, но применяемого по отношению к одной лишь высоте компонента Panel, можно задать значение `height` свойству `property` данного эффекта, а также необходимые значения `fromValue` и `toValue`.

Звуковые эффекты

Кроме визуальных эффектов в приложении могут применяться и звуковые эффекты. Их удобно использовать в качестве звукового оповещения при открытии диалогового окна – например, при выводе напоминания в приложении, выполняющем функции календаря. Если вы захотите разрабатывать игры на базе Flex, вам наверняка приглянется возможность добавления различных звуков к соответствующим событиям.

Примечание

Flex не предоставляет разработчику набора готовых звуков, так что придется использовать свои собственные.

Для создания звукового эффекта используется тег `<mx:SoundEffect/>`. В его свойстве `source` можно указать ссылку на файл в формате MP3. На созданный объект `SoundEffect` можно ссылаться по его идентификатору (`id`), связывая его с такими поведениями компонента, как `showEffect`, `resizeEffect`, `mouseDownEffect` и т. п. Использование данного эффекта аналогично рассмотренным ранее визуальным эффектам, хотя он сам таковым не является.

Допустим, у вас есть записанный звук щелчка затвора фотоаппарата в формате MP3, и вы хотите, чтобы этот звук сопровождал загрузку

каждой новой фотографии в приложении PhotoGallery. Такого эффекта можно достичь с помощью поведения completeEffect элемента Image.

Тег данного эффекта стоит разместить в верхней части кода приложения. Он может выглядеть следующим образом:

```
<mx:SoundEffect id="shutterSound"
  source="camera_shutter.mp3"/>
```

Эффективные эффекты

При добавлении различных поведений и эффектов легко перейти грань разумного. Их так просто добавлять и результат настолько приятен для глаз, что вы можете захотеть их использовать при любой возможности. Конечно, эффекты – это хорошо. Но, каждый раз применяя тот или иной эффект, задумайтесь, с какой целью вы это делаете. Просто потому, что такой крутой эффект невозможно не добавить, или же на то есть более веская причина? Собственно, придать приложению крутой вид – достаточно убедительный довод – и, в конце концов, это же Flash, – но излишнее увлечение эффектами может привести к созданию негативного впечатления, и вдобавок к существенному замедлению работы приложения.

Длительность эффекта – существенный фактор, нередко служащий для оценки пользователем производительности приложения. Слишком медленный переход может навести на мысль о том, что приложение «тормозит», а слишком быстрый – создать впечатление работы скачками. Старайтесь тщательно регулировать длительность эффекта, чтобы создать нужное впечатление. Также не забывайте о том, что, когда пользователь впервые видит эффект, он может быть просто восхищен, но каждый раз наблюдать один и тот же эффект может оказаться весьма утомительным.

Будьте особенно внимательны при использовании звуковых эффектов. Некоторым пользователям нравится дополнительное звуковое сопровождение определенных операций, другие же предпочитают работать в тишине. Учитывая это, лучше добавить в приложение возможность включения и выключения звуков.

Лучше всего использовать эффекты, чтобы сообщать пользователю дополнительную информацию о приложении. К примеру, пользователь может и не заметить, что произошло перемещение панели из одной части приложения в другую, если она просто исчезла в одном месте и вдруг появилась в другом (возникает вопрос: «И куда же делась панель?»). При применении эффекта Move перенос панели становится очевидным. Эффекты могут сделать приложение более дружественным, а это – большое достоинство.

Применить данный эффект по отношению к элементу Image можно так:

```
<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"
  completeEffect="{shutterSound}"/>
```

Примечание

В качестве источника звукового эффекта лучше использовать небольшие файлы, поскольку перед воспроизведением они должны быть полностью загружены, а загрузка может занять некоторое время. В следующей главе мы обсудим способ, альтернативный загрузке звуков в момент запуска приложения, – встраивание необходимых ресурсов в скомпилированное приложение.

Новые возможности состояний

Конечно, вы можете применять эффекты для изменения поведения любого отдельного компонента, но их можно использовать и при смене состояний приложения. Используя *переходы* (*transitions*), можно применить эффект (или множество эффектов) по отношению ко многим компонентам при изменении состояния.

Вернемся к приложению Search, с которым мы работали в предыдущей главе. В нем используются состояния. Данное приложение функционирует следующим образом: вначале в его центре отображается большое поле для ввода поискового запроса; при поступлении данных, содержащих результаты поиска, поле уменьшается и перемещается в верхнюю левую часть приложения, освобождая место для их отображения. Сейчас это перемещение происходит довольно резко, и не сразу можно догадаться, что именно произошло. Чтобы данный процесс был более плавным и выглядел более естественно, можно добавить переход.

Для использования переходов применяется свойство transition тега Application или компонента, созданного разработчиком. Как правило, значением данного свойства являются данные сложной структуры, поэтому он представлен в виде дочернего тега, а не атрибута. Подобно свойству states тега Application, принимающего в качестве значения массив тегов <mx:State/>, свойство transition может принимать значение в виде одного или нескольких тегов <mx:Transition/>, каждый из которых, в свою очередь, принимает в качестве значения определенный эффект. Для нашего примера достаточно одного перехода – с одновременными (Parallel) эффектами перемещения и изменения размера.

Итак, откройте проект Search и поместите следующий MXML-код где-нибудь внутри корневого тега <mx:Application/> основного файла приложения *Search.mxml*. (Лично я предпочитаю располагать код переходов рядом с кодом, описывающим состояния.)

```
<mx:transitions>
  <mx:Transition>

    <mx:Parallel target="{hbox1}">
      <mx:Move />
      <mx:Resize />
    </mx:Parallel>

  </mx:Transition>
</mx:transitions>
```

Обратите внимание на атрибут target данного перехода, осуществляющего эффект Parallel. Он указывает на контейнер HBox, содержащий поля для ввода поискового запроса. Поскольку самому контейнеру не присвоено определенное поведение (например, moveEffect или resizeEffect), он указывается с помощью свойства target в качестве целевого объекта, по отношению к которому применяется эффект.

Данный код будет выполняться при каждой смене состояний приложения; при этом к компоненту с именем hbox1 будет применен эффект Parallel.

Примечание

Для определения имени объекта, к которому применяется эффект, используется метод связывания данных. Если объектов несколько, можно воспользоваться свойством targets, значением которого является массив элементов. К примеру, чтобы применить эффект Move к двум кнопкам с именами submitButton и resetButton, используется следующий код:

```
<mx:Move targets="{ [submitButton, resetButton] }" />
```

В связи с этим возникает следующий вопрос: почему бы вместо этого не определить поведение moveEffect и resizeEffect самого контейнера HBox? Это, конечно, неплохой вариант, но использование переходов позволяет применять эффекты, основываясь на изменении состояний. Использование moveEffect и resizeEffect контейнера HBox позволяет применять эффекты при изменении его размеров или перемещении, в то время как с помощью переходов можно настроить применение эффектов только при определенных изменениях состояний.

Примечание

Объявлять и применять эффекты можно и программным путем. Вместо использования переходов состояний или поведений для применения эффектов можно вызвать метод play() необходимого эффекта (как визуального, так и звукового). Такой способ позволяет полностью контролировать выполнение эффекта. Однако для его правильного функционирования необходимо также либо зара-

нее задать эффекту свойство target, либо определить компонент для воздействия в самом методе play(). К примеру, чтобы применить эффект Dissolve с именем fastDissolve к элементу управления списком List, названному shortList, необходимо осуществить вызов fastDissolve.play(shortList).

При запуске данного кода вы сразу заметите, что контейнер HBox (вместе с полями для ввода поискового запроса и кнопкой подтверждения) сначала вылетит из верхнего левого угла. Дело в том, что начальное состояние приложения отлично от основного – это было указано нами с помощью свойства currentState – и данный контейнер в нем размещается в левом верхнем углу приложения. Это интерпретируется как изменение размера и местонахождения компонента, и срабатывают заданные эффекты. Скорее всего, вы не ожидали такого поворота событий – сейчас я объясню, как все исправить.

Одним из значительных достоинств использования переходов является возможность определения, какие именно изменения состояний должны осуществить вызов их выполнения. Чтобы заданный эффект применялся только при смене состояния поиска, нужно изменить код перехода следующим образом:

```
<mx:transitions>
  <mx:Transition fromState="search">

    <mx:Parallel target="{hbox1}">
      <mx:Move />
      <mx:Resize />
    </mx:Parallel>

  </mx:Transition>
</mx:transitions>
```

В данном случае используется свойство перехода fromState, определяющее, что эффект должен применяться только при переходе из состояния search в любое другое. Теперь после запуска приложения эффект будет выполняться не сразу, а после того, как пользователь воспользуется поиском, поскольку при загрузке приложение переключается из основного состояния в состояние search, а затем, при отправлении поискового запроса, осуществляется обратный переход в основное состояние.

Примечание

Класс Transition также обладает свойством toState, с помощью которого можно определить применение эффекта при переходе в определенное состояние. В зависимости от конкретных задач можно применять свойство fromState или toState или оба вместе. При отсутствии специально заданных указаний будут использованы настройки по умолчанию – это означает, что эффекты будут применяться при любом изменении состояний.

Использование эффектов действий

Наша работа с переходами подходит к завершению, но для полного совершенства нашему приложению все еще чего-то не хватает. Поскольку при поступлении результатов поиска список resultsList вновь добавляется в список отображения (и, соответственно, осуществляется переход в основное состояние), это прерывает отображение эффектов Move и Resize для контейнера HBox. Конечно, хотелось бы, чтобы вначале все эффекты были отображены полностью, а *затем* уже происходило добавление списка данных List.

Для этого будем использовать тег <mx:Sequence/> в сочетании со специальном тегом эффекта <mx:AddChildAction/>. Поместите тег <mx:Parallel/> вместе со всем его содержимым внутрь тега <mx:Sequence/>, а в конце добавьте тег <mx:AddChildAction/>:

```
<mx:transitions>
  <mx:Transition fromState="search">

    <mx:Sequence>
      <mx:Parallel target="{hbox1}">
        <mx:Move />
        <mx:Resize />
      </mx:Parallel>

      <mx:AddChildAction target="{resultsList}"/>
    </mx:Sequence>

  </mx:Transition>
</mx:transitions>
```

Тег <mx:AddChildAction/> позволяет определить, *когда* в список отображения должен быть добавлен дочерний элемент. В нашем приложении вначале должны отобразиться эффекты Move и Resize для HBox, а затем – список resultsList, поэтому внутри тега Sequence после тега <mx:Parallel/> помещается специальный тег <mx:AddChildAction/>. Теперь все должно работать в правильной последовательности: вначале отображение эффектов, а затем уже вывод результатов поиска.

Мы только что использовали *эффект действия*. Несмотря на такое название, он не создает никакой анимации, а вместо этого производит действие, определяемое соответствующим тегом. В вашем распоряжении имеются четыре разных эффекта действия, соответствующие тегам изменения состояния с похожим названием:

SetPropertyAction

Присваивает свойство компоненту. Относится к дочернему тегу <mx:SetProperty/> состояния.

SetStyleAction

Присваивает компоненту свойство из категории стилей. Относится к дочернему тегу <mx:SetStyle/> состояния.

AddChildAction

Добавляет компонент в список отображения. Относится к дочернему тегу `<mx:AddChild/>` состояния.

RemoveChildAction

Удаляет компонент из списка отображения. Относится к дочернему тегу `<mx:RemoveChild/>` состояния.

И еще об эффектах действий

Для того чтобы научиться правильно использовать эффекты действий, нужно немного попрактиковаться. Это прекрасный инструмент, позволяющий осуществлять тонкий контроль за переходами в вашем приложении, поэтому стоит задуматься о том, каким образом они должны происходить. Чтобы определить, что и когда должно происходить, можно построить небольшую временную диаграмму каждого перехода. Она поможет сориентироваться в правильном расположении тегов, и все необходимые эффекты будут реализованы надлежащим образом.

В качестве еще одного примера использования эффектов действий можно попробовать использовать эффект Dissolve для компонента, добавляемого с помощью тега `<mx:AddChild/>` в определенном состоянии приложения. Для этого необходимо добавить в последовательность эффектов Sequence сначала тег `<mx:Dissolve/>`, а затем `<mx:AddChildAction/>`, что обеспечивает вначале появление дочернего элемента, а затем применение к нему эффекта Dissolve. В противном случае могла бы возникнуть ситуация, когда вначале будет отображен эффект, реализующий постепенное появление компонента. Затем компонент полностью исчезнет и вновь появится, как только произойдет добавление компонента (AddChild). В результате мы получим невразумительное мигание компонента и совершенно не вовремя отображающийся эффект Dissolve.

В следующем примере реализуется постепенное появление двух элементов Button, один за другим. Элемент Button с именем `button1` появляется перед элементом, названным `button2`.

```
<mx:Sequence targets="{[button1, button2]}">

  <mx:AddChildAction target="{button1}"/>
  <mx:Dissolve target="{button1}"/>

  <mx:AddChildAction target="{button2}"/>
  <mx:Dissolve target="{button2}"/>

</mx:Sequence>
```

Как правило, нельзя точно предугадать, в какой момент произойдет какое-либо действие при смене состояний, поэтому по возможности рекомендуется использовать специальные эффекты действия. Чтобы действие осуществлялось в надлежащее время, эффекты действия следует использовать внутри последовательности (Sequence) непосредственно перед визуальными эффектами, – это дает возможность полноценного управления применением свойств и добавлением или удалением дочерних элементов и позволяет получить запланированный результат.

Фильтры целевых объектов

При использовании переходов эффекты можно применять в зависимости от каких-либо условий. С помощью *фильтров целевых объектов* можно применять эффекты к определенным компонентам, отвечающим заданным критериям. Для использования таких фильтров необходимо определить все объекты, по отношению к которым будут применяться эффекты, а затем установить критерии, которым они должны отвечать при помощи свойства `filter` эффекта.

Существуют следующие возможные критерии фильтров:

add

Эффект применяется по отношению к любому компоненту, добавляемому в список отображения.

remove

Эффект применяется по отношению к компонентам, удаляемым из списка отображения.

show

Эффект применяется по отношению к компонентам при изменении их свойства `visible` с `false` на `true`.

hide

Эффект применяется по отношению к компонентам при изменении их свойства `visible` с `true` на `false`.

move

Эффект применяется по отношению к компонентам при изменении их свойств `x` или `y`.

resize

Эффект применяется по отношению к компонентам при изменении их свойств `width` или `height`.

Можно изменить переход в нашем приложении Search с использованием фильтров целевых объектов:

```
<mx:transitions>
  <mx:Transition fromState="search">

    <mx:Sequence targets="{[hbox1, resultsList]}">
```

```
        <mx:Parallel>
          <mx:Move filter="move" />
          <mx:Resize filter="resize" />
        </mx:Parallel>

        <mx:AddChildAction filter="add" />

      </mx:Sequence>

    </mx:Transition>
  </mx:transitions>
```

Теперь, вместо того чтобы определять объект для воздействия с помощью свойства `target` конкретного эффекта, можно перечислить все возможные целевые объекты для данного перехода, а затем определить фильтр для каждого типа эффекта. В данном примере потенциальными целевыми объектами перехода являются поля для ввода запроса и список с результатами поиска. Тогда при перемещении любого целевого объекта к нему будет применен эффект Move, а при изменении его размеров – эффект Resize; и, наконец, будут добавлены компоненты, для которых задано, что они должны быть добавлены.

Примечание

Определение контейнера как целевого объекта для некоторого эффекта не означает, что его дочерние элементы тоже устанавливаются в качестве целевых объектов для фильтров эффектов. Так, эффект Dissolve, применяемый по отношению к панели Panel, безусловно, будет воздействовать и на ее дочерние элементы, но попытка применить данный эффект к конкретному элементу TextInput внутри панели не увенчается успехом.

Фильтры

Вы когда-нибудь пользовались Photoshop или подобными графическими программами для обработки своих фотографий и добавления необычных эффектов? С помощью *фильтров* можно буквально преображать статичные изображения.

Так как многие из рассмотренных эффектов по сути используют такие же фильтры, то вам будет нетрудно догадаться, какой фильтр за что отвечает. Кроме того, свойства многих фильтров практически совпадают со свойствами соответствующих эффектов, поэтому вам не составит никакого труда освоить этот материал.

Ниже представлен список наиболее распространенных фильтров.

BevelFilter и GradientBevelFilter

Эти фильтры придают элементу объемный, рельефный вид (рис. 13.1). Фильтр GradientBevelFilter, кроме того, накладывает цветовой градиент, благодаря чему эффект выглядит более реалистично.

BlurFilter

Как и эффект Blur (использующий данный фильтр), придает компоненту размытый, расфокусированный вид. Пример его использования показан на рис. 13.2.

DropShadowFilter

Компонент, к которому применяется данный фильтр, отбрасывает тень, создавая впечатление, будто он слегка возвышается над остальными элементами приложения. Пример использования фильтра показан на рис. 13.4.

GlowFilter и GradientGlowFilter

Данные фильтры создают эффект свечения по краям компонента (рис. 13.3).

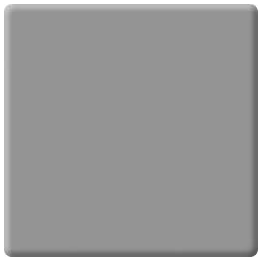

Рис. 13.1. *Серый квадрат с применением фильтра BevelFilter*

Рис. 13.2. *Серый квадрат с применением фильтра BlurFilter*

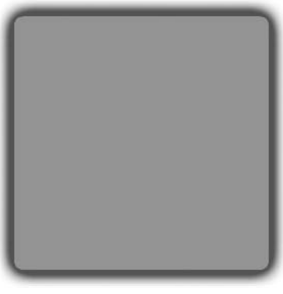

Рис. 13.3. *Серый квадрат с применением фильтра GlowFilter*

ColorMatrixFilter, ConvolutionFilter и DisplacementMapFilter

Использование этих трех фильтров сложнее остальных перечисленных в плане установки их свойств. Более подробную информацию о применении фильтров ColorMatrixFilter, ConvolutionFilter и DisplacementMapFilter можно найти в документации Flex. Богатые возможности этих фильтров позволяют полностью преобразить вид компонентов приложения. С помощью свойства DisplacementMapFilter можно даже придать целому приложению вид сферы!

Примечание

Фильтры можно использовать как для изменения визуального отображения компонентов, так и для создания определенного вида пользовательского интерфейса. К примеру, чтобы добиться более классического вида кнопки, можно применить к ней фильтр для придания объема. Чтобы ее выделение стало более заметным, можно использовать фильтр GlowFilter для события `rollOver` (и осуществлять возврат к прежнему состоянию при событии `rollOut`).

Применение фильтров

Для применения фильтров используется свойство `filters` компонента. Его значением может быть массив фильтров, поскольку можно применять несколько фильтров одновременно, чтобы добиться желаемого впечатления. Данное свойство удобнее всего присвоить компоненту в виде тега `<mx:filters/>`, поместив внутри него отдельные теги для каждого фильтра.

Снова откройте приложение `PhotoGallery` и попробуйте применить фильтры на практике. Например, можно применить фильтр DropShadowFilter по отношению к основному изображению, чтобы выделить его:

```
<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"
  completeEffect="Fade" >
  <mx:filters>
    <mx:DropShadowFilter distance="10"/>
  </mx:filters>
</mx:Image>
```

Рисунок 13.4 демонстрирует результат применения данного фильтра к приложению `PhotoGallery`.

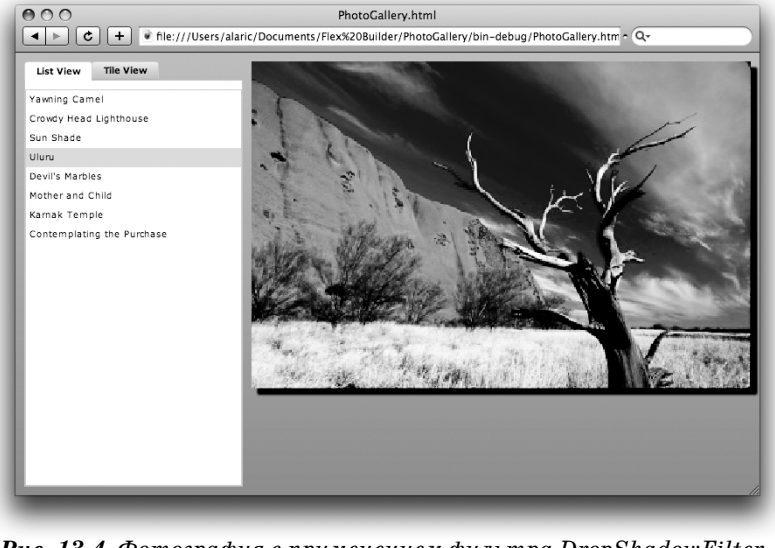

Рис. 13.4. *Фотография с применением фильтра DropShadowFilter в приложении PhotoGallery*

Раскрывая секреты мастерства

К сожалению, некоторые эффекты могут работать некорректно из-за способа отображения шрифтов устройства. Однако отчаиваться не стоит – существует эффективный прием, позволяющий решить эту проблему. Достаточно лишь применить фильтр по отношению к компоненту, и он отобразит шрифты устройства таким образом, что эффекты будут прекрасно работать. Дело в том, что текст, обработанный фильтром, воспринимается проигрывателем Flash Player как графическое изображение, что позволяет воздействовать с помощью эффектов даже на шрифты устройства.

Если при использовании этого приема вы не хотите, чтобы фильтр действительно изменил визуальное отображение компонентов, можно просто установить его настройки соответствующим образом. К примеру, я часто использую фильтр BlurFilter с нулевыми значениями свойств `blurX` и `blurY` для текстовых элементов, если их эффекты отображаются некорректно. То есть фильтр используется для нормализации работы эффектов, но не изменяет вид компонента, поскольку размытие равно нулю.

```
<mx:Button
  label="Fancy Button Text" >
  <mx:filters>
    <mx:BlurFilter blurX="0" blurY="0"/>
  </mx:filters>
</mx:Button>
```

Совместно с таким нехитрым приемом можно применять любые эффекты, не задумываясь о том, что использование шрифтов устройства может все испортить.

Примечание

Некоторые элементы управления и контейнеры обладают свойством `dropShadowEnabled`, с помощью которого можно создать тень компонента, не прибегая к использованию фильтра DropShadowFilter.

Большинство операций в процессе разработки Flex-приложений можно выполнять как с помощью MXML, так и с помощью ActionScript. Так же дело обстоит и с фильтрами. Только что рассмотренный пример добавления фильтра DropShadowFilter к элементу Image можно реализовать и средствами ActionScript. При этом используется следующая синтаксическая конструкция, создающая функцию, вызов которой осуществляется через некоторое время после того, как загрузится изображение в элемент Image, например при событии `creationComplete`:

```
var dropShadowFilter:DropShadowFilter = new DropShadowFilter();
dropShadowFilter.distance = 10;

image.filters = [dropShadowFilter];
```

Благодаря возможностям ActionScript фильтры можно применять и динамически во время работы приложения. В самом деле, с помощью нескольких несложных действий можно превратить приложение `PhotoGallery` в простую программу для обработки фотографий, с помощью которой пользователи смогут самостоятельно применять по отношению к изображениям различные фильтры.

Примечание

Для демонстрации еще одного доказательства широчайших возможностей Flex обратитесь по адресу *www.photoshop.com/express*. Там размещена бесплатная версия Adobe Photoshop, которая предоставляет, кроме всего прочего, различные фильтры для создания некоторых эффектов в изображении.

Заключение

В настоящей главе вы узнали о новых возможностях Flex, касающихся добавления интересных визуальных и звуковых эффектов. Вы применили некоторые из этих эффектов к созданным ранее приложениям, которые стали выглядеть еще привлекательнее для пользователя, – и все это с помощью нескольких строчек кода. Вы уже создали несколько приложений – не бойтесь экспериментировать с ними! Далее мы рассмотрим более тонкую настройку внешнего вида ваших Flex-приложений, чтобы они выглядели совершенно уникально.

14

В этой главе:
- Атрибуты стилей
- Использование таблиц стилей
- Встроенные ресурсы
- Использование скинов
- Темы оформления

Визуальное оформление приложения

Flex предоставляет разработчику отличный расширяемый набор компонентов. Их стандартный внешний вид определяется настройками по умолчанию, однако вовсе ими не ограничен: его можно легко изменять с помощью стилей. У разработчика есть возможность модифицирования стилевых свойств конкретного компонента и даже создания стилей, применяемых ко всем компонентам. С помощью каскадных таблиц стилей (CSS) можно создавать стили, позволяющие кардинально изменить внешний вид приложения; кроме того, их можно использовать многократно в разных приложениях. Визуальное отображение компонентов можно даже полностью изменить с помощью изображений в формате *.jpg* или *.gif*.

Атрибуты стилей

Вид компонента можно изменять с помощью стилей несколькими способами. Мы уже использовали некоторые свойства стилей при создании Flex-приложений. К примеру, мы изменяли вид элементов управления в приложении `ContactManager` с помощью свойств `fontWeight`, `fontSize` и `backgroundColor`. При этом были использованы *атрибуты стилей*, применяемые внутри MXML-тега компонента.

Стили проще всего применять в режиме редактирования Design среды разработки Flex Builder. В данном режиме вид Standard view панели Flex Properties содержит самые общие свойства стилей для конкретного компонента. Для применения таких сложных стилей, как градиентная заливка, эта панель – просто находка. При этом вам не нужно думать о деталях; достаточно лишь выбрать необходимые цвета с помощью выпадающей палитры или изменить иные свойства компонентов, например отображение шрифта, с помощью простых текстовых полей и всплы-

вающих списков. Не забывайте, стили отличаются от обычных свойств, несмотря на внешне одинаковый способ их применения в MXML-коде. В ActionScript для задания стилей применяются методы `setStyle()` и `getStyle()`.

Посмотрите на рисунок 14.1, на котором представлены самые общие стили. Набор возможных стилей различен для каждого конкретного компонента, но знакомство с этими основными свойствами обязательно вам пригодится в будущем.

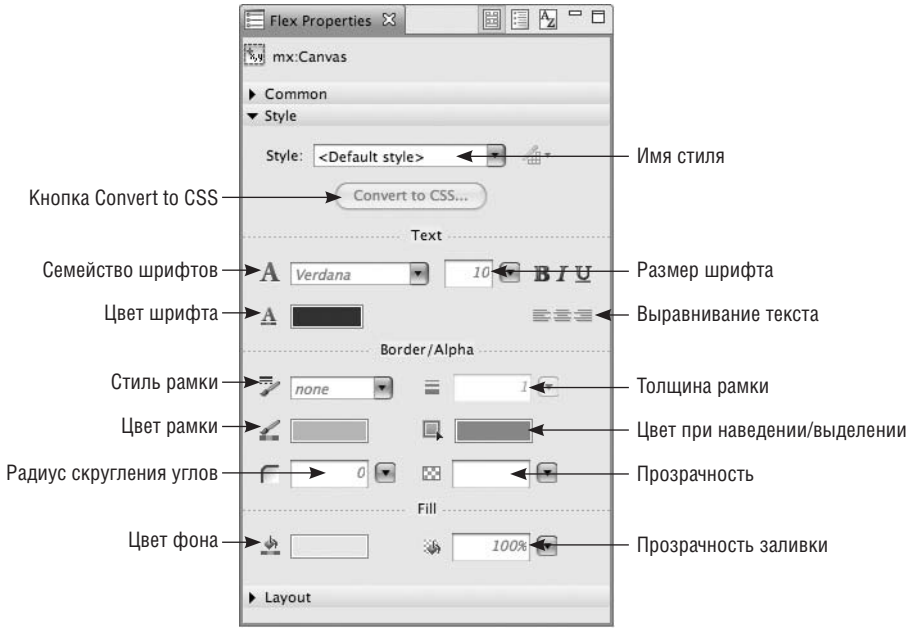

Рис. 14.1. *Раздел Style панели Flex Properties*

Вернемся к приложению `PhotoGallery`, над которым мы только что работали, и придадим ему совершенно новый внешний вид. Откройте основной файл приложения и перейдите в режим `Design`, который позволит нам быстро изменять стили.

Изменим окраску приложения на более темные тона. Для этого прежде всего нам предстоит изменить цвет фона приложения в теге Application. Это можно сделать с помощью свойства `backgroundColor`, однако данный тег обладает также свойством `backgroundGradientColors`, позволяющим создать *градиент*, т. е. плавный переход между несколькими цветами вместо одноцветного фона. Его можно установить с помощью раздела Fill панели Flex Properties. Вы выбираете цвета из всплывающих палитр, а Flex создает соответствующий массив цветов в коде приложения.

С помощью раздела Fill создайте градиент, осуществляющий переход от черного цвета (#000000) к темно-серому (#999999). При этом будет добавлено соответствующее значение атрибута backgroundGradientColors тега <mx:Application/>. Кроме того, существует настройка backgroundGradientAlphas, позволяющая регулировать прозрачность градиента. Ее удобно использовать для создания градиента, различные части которого должны иметь различную прозрачность. Данная настройка изменяется с помощью всплывающих бегунков, определяющих прозрачность заливки (Fill alpha) в том же разделе Fill.

Теперь тег <mx:Application/> вашего приложения будет выглядеть примерно следующим образом; результат внесенных изменений показан на рис. 14.2.

```
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  layout="absolute"
  applicationComplete="service.send()"
  backgroundGradientAlphas="[1.0, 1.0]"
  backgroundGradientColors="[#000000, #999999]">
```

Далее необходимо изменить оформление контейнера TabNavigator в соответствии с новой цветовой схемой. С помощью палитры Background color на панели Flex Properties установите черный цвет фона (#000000) для данного контейнера и обоих элементов управления списком. Для них нельзя использовать градиент, возможен только одноцветный фон. Поскольку фон всех этих компонентов по умолчанию заливается белым цветом, необходимо изменить эту настройку свойства backgroundColor и установить черный цвет фона для каждого из них.

После изменения цвета фона текст читается с трудом, поскольку цвет шрифта также темный. Значит, нужно изменить его оттенок на более

Рис. 14.2. *Приложение PhotoGallery с фоновой градиентной заливкой в темных тонах*

Каскадные стили

Свойство `color`, как и другие свойства стилей, действует по принципу каскада по отношению к дочерним элементам. Таков механизм *наследования* CSS – стиль, установленный для контейнера (или Application), передается его внутренним элементам, в данном случае окрасив их текст тем же цветом. Благодаря такой возможности можно определить цвет текста большинства компонентов, присвоив данное свойство тегу Application.

светлый. Цвет текста большинства элементов управления можно изменить с помощью инструмента для выбора цвета Font color панели Flex Properties, обозначенного значком, который выглядит как буква A с расположенной под ней разноцветной полоской. Его использование присваивает соответствующее значение свойству `color`.

Благодаря изменению цветов нескольких элементов приложение `PhotoGallery` теперь выглядит совсем по-другому (рис. 14.3).

Примечание

Чтобы определить, наследуется ли некоторое свойство стиля дочерними компонентами, обратитесь к соответствующему разделу в Flex Adobe Flex 3 Language Reference.

Однако цвет подсветки пунктов списка и фокусировки других компонентов все еще определяется настройками по умолчанию (что означает использование голубого оттенка). Их также можно изменить по своему усмотрению.

***Рис. 14.3.** Приложение PhotoGallery в темных тонах*

Для изменения цвета подсветки элементов списка можно использовать их свойство selectionColor, но существует и более эффективный способ – изменение свойства themeColor тега Application. Данное свойство может принимать значение любого цвета. С его помощью можно кардинально изменить внешний вид приложения, поскольку оно одновременно изменяет как свойство selectionColor элементов списка, так и цвет подсветки и фокусировки других элементов приложния. Его значение передается на столько уровней вложения, что достигает даже цвета заливки компонента ProgressBar. Поэтому остается лишь правильно выбрать цвет и любоваться результатом. Результат присваивания themeColor зеленого цвета показан на рис. 14.4.

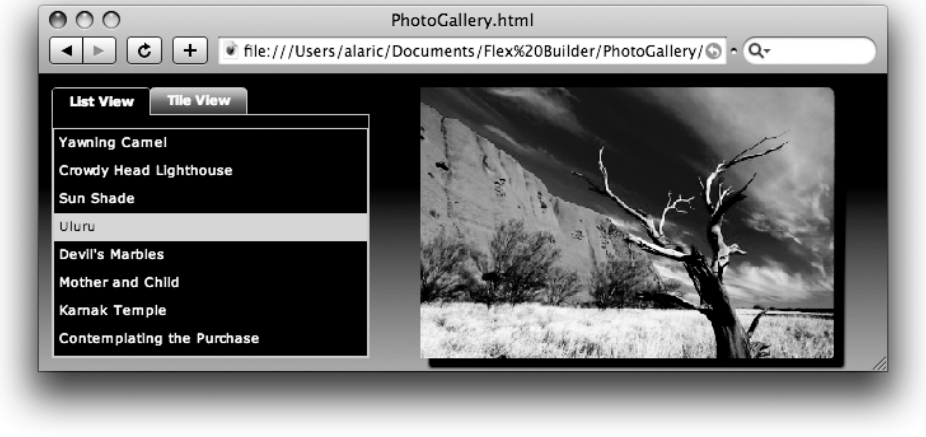

Рис. 14.4. *Приложение PhotoGallery с измененной темой зеленого цвета*

Названия цветов

Вам уже известно, что цвета можно представлять с помощью шестнадцатеричной системы счисления, но с этой целью можно использовать и действительные названия цветов. К примеру, задать черный цвет фона с помощью атрибута MXML-тега можно двумя способами:

```
backgroundColor="0x000000"
```

или

```
backgroundColor="black"
```

Второй способ удобнее при использовании основных цветов (таких как красный, зеленый, синий белый и черный (если считать черный цветом), но если вам требуются более сложные цвета, лучше будет выбрать цвет в шестнадцатеричном представлении

с помощью палитры. Flex позволяет использовать следующие названия цветов в качестве значений свойств: "black" (черный), "blue" (синий), "green" (зеленый), "gray" (серый), "silver" (серебристый), "lime" (светло-зеленый), "olive" (оливковый), "white" (белый), "yellow" (желтый), "maroon" (коричневый), "navy" (темно-синий), "red" (красный), "purple" (фиолетовый), "teal" (бирюзовый), "fuchsia" (розовый), "aqua" (цвет морской волны), "magenta" (пурпурный), "cyan" (голубой).

Кроме того, во Flex существует четыре особых цвета, называемых "haloOrange", "haloBlue", "haloSilver" и "haloGreen".

Использование таблиц стилей

Примечание

CSS – отдельный язык, не связанный с MXML и ActionScript. Он используется для создания таблиц стилей. В данной книге я не ставлю перед собой цель рассказать обо всех тонкостях CSS – мы поговорим об основных особенностях этого языка, знание которых поможет вам при моделировании внешнего вида ваших приложений.

Существует еще один способ применения стилей – использование тега <mx:Style/>. Он позволяет хранить все настройки стилей в отдельном месте. Таким образом, вместо того чтобы задавать стили для каждого конкретного компонента внутри соответствующего тега, можно с помощью CSS создать определения стилей, которые могут быть использованы неоднократно различными компонентами. Такая технология напоминает применение стилей в программах для обработки текстов, когда вместо выделения каждого заголовка документа определенным цветом и задания параметров его шрифта можно создать стиль, применяемый ко всем заголовкам. Если у вас есть опыт такой работы, то вам прекрасно известно, сколько времени экономит применение этой технологии при необходимости, скажем, изменения определенного стиля, так как достаточно внести только в его описание необходимые изменения, и все текстовые элементы, к которым он применяется, будут обновлены автоматически. В контексте разработки Flex-приложений это означает, что можно сохранить однажды заданные значения, например таких свойств, как цвет и размер шрифта, цвет фона, радиус скругления и т. д., и многократно использовать их в приложении.

CSS-код размещается в приложении внутри тега <mx:Style/>. Его можно также поместить во внешний файл с расширением *.css* (подробнее о таком способе читайте в разделе «Таблицы стилей во внешнем файле»).

Синтаксис CSS

Примечание

Использование фигурных скобок может навести вас на мысль о сходстве с ActionScript, но это не так: CSS – совершенно другой язык (описательный, как и MXML, в то время как ActionScript – язык процедурный). Обратите внимание, например, на использование двоеточия (:) вместо знака равенства (=) для присваивания значения свойству.

В основе синтаксиса CSS лежит правило стиля, в которое входит имя стиля, за которым следует описание стиля, заключенное в фигурные скобки. Описание состоит из свойства стиля, за которым следует его значение, отделенное двоеточием. Описания разделяются точкой с запятой и, как правило, располагаются по одному на каждой строке (рис. 14.5).

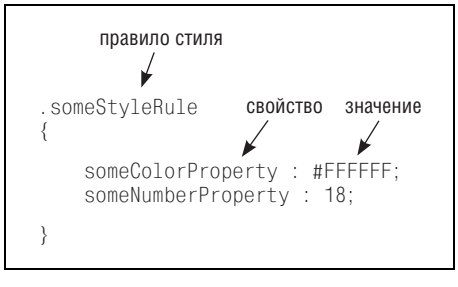

***Рис. 14.5.** Синтаксис CSS*

Описания CSS-стилей можно создавать тремя способами: с помощью *селекторов класса*, *селекторов типа* и *глобальных стилей*.

Селекторы класса

Селекторы класса – это правила стилей с определенным именем, которые могут применяться по отношению к любому компоненту. Они позволяют сгруппировать настройки стилей и присвоить их компоненту. К примеру, можно создать правило стиля с именем `greatWhite`, задающее белый цвет и 18-й размер шрифта. Согласно правилам построения селекторов класса его имени должна предшествовать точка. CSS-код этого примера выглядит так:

```
<mx:Style>
.greatWhite
{
  color: #FFFFFF;
  fontSize: 18;
}
</mx:Style>
```

Чтобы присвоить этот стиль определенному компоненту, необходимо задать ему свойство styleName со значением, соответствующим имени стиля (без точки). К примеру, чтобы присвоить стиль greatWhite кнопке Button, можно использовать следующий MXML-код:

```
<mx:Button styleName="greatWhite" />
```

Примечание

Если вы имеете опыт совместного использования CSS и HTML, то наверняка помните, что в HTML атрибут, используемый для присваивания стилей, определенных селектором класса, называется class; во Flex соответствующий атрибут называется styleName.

Имена свойств в CSS

В данной главе вы убедились, что имена CSS-стилей соответствуют именам свойств из категории стилей отдельных компонентов. Таким образом, можно задать для элемента Button размер шрифта в теге с помощью атрибута fontSize="14" или используя следующий CSS-код:

```
Button
{
  fontSize: 14;
}
```

Однако на самом деле при использовании CSS в разработке Flex-приложений возможны два варианта наименования. Мы уже применяли способ, основанный на использовании «верблюжьего регистра», как в вышеприведенном примере. Но, как правило, в стандартных таблицах стилей, используемых на веб-страницах, применяется другой способ наименования свойств, в котором смысловые части разделяются с помощью дефисов, а не заглавных букв.

Поэтому следующая запись также верна:

```
Button
{
  font-size: 14;
  font-weight: bold;
  corner-radius: 7;
}
```

При создании таблицы стилей можно задавать имена свойств любым способом, но при определении стилей компонента с помощью атрибутов возможно использование только «верблюжьего регистра».

Селекторы типа

Принципы работы селекторов класса и селекторов типа сильно различаются. Селекторы типа позволяют определить стиль для всех компонентов определенного вида. С их помощью можно создать набор стилей для применения по отношению ко всем элементам Button или, скажем, Panel, в вашем приложении. В определении селектора типа необходимо указать имя компонента:

```
<mx:Style>
Button
{
  color: #FFFFFF;
  fontSize: 18;
}
</mx:Style>
```

Данный код задает белый цвет и 18-й размер шрифта для всех элементов Button приложения, при этом нет необходимости устанавливать какие-либо дополнительные свойства конкретных объектов Button.

Глобальные стили

С помощью глобальных стилей можно применить определенные стили к приложению в целом. При применении селекторов типа большинство стилей (но не все) будут наследоваться дочерними элементами (подробнее о наследовании во врезке «Каскадные стили»). Использование же глобального стиля предполагает наследование данных свойств всеми компонентами приложения.

В качестве имени правила глобального стиля используется слово `global`. Ниже приведен код, реализующий присваивание определенного цвета и размера шрифта всем компонентам приложения:

```
<mx:Style>
global
{
  color: #FFFFFF;
  fontSize: 18;
}
</mx:Style>
```

Таблицы стилей во внешнем файле

Таблицы стилей можно создавать во внешнем файле, который затем подключается к Flex-приложению при помощи свойства `source` тега `<mx:Style/>`. Это позволяет хранить настройки стилей отдельно, благодаря чему становится возможным, во-первых, подключать к одному и тому же приложению различные файлы стилей с расширением *.css*, полностью изменяя его внешний вид. Вы можете создать несколько различных файлов таблиц стилей и менять их – достаточно одной

строчки MXML-кода для полного преображения приложения. Во-вторых, при работе в команде это позволяет разделить обязанности программиста и дизайнера, так что каждый может работать в отдельном файле – дизайнер с файлом стилей, а программист – с основным кодом приложения.

Примечание

Flex наследует многие (но не все) атрибуты CSS, используемые совместно с HTML, поэтому во Flex-приложениях часто можно использовать таблицы стилей, написанные для веб-страниц.

Создание нового CSS-файла

Если вы хорошо знакомы с технологией CSS или собираетесь начать это знакомство, начните с создания чистого файла с расширением *.css*, который сразу же будет доступен для использования в вашем приложении. Для этого нужно выделить проект на панели Flex Navigator и выбрать команду меню File→New→CSS File. Откроется диалоговое окно, запрашивающее имя нового файла (рис. 14.6).

По окончании щелкните по кнопке Finish; при этом во Flex Builder откроется чистый файл с расширением *.css*, который можно редактировать как путем написания CSS-кода вручную, так и с помощью режима Design.

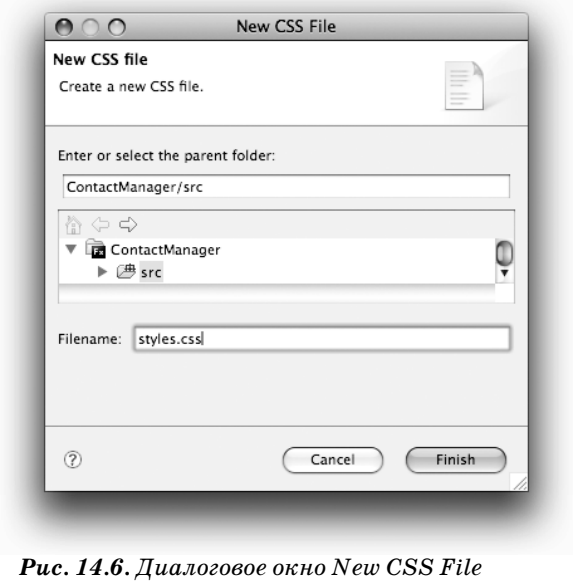

Рис. 14.6. *Диалоговое окно New CSS File*

Теперь вы можете попробовать изменить оформление приложения ContactManager, используя таблицы стилей CSS и тег <mx:Style/>.

Я создал CSS-файл, с помощью которого можно придать новый вид данному приложению. Его можно загрузить с *www.greenlike.com/flex/learning/projects/contactmanager/styles.css*. Разместите данный файл в папке с исходным кодом проекта ContactManager.

Затем можно добавить в приложение тег <mx:Style/>, импортирующий в него данную таблицу стилей:

```
<mx:Style source="styles.css"/>
```

CSS-файл, загружаемый с помощью тега <mx:Style/>, компилируется во Flex-приложение при сборке, в то время как HTML позволяет подключать внешние файлы стилей при загрузке страницы. Иными словами, при работе с Flex-приложениями не имеет смысла заменять расположенный на сервере CSS-файл другим файлом стилей – для его изменения придется перекомпилировать все приложение. Однако Flex предоставляет возможность загрузки стилей в момент запуска приложения, что реализуется с помощью компиляции таблицы стилей в файл в формате *.swf* и ее загрузки при запуске приложения. Более подробную информацию об этом способе можно найти в документации Flex, введя запрос *runtime styles* в окно поиска.

Разместите тег <mx:Style/> в верхней части кода, как вы ранее поступали с тегами <mx:Script/> и тегами эффектов. Это заставит приложение импортировать этот CSS-файл. При работе в режиме Design вы сразу заметите внесенные изменения. Теперь приложение ContactManager будет выглядеть примерно следующим образом (рис. 14.7).

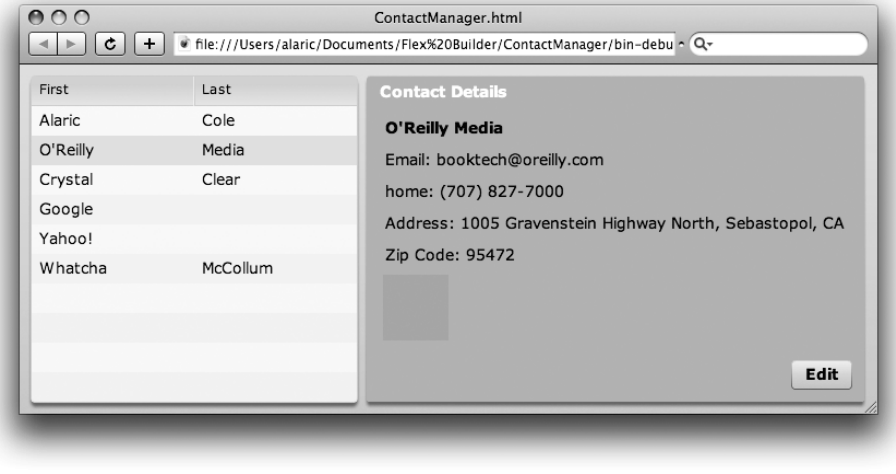

Рис. 14.7. *Приложение ContactManager, использующее новый файл таблицы стилей*

Теперь вы на практике смогли убедиться в многочисленных достоинствах использования внешних CSS-файлов – сейчас вы, по сути, полностью изменили внешний вид приложения, добавив одну-единственную строчку кода. Таким образом можно подключать различные файлы таблиц стилей, найденные в сети или созданные вами самостоятельно, и приложение будет каждый раз выглядеть по-разному.

Код файла *styles.css* представлен ниже.

```
global
{
  backgroundAlpha: .7;
  fontSize: 14;
}
Application
{
  backgroundColor: haloSilver;
  themeColor: #69a6fa;
}
DataGrid
{
  alternatingItemColors: #F7F7F7, #E2E8F4;
  backgroundDisabledColor: #C4DFF4;
  dropShadowEnabled: true;
  headerColors: #96BEF4, #C7DCF9;
  headerStyleName: dataGridHeader;
  horizontalGridLines: false;
  verticalGridLines: false;
}
.dataGridHeader
{
  fontSize: 12;
  color: #474545;
}
Panel
{
  backgroundAlpha: 1.0;
  backgroundColor: #6B9BC8;
  borderAlpha: 1.0;
  borderColor: #6B9BC8;
  controlBarStyleName: panelControlBar;
  cornerRadius: 3;
  dropShadowEnabled: true;
  titleStyleName: panelTitle;
}
.panelControlBar
{
  horizontalAlign: right;
}
```

```
.panelTitle
{
  color: #FFFFFF;
  fontWeight: bold;
  fontSize: 18;
}
TextInput
{
  cornerRadius: 7;
  borderStyle: solid;
  backgroundAlpha: 0.8;
}
```

В данном CSS-файле используются как селекторы класса, так и селекторы типа. Преобладание последних объясняется тем, что данная таблица стилей предназначена для изменения оформления приложения в целом, без добавления дополнительного MXML-кода, за исключением тега, импортируюшего сам файл. Однако также используются и селекторы класса – для присвоения определенных стилей компонентам, которые требуют задания сложных стилей. К примеру, компонент Panel обладает свойством `titleStyleName`, определяющим вид текста, отображаемого в строке его заголовка. Поскольку это свойство подразумевает возможность использования настроек нескольких стилей, например размера и цвета шрифта, с помощью селектора класса создается правило стиля с именем `.panelTitle`, на которое затем ссылается `titleStyleName`.

А теперь, когда у вас есть отдельный файл с таблицей стилей, вы можете открыть для себя еще одну невероятную возможность Flex Builder – CSS-файлы можно редактировать визуально в режиме Design, как и MXML-код. Откройте файл *styles.css* и перейдите в режим Design. На нанели инструментов рядом с переключателем режимов Source/Design и кнопкой Refresh появится несколько новых кнопок (рис. 14.8).

С помощью этих кнопок можно выбирать, создавать и удалять стили. Попробуйте выбрать правило стиля для компонента Panel из выпадающего списка Style – при этом отобразится образец панели с заданными свойствами. При изменении свойств этот образец обновляется автоматически, позволяя разработчику быстро настраивать необходимые стили в визуальном режиме.

***Рис. 14.8.** Панель инструментов для редактирования CSS-файлов в режиме Design*

Панель Flex Properies при редактировании таблиц стилей выглядит примерно так же, как и при редактировании MXML-кода, разве что в ней представлен более полный список свойств. При переключении в вид Standard view у вас под рукой буквально все возможные стили. Для модификации стилей, требующих задания правила стиля, таких как свойство `titleStyleName` компонента Panel, используется кнопка Edit, позволяющая редактировать именно данное правило стиля.

Для редактирования файлов таблиц стилей в режиме Design не требуется особых знаний CSS, поскольку большинство операций выполняются в визуальном режиме. Меняйте свойства в этой таблице стилей в свое удовольствие. Затем переключитесь в режим Source и посмотрите, какие изменения были внесены в код, – это хороший способ изучения CSS.

Преобразование атрибутов стилей в CSS

Существует способ вынесения стилей, указанных в качестве атрибутов элемента, во внешний файл таблицы стилей. Прежде всего необходимо выделить компонент в режиме Design. В разделе Style панели Flex Properties вы увидите кнопку Convert to CSS (рис. 14.1). При щелчке на этой кнопке открывается диалоговое окно New Style Rule (рис. 14.9), с помощью которого можно присоединить новый файл таблицы стилей и добавить в него правила стиля. (Если Flex Builder предлагает вам вначале сохранить приложение, последуйте этому совету.)

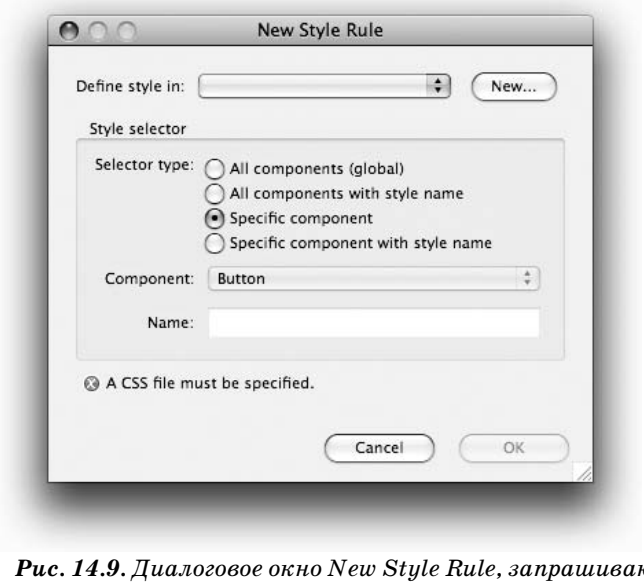

Рис. 14.9. *Диалоговое окно New Style Rule, запрашивающее CSS-файл*

Если данное приложение не использует внешний файл таблицы стилей, будет выведено оповещение «A CSS file must be specified» («Необходимо задать CSS-файл»). Вы можете создать его тут же, нажав на кнопку New в правом верхнем углу.

При этом откроется диалоговое окно New CSS File, с помощью которого можно создать новый файл с расширением *.css*, как показано на рис. 14.6. Данный файл автоматически отобразится в диалоговом окне New Style Rule (рис. 14.10).

Рис. 14.10. *Создание селектора типа для компонента Button с помощью диалогового окна New Style Rule*

Вы можете создавать стили разных типов. К примеру, для создания селектора типа, изменяющего вид всех компонентов Button, использующих данную таблицу стилей, выберите тип селектора Specific component. Для создания селектора класса используется опция All Components with style name. При нажатии на кнопку Finish создается правило CSS и открывается новый файл с расширением *.css*.

Примечание

Для создания нового правила стиля используется кнопка New Style на панели инструментов (рис. 14.8). Она вызывает диалоговое окно New Style Rule. Подробнее о создании новых правил стилей говорится во врезке «Преобразование атрибутов стилей в CSS».

Приоритет стилей

Стили можно применять различными способами, имеющими различный приоритет (рис. 14.11). Самым высоким приоритетом обладают стили, присваиваемые компоненту с помощью атрибутов, – их значение превалирует даже в том случае, когда для него задан селектор типа или класса. Селекторы класса имеют преимущество перед селекторами типов. Это означает, что если с помощью селектора типа установлены определенные настройки для всех элементов Button, значение свойства `styleName` конкретной кнопки будет иметь больший приоритет и именно оно будет использоваться для определения ее внешнего вида, а не селектор типа.

Рис. 14.11. *Приоритеты стилей во Flex*

Чтобы было легче запомнить иерархию стилей, можно сформулировать общее правило: во-первых, стили, определенные для конкретного экземпляра компонента, имеют самый высокий приоритет, и во-вторых, конкретные свойства стилей всегда перевесят более общие свойства, заданные в атрибуте `styleName`.

Встроенные ресурсы

Сейчас мы будем учиться встраивать внешние ресурсы в приложение на примере программы Search, а заодно применим к ней новые стили. Откройте файл *Search.mxml* в режиме Design и перейдите к его основному состоянию. Затем выделите контейнер Application (в режиме Design для этого достаточно щелкнуть мышью по фону приложения) и примените к нему новый стиль с помощью панели Flex Properties – сделайте заливку фона новым градиентом. Для этого приложения прекрасно подойдет оформление в серых тонах, поэтому можно создать градиент,

осуществляющий переход от белого к серому цвету (например, от 0xFFFFFF к 0xA1A1A1). Чтобы оформление приложения выглядело более гармонично, можно установить в качестве значения свойства `themeColor` серый или серебристый цвет.

Примечание

Существует отличный визуальный инструмент Flex Style Explorer, с помощью которого можно быстро и без особого труда сгенерировать CSS-код для ваших компонентов. Он доступен в режиме он-лайн по адресу *http://examples.adobe.com/flex3/consulting/styleexplorer/Flex3StyleExplorer.html*. Безусловно, набор инструментов Flex Builder обладает более широкими возможностями, но Flex Style Explorer позволяет существенно сэкономить время и может быть особенно полезен, если вы не используете среду Flex Builder для разработки приложений.

Иконки

Один из стилей, доступный определенным компонентам, обеспечивает отображение небольшой иконки. Например, отображение иконки доступно для кнопок – через свойство `icon`: иконка размещается внутри кнопки, по умолчанию ближе к ее левому краю. Однако, чтобы добавить иконку в кнопку или другой компонент, необходимо встроить в приложение соответствующее графическое изображение, т. к. иконка не загружается во время исполнения. Проще всего встроить в приложение дополнительные ресурсы с помощью нанели Flex Properties в режиме Design. Для этого используется специальная команда компилятора.

Для иконки требуется графическое изображение. Вы можете выбрать любую картинку, которая вам по душе, или загрузить созданное мной изображение лупы с моего сайта *www.greenlike.com/flex/learning/projects/search/search_icon.png*. Поместите выбранный файл с изображением в папку с исходным кодом проекта `Search`.

При выделении кнопки на панели Flex Properties в разделе `Common`, содержащем перечень наиболее часто используемых свойств, появляется поле `Icon`. Чтобы добавить необходимый код для встраивания иконки, щелкните по пиктограмме с изображением папки, находящейся справа. При этом откроется диалоговое окно `Open`, в котором можно указать путь к соответствующему встраиваемому изображению. Укажите в структуре папок путь к файлу *search_icon.png* или к своему собственному изображению. По завершении работы с диалоговым окном вид приложения во Flex Builder будет обновлен, и в режиме `Design` на кнопке отобразится соответствующая иконка (рис. 14.12).

Примечание

В контейнере навигации TabNavigator на вкладках также можно отображать иконки. Для задания необходимого изображения используется свойство `icon` соответствующего дочернего контейнера, – аналогично заданию значения свойства `label` для отображения на вкладке.

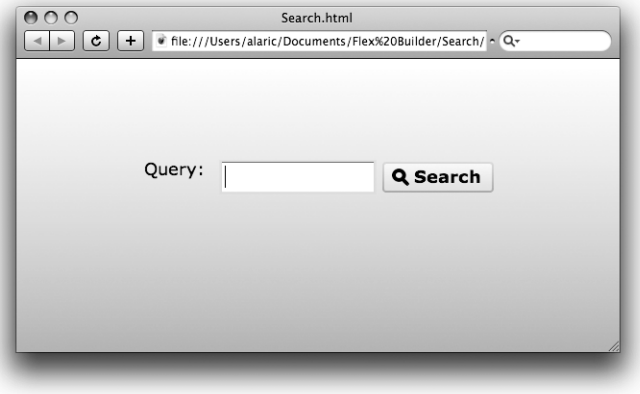

Рис. 14.12. *Использование иконки в приложении Search*

В сгенерированном при добавлении иконки в режиме Design коде вы увидите выражение `@Embed`, сопровождаемое указанием источника для встраиваемой иконки, заключенного в скобки.

```
icon="@Embed(source='search_icon.png')"
```

Таким образом можно встраивать внешние ресурсы с помощью атрибутов MXML; та же синтаксическая конструкция используется для встраивания дополнительных ресурсов других типов.

Звуки

В приложение, точно так же, как и изображения, можно встраивать звуковые файлы. В предыдущей главе были рассмотрены возможности применения звуковых эффектов, и я упоминал о возможном замедлении в работе приложения при загрузке звуковых файлов. Эту проблему можно решить, встроив MP3-файл прямо во Flex-приложение.

Попробуем встроить звуковой файл для создания эффекта в приложение `Search`. Откройте его в режиме редактирования Source и посмотрите на код созданных ранее переходов. В нем реализуется одновременное (Parallel) выполнение эффектов Move и Resize. К ним можно добавить эффект для создания звукового сопровождения при отображении этих эффектов.

Для этого нам понадобится файл в формате MP3. Вы можете использовать любой звуковой файл на ваш вкус или загрузить мой файл, расположенный по адресу *www.greenlike.com/flex/learning/projects/search/whoosh.mp3*. Поместите его в папку с исходным кодом проекта. Тег `<mx:SoundEffect/>` обладает свойством `source`, в качестве значения которого в данном случае следует указать выражение `@Embed` – аналогично уже использованному нами в примере с иконкой для кнопки. Для установки звукового эффекта можно использовать следующий код:

```
<mx:Transition fromState="search">
  <mx:Sequence targets="{[frm, queryTextInput, searchButton,hbox1,
  resultsList]}">
    <mx:Parallel>
      <mx:Move filter="move" />
      <mx:Resize filter="resize" />
      <mx:SoundEffect source="@Embed(source='whoosh.mp3')"/>
    </mx:Parallel>
    <mx:AddChildAction filter="add"/>
  </mx:Sequence>
</mx:Transition>
```

Теперь каждый раз перемещение полей для поиска при переходе из состояния search в основное состояние будет сопровождаться звуковым эффектом.

Примечание

Звук будет проигрываться только во время выполнения последовательности эффектов, а длительность этого процесса может оказаться меньше, чем необходимо. Продолжительность звукового эффекта (свойство `duration`) не устанавливается автоматически равной длительности встроенного MP3-файла, поэтому рекомендуется задать свойство `duration` вручную (в миллисекундах).

Шрифты

В предыдущей главе я уже говорил о возможности встраивания шрифтов. По умолчанию Flex использует системные шрифты (шрифты устройства), но разработчик может использовать и свои шрифты. Проще всего встроить шрифт с помощью CSS-описания в режиме Design. При встраивании своих шрифтов я рекомендую создать внешний файл таблицы стилей и использовать режим Design для автоматического написания необходимого кода.

Код встраивания шрифта Arial может выглядеть примерно так (в таблице стилей используется определение `@font-face`):

```
@font-face
{
  src:local("Arial");
  fontFamily: myEmbeddedFont;
}
```

Примечание

Многие шрифты защищены лицензией от несакционированного распространения (в том числе от встраивания в приложения Flex). Прежде чем использовать конкретный шрифт для встраивания в ваше приложение, ознакомьтесь с правилами его распространения.

С помощью данного кода происходит встраивание шрифта Arial из системы разработчика; в дальнейшем его можно использовать в приложении, ссылаясь на него по имени myEmbeddedFont. Его можно устанавливать в качестве значения свойства `fontFamily` конкретного компонента. Следующий код определяет данный шрифт в качестве шрифта по умолчанию, используемого во всем приложении:

```
Application
{
  fontFamily: myEmbeddedFont;
}
```

Примечание

Использование встроенных шрифтов имеет некоторые недостатки, такие как увеличение размера файла приложения и возможное возникновение проблем с их читаемостью при изменении размера шрифта, поэтому стоит использовать эту возможность только при действительной необходимости. Например, если используемые вами эффекты основаны на встроенных шрифтах или если данный шрифт является неотъемлемой частью дизайна приложения и при этом вряд ли установлен в системе большинства пользователей.

Использование скинов

Возможности применения стилей для Flex-приложений весьма обширны, но в некоторых случаях и их может оказаться недостаточно. Если вы хотите придать элементу CheckBox вид выключателя света или использовать кнопки нестандартной формы, стилями тут не обойтись. В этом случае для придания компоненту определенного вида можно использовать графические изображения.

Внешнее оформление любого визуального компонента Flex можно модифицировать с помощью графики. Этот метод называется сменой *скинов* (*skin* буквально – кожа), поскольку изменение внешнего вида никак не затрагивает основные функции компонента. С помощью скинов можно модифицировать визуальное оформление компонента.

Большинство компонентов имеют несколько скинов, представляющих его различные части или разный вид в различных состояниях. Они определяются с помощью свойств из категории стилей. Как правило, названия этих свойств оканчиваются на *Skin*: к примеру, свойство `borderSkin` панели Panel позволяет изменять графическое отображение ее рамки, а `titleBackgroundSkin` отвечает за строку заголовка. Элемент Button обладает свойством `upSkin`, определяющим его вид в обычном состоянии, и свойством `downSkin`, изменяющим его при нажатии, `overSkin` – при наведении на него указателя мыши и `disabledSkin`, указывающим на его отображение при значении `false` его свойства `enabled`.

К примеру, кнопка может использовать четыре разных скина в виде четырех изображений в формате PNG для отображения в различных состояниях. Допустим, у нас есть четыре подходящих файла с названиями *up.png*, *over.png*, *down.png*, и *disabled.png*; тогда для создания различных скинов кнопки можно использовать следующий код:

```
<mx:Button
  upSkin="@Embed(source='up.png')"
  overSkin="@Embed(source='over.png')"
  downSkin="@Embed(source='down.png')"
  disabledSkin="@Embed(source='disabled.png')" />
```

В данном примере также используется выражение `@Embed`, применяемое для встраивания в приложение внешних ресурсов, таких как звуки и графика.

Примечание

Кнопке присуще также свойство `skin`, определяющее единое изображение для ее отображения в любом состоянии.

При использовании скинов для определения внешнего вида компонента целесообразно установить их для любого возможного состояния компонента. Это позволяет контролировать все аспекты его оформления, ведь если задать кнопке Button скин только с помощью свойства `upSkin`, при наведении указателя мыши на нее появится изображение, используемое по умолчанию, а такое несоответствие выглядит не слишком эстетично.

Перечень всех доступных свойств для использования скинов для каждого конкретного компонента можно найти в виде Category view панели Flex Properties. Однако для изменения оформления компонентов с помощью скинов предпочтительнее всего использовать внешний файл таблицы стилей, поскольку в этом случае будут доступны дополнительные опции из раздела Skin при редактировании этого файла в режиме Design. Так, при выделении правила стиля обратите внимание на переключатель Style/Skin, расположенный в правом верхнем углу панели Flex Properties (рис. 14.13). С его помощью можно перейти в раздел Skin, в котором можно легко встраивать графические ресурсы, используя выпадающий список Skin.

Вы можете использовать как стандартные графические файлы с расширениями *.png, .gif, or .jpg*, так и скомпилированные файлы *.swf или .swc*. Таким образом, вам доступны ресурсы из различных источников. При выборе типа файла откроется диалоговое окно со списком всех доступных скинов, в котором можно выбрать необходимый файл или Flash-символ для импорта, а весь необходимый код будет написан автоматически. В режиме Design можно также увидеть, как будет отображаться данный скин. Для компонентов типа Button может быть отображен вид во всех возможных состояниях, как показано на рис. 14.14.

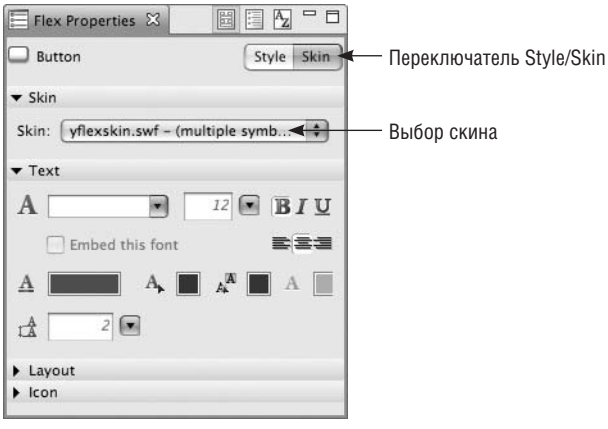

Рис. 14.13. *Вид панели Flex Properties при редактировании CSS-файла в режиме Design*

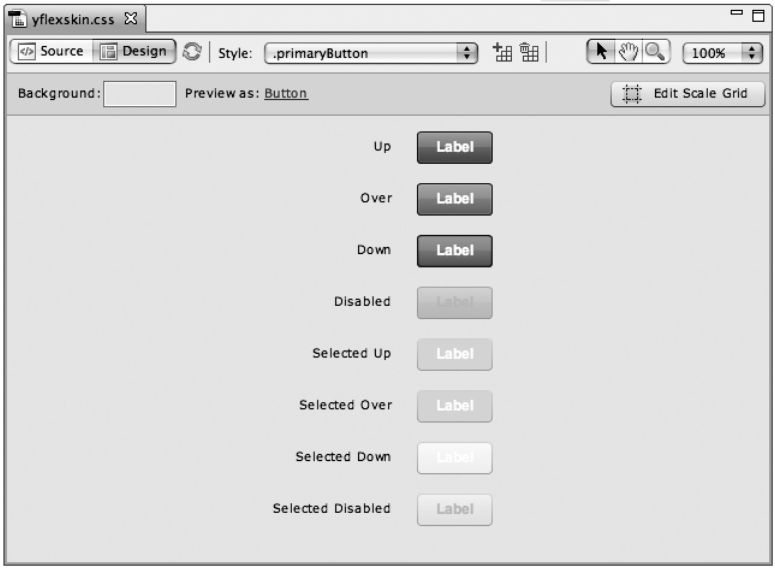

Рис. 14.14. *Скины компонента Button, вид в режиме Design*

Масштабирование скина

При редактировании CSS-файлов в режиме Design под рукой у разработчика есть еще одна полезная функция – возможность редактирования сетки масштабирования компонентов, т. е. невидимой сетки, с помощью которой можно определить, какие части скина необходимо масштабировать при изменении размера компонента, а какие – нет.

К примеру, скин вашей кнопки имеет закругленные края, и вам не нужно менять параметры скругления при увеличении размеров кнопки, а нужно только изменить размер внутренней части скина. Именно это позволяет реализовать сетка масштабирования (рис. 14.15 и 14.16). Все, что находится за пределами ограниченного условными линиями прямоугольника, останется неизменным, а заключенная в него часть будет масштабирована. На рис. 14.17 показан вид инструмента для редактирования сетки масштабирования.

Рис. 14.15. *Изменение размера скина кнопки по горизонтали без использования сетки масштабирования*

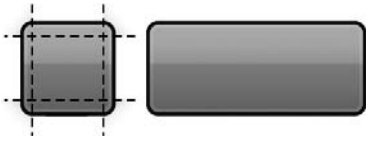

Рис. 14.16. *Изменение размера скина кнопки по горизонтали с использованием сетки масштабирования*

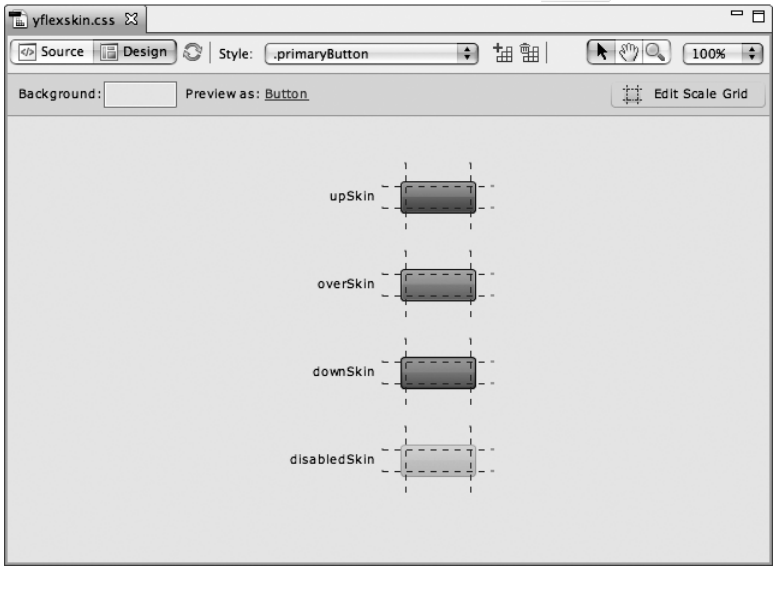

Рис. 14.17. *Редактирование сетки масштабирования скина компонента Button*

Примечание

Элемент Button может отображать в различных состояниях как различные скины, так и иконки. Его свойство icon определяет единую иконку, которая будет показана в любом состоянии. Однако с помощью режима редактирования Design для CSS можно установить различные иконки для различных состояний кнопки (например, в обычном состоянии, при наведении на нее указателя мыши или заблокированном состоянии).

Темы оформления

Flex позволяет изменять внешний вид приложения в целом, а не только его отдельных частей, с помощью стилей или скинов, а чаще всего – сочетания этих способов. Помещая некоторый набор настроек и/или скинов в отдельный файл таблицы стилей, вы создаете *тему оформления* для приложения.

Созданные темы можно применять к различным проектам или распространять в сети Интернет, что дает возможность их использования другими пользователями. В сети можно найти немало полезных тем, подходящих для ваших приложений. Поскольку при создании тем используются дополнительные ресурсы, они, как правило, включают в себя как файл в формате CSS, так и графические изображения с расширением *.png* или *.swf*. Чтобы применить такую тему к своему приложению, нужно скопировать данные файлы в папку с исходным кодом вашего приложения и указать в качестве значения свойства source тега <mx:Style/> путь к CSS-файлу темы.

Некоторые темы распространяются в виде одного скомпилированного файла с расширением *.swc*, содержащего как CSS-код, так и необходимые дополнительные внешние ресурсы. Для применения таких тем в своих приложениях необходимо использовать специальную возможность компилятора. Для этого выделите ваш проект и выберите Project→ Properties. В открывшемся диалоговом окне выберите пункт Flex Compiler из списка слева, чтобы перейти в раздел настроек компилятора Flex. Введите в поле Additional compiler arguments аргумент -theme и далее название файла вашей темы с расширением *.swc* (рис. 14.18).

К примеру, Flex Builder использует тему оформления Halo Classic, путь к которой указан ниже:

Mac:

/Mac/Applications/Adobe Flex Builder 3/sdks/3.0.0/frameworks/themes/HaloClassic/haloclassic.swc

Windows:

C:\Program Files\Adobe Flex Builder 3\sdks\3.0.0\frameworks\themes\HaloClassic\haloclassic.swc

Рис. 14.18. *Раздел Flex Compiler диалогового окна Properties*

Этот SWC-файл можно скопировать в папку вашего проекта, указать соответствующий аргумент компилятора (`-theme`), и эта тема будет использована для оформления ваших приложений.

Примечание

Если вы хотите освоить основы рисования для Flash Player, поищите информацию о Drawing API в документации Flex. Это позволит вам рисовать любые компоненты с использованием команд векторной графики. Вы сможете даже создавать программным путем свои собственные скины.

Примеры приложений с использованием различных тем оформления представлены на рис. 14.19–14.21.

Примечание

Множество тем, с помощью которых можно мгновенно преобразить ваше приложение, доступно в режиме он-лайн по адресу *www.scalenine.com*.

Тема оформления, используемая по умолчанию

Flex предоставляет разработчику встроенную тему оформления, называемую Halo Aeon. В ее основе лежит принцип программного создания скинов, т. е. скины создаются путем рисования средствами ActionScript, а не выбором файлов с расширением *.jpg* или *.png*. Именно благодаря такому механизму вид стандартных компонентов Flex можно изменять с помощью множества стилевых свойств, таких как `cornerRadius`, `borderStyle` или `backgroundColor`. Их использование становится возможным, поскольку скины компонентов создаются посредством скриптов.

Тема оформления, используемая Flex по умолчанию, содержится в файле с расширением *.css*, и если вы не меняли настройки расположения файлов при установке Flex Builder, этот файл можно найти в следующем месте:

Mac

/Mac/Applications/Adobe Flex Builder 3/sdks/3.0.0/frameworks/projects/framework/defaults.css

Windows

C:\Program Files\Adobe Flex Builder 3\sdks\3.0.0\frameworks\projects\framework\defaults.css

Если вам интересно, каким образом данный CSS-код влияет на отображение Flex-компонентов, не поленитесь изучить этот файл.

При установке Flex Builder поставляется еще одна тема по умолчанию – уже графическая. Ее CSS-файл используется для применения графических скинов, доступных в виде файлов с расширением *.fla* (Flash IDE). Если в вашей системе установлена среда разработки Flash IDE, вы можете посмотреть данный файл, расположенный в следующей папке:

Mac

/Mac/Applications/Adobe Flex Builder 3/sdks/3.0.0/frameworks/themes/AeonGraphical/src

Windows

C:\Program Files\Adobe Flex Builder 3\sdks\3.0.0\frameworks\themes\AeonGraphical\src

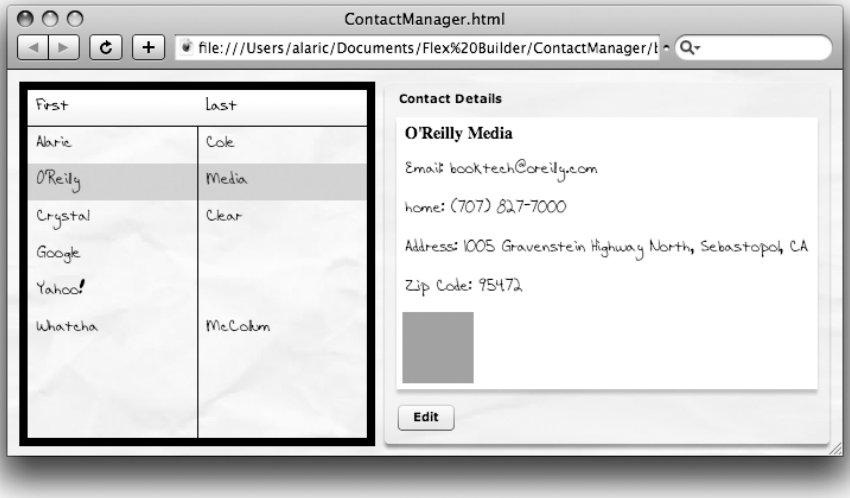

Рис. 14.19. *Приложение ContactManager с использованием темы Flekscribble (автор Ральф Сцепан (Ralf Sczepan))*

Рис. 14.20. *Приложение PhotoGallery с использованием темы, создающей перекрывающие вкладки (автор Хуан Санчес (Juan Sanchez))*

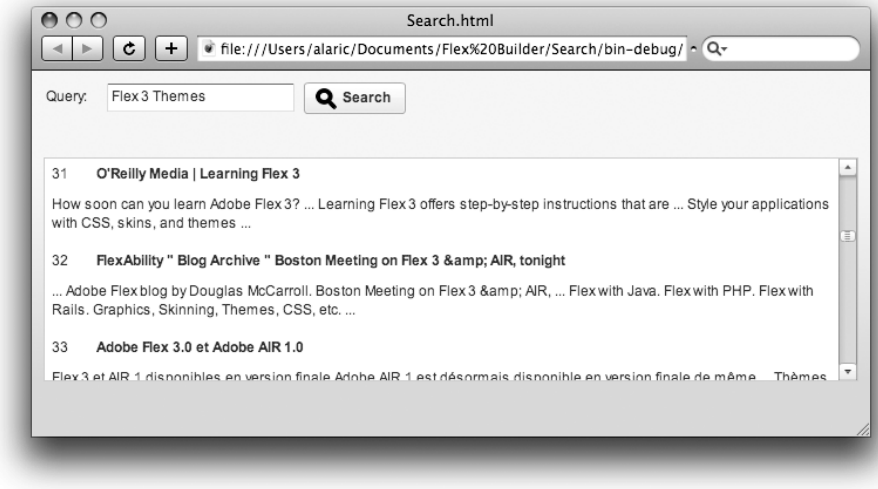

Рис. 14.21. *Приложение Search с использованием темы Yahoo!, доступной по адресу http://developer.yahoo.com/flash/articles/yahoo-flex-skin.html*

Заключение

В данной главе мы плавно перешли от обсуждения вопросов создания Flex-приложения к способам его оформления. Вы научились изменять внешний вид ваших компонентов с помощью атрибутов стилей, внешних CSS-файлов и даже графических изображений. Теперь вы можете самостоятельно изучить возможности CSS, но не стоит забывать и об отличных инструментах Flex Builder, предназначенных для редактирования внешнего вида компонентов в визуальном режиме.

Теперь вам наверняка захочется так или иначе изменить оформление некоторых ваших компонентов, а, возможно, и всего приложения в целом. Возможности изменения внешнего вида создаваемых Flex-приложений ограничены только вашим воображением.

Итак, мы рассмотрели все основные навыки, необходимые разработчику для создания Flex-приложений. В следующей, последней главе вы наконец узнаете, как рассказать всему миру о существовании ваших приложений.

15

В этой главе:
- Интернет-приложения
- Настольные приложения

Распространение приложения

В этой – заключительной – главе я подробно расскажу о том, как подготовить созданное вами приложение для распространения. В этой главе вы узнаете, что для этого нужно независимо от того, будет ли ваше приложение настольным или доступным только через Интернет (или вы захотите использовать обе эти возможности).

Каждый из перечисленных способов применения приложений имеет свои преимущества и недостатки. Веб-приложения расположены на определенной веб-странице, что делает их доступными для большинства пользователей, чей компьютер имеет подключение к сети Интернет. Однако время от времени подключение может отсутствовать или же может возникнуть необходимость использования приложения в автономном режиме.

Настольные приложения всегда доступны, поскольку они установлены прямо в системе пользователя. Они обладают такими традиционными функциями, как возможность запуска с помощью меню Start в Windows или Dock в Mac OS X, и т. п. Такие приложения выполняются в собственном окне, поэтому нет необходимости в использовании броузера. Они также поддерживают технологию перемещения объектов из других приложений с помощью мыши (drag and drop), а также могут иметь доступ к файлам локальной файловой системы. Однако, чтобы воспользоваться этими преимуществами, нужно установить приложение на локальный компьютер.

Какой бы способ вы не выбрали, в одном можно не сомневаться – единожды созданные, ваши Flex-приложения смогут работать на базе всех основных операционных систем.

Интернет-приложения

Приложения, разрабатываемые вами по ходу прочтения данной книги, тестировались с помощью веб-броузера. При создании приложений ContactManager, PhotoGallery и Search в диалоговом окне New Flex Project вы указывали, что планируете разработку приложений для веб-применения. Однако это вовсе не ограничивает возможности применения приложения только веб-средой, поскольку их также можно будет использовать и в качестве настольных. О том, как это сделать, говорится чуть ниже в разделе «Настольные приложения», а пока поговорим о механизме подготовки к использованию веб-приложения.

Первым шагом в этом направлении является сборка чистовой (release) версии. В процессе разработки скомпилированное приложение помещается в папку, по умолчанию названную bin-debug. Эта версия содержит вспомогательную информацию, используемую для отладки, которая используется такими средствами Flex Builder, как Debugger и Profiler (подробнее об этом во врезке «Оптимизация производительности»). Это очень удобно в процессе разработки приложения, но при подготовке приложения к использованию целесообразно создать чистовую версию, избавившись от информации для отладки, благодаря чему размер файла *.swf* несколько сократится.

Откройте проект PhotoGallery; попробуем создать чистовую версию приложения на его примере. Прежде всего следует установить ее необходимые настройки. В действительности это относится как к чистовой версии, так и к отладочной версии, поэтому разработчику необходимо ознакомиться с этим процессом.

Настройки приложения

Для доступа к настройкам вашего приложения выделите проект на панели Flex Navigator и выберите меню Project→Properties. В открывшемся диалоговом окне Properties выберите пункт Flex Compiler из списка слева (рис. 15.1), чтобы перейти к разделу настроек компилятора Flex, с помощью которого можно изменить опции сборки вашего приложения.

Версия Flex SDK

Первая группа настроек в рассматриваемом диалоговом окне касается используемой версии Flex SDK. По умолчанию используется значение Flex 3 SDK, поэтому (если только вы не намерены разрабатывать приложения с использованием предыдущих версий Flex SDK) не стоит ничего менять. Если вы когда-либо использовали более старые версии Flex, в вашем коде могут быть использованы компоненты из предыдущих версий SDK (например, Flex 2.0). Определенные компоненты API могут изменяться от одной версии к другой, и при необходимости

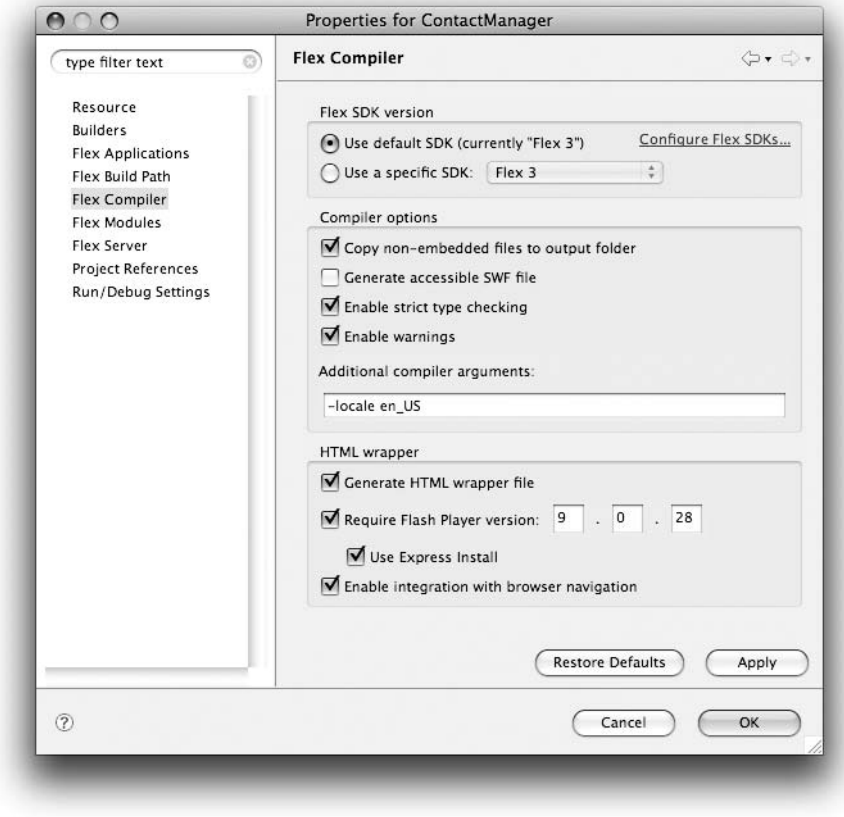

Рис. 15.1. *Вид раздела Flex Compiler диалогового окна Properties*

можно использовать опцию Use a specific SDK для определения используемой вами версии. В нашем примере достаточно оставить настройку по умолчанию.

Дополнительные ресурсы

Настройка Copy non-embedded files to output folder (Копировать невстраиваемые ресурсы в выходную папку) определяет, нужно ли включать в окончательную версию дополнительные ресурсы, например файлы *.xml* или графические изображения. Ее стоит включить, чтобы файлы, необходимые для работы приложения, такие как *photos.xml*, были скопированы автоматически. Если вы предпочитаете копировать нужные файлы вручную, можно выключить данную опцию, но учтите, что в этом случае при каждом изменении исходных файлов их придется самостоятельно копировать в папку с чистовой сборкой.

Доступность для людей с ограниченными возможностями

Следующая опция касается доступности приложения для людей с ограниченными возможностями. В этом контексте понятие *доступности* означает предоставление возможности использования приложения любым пользователям. Как правило, в данном случае особое внимание уделяется людям со слабым зрением, ухудшением слуха, некоторыми нарушениями опорно-двигательного аппарата или расстройствами восприятия. Опция Generate accessible SWF file указывает на использование *экранного диктора*, специальной программы, предназначенной для чтения отображаемого на экране текста для пользователей со слабым зрением. По умолчанию она отключена, поскольку в противном случае в ваше приложение будет добавлен необходимый дополнительный код, а это увеличивает размер скомпилированного файла. Рекомендуется включить данную настройку, за исключением тех случаев, когда вы четко представляете себе будущую аудиторию и точно знаете, что использование экранного диктора не потребуется. (При включении этой опции размер скомпилированного SWF-файла приложения PhotoGallery составит 328 килобайт, в то время как то же приложение без поддержки экранного диктора займет 318 килобайт.) Однако для создания доступных приложений одной этой опции недостаточно. Более подробно данный вопрос рассматривается во врезке «Приложения, доступные каждому».

Примечание

При проектировании общественных зданий обязательно предусматривается наличие пандусов и других конструкций, благодаря которым оно становится доступным для любых посетителей. Точно так же дело обстоит и с приложениями – каждый человек должен иметь возможность ими пользоваться.

Приложения, доступные каждому

Включение опции Generate accessible SWF file – важный шаг на пути создания доступного приложения, но это лишь первый шаг. Возможно, до сих пор вы об этом не задумывались, но ведь многие люди, заинтересованные в использовании разработанного вами приложения, могут иметь проблемы со здоровьем или иные ограничения.

Среди потенциальных пользователей вашего приложения может быть немало людей со слабым зрением и даже полностью слепых. Возможно, в случае с фотоальбомом это не актуально, но, в целом, принципы доступности распространяются и на эти приложения. Ведь человек может быть не полностью слепым, а лишь иметь слабое зрение или страдать дальтонизмом. В этом случае

важно исключить определяющую роль цветов при использовании функций приложения и убедиться, что используемые цвета не мешают восприятию текста. Кроме того, многие пользователи, имеющие проблемы со зрением, устанавливают более низкое разрешение экрана, и ваше приложение должно нормально отображаться в таком режиме. И даже полностью лишенные зрения пользователи должны иметь возможность взаимодействия с вашим приложением – по крайней мере на уровне понимания, что данная программа применяется для просмотра изображений. Конечно, они не смогут увидеть сами изображения, но им может быть доступна информация об их описании и т. п.

Будьте осторожны при использовании звуков в ваших приложениях. Иногда их использование может мешать нормальному восприятию текста, воспроизводимого с помощью экранного диктора, поэтому следует добавить в приложение возможность выключения звуков или музыкального сопровождения. Также позаботьтесь о создании возможности включения субтитров.

Кроме того, убедитесь, что любые компоненты вашего приложения можно использовать без мыши. Иными словами, необходимо создать возможность полного управления приложением только с помощью клавиатуры. Некоторые пользователи не имеют возможности пользоваться мышью, но это не должно стать препятствием при их работе с вашим приложением. Как правило, для разработчика это не проблема, поскольку все стандартные компоненты Flex имеют встроенную поддержку управления с помощью клавиатуры. Однако доступ к некоторым функциям вашего приложения может осуществляться с помощью контекстного меню или двойного щелчка мышью – одной клавиатурой тут не обойтись. В таком случае рекомендуется создать несколько способов выполнения подобного рода действий – как с помощью мыши, так и посредством одной клавиатуры.

Далее, не обойдите вниманием *порядок перехода по Tab*, т. е. порядок смены фокуса компонентов при нажатии на клавишу Tab. Как правило, Flex автоматически создает последовательность смены фокуса, основываясь на логике расположения компонентов. Тем не менее, на всякий случай стоит протестировать порядок перехода, в особенности если вы используете собственные настройки. Лучший метод для определения степени доступности приложения – тестирование возможности управления им с помощью одной лишь клавиатуры. На минуту забудьте о существовании мыши – это поможет понять, насколько удобно будет использовать ваше приложение.

И наконец, ваше приложение должно иметь логичную и понятную структуру и удобную навигацию. По возможности используйте иконки и другие изображения, дающие дополнительную информацию о назначении тех или иных компонентов, но не полагайтесь на них полностью. К примеру, изображение лупы на кнопке в приложении Search наводит на мысль, что нажатие на кнопку запустит процесс поиска. Однако одной иконки недостаточно для получения правильного представления о назначении кнопки (лупа может также означать, к примеру, увеличение), поэтому вместе с ней используется текстовая метка Search.

Вы наверняка заметите, что применение данных методов делает ваши приложения более логичными, понятными и простыми в использовании – и это касается не только людей с ограниченными возможностями, но и среднестатистических пользователей. Поэтому следование этим несложным правилам поможет вам создавать не просто доступные, но и по-настоящему профессиональные, хорошо спроектированные и удобные приложения. Логичность и понятность принципов использования в любых условиях воспринимаются как достоинство приложения.

Предупреждения компилятора

Следующие несколько опций касаются возможности строгой проверки типов данных и вывода предупреждений. Рекомендую вам включить обе настройки – это обеспечит отображение важных оповещений на панели Problems и поможет добиться идеального функционирования ваших приложений.

HTML-обертка

HTML-обертка – это HTML-файл, генерируемый для отображения в нем Flex-приложения. Его код встраивает файл *PhotoGallery.swf* в веб-страницу, которая будет размещена в выходной папке под названием *PhotoGallery.html*. Оставьте настройку Generate HTML wrapper включенной, чтобы страница была создана автоматически, если, конечно, вы не написали для нее свой собственный код. (Подробную информацию о генерируемом HTML-файле и его настройке читайте во врезке «Конфигурация HTML-обертки».)

Конфигурация HTML-обертки

Во многих случаях сгенерированный автоматически HTML-файл вполне подойдет для вашего приложения, но иногда может потребоваться внести некоторые изменения в его шаблон или даже создать свой собственный файл. Для этого необходимо понимать, каким образом он устроен.

HTML-шаблон хранится в папке html-template, которая расположена в корне вашего проекта. В ней также находятся дополнительные файлы, которые могут быть скопированы в выходную папку, но основной файл называется *index.template.html*. Это еще не конечный HTML-файл, а шаблон для обработки компилятором Flex. После добавления некоторой дополнительной информации на его основе будет сформирован конечный HTML-файл, в который будет встроено приложение Flex.

Данный шаблон содержит переменные, так называемые *маркеры* (например, `${title}`), которые компилятор Flex заменяет действительными текстовыми значениями. В табл. 15.1 представлен список возможных маркеров. Как правило, их настройки по умолчанию вполне удовлетворят разработчика, но некоторые могут потребовать изменения (например, `${title}`, отвечающий за текст, отображаемый в строке заголовка HTML-страницы).

Можно изменять и другие части данного файла, например информацию, отображаемую в том случае, когда в системе пользователя не установлен проигрыватель Flash Player. Она определяется с помощью переменной `alternateContent`, которую вы легко найдете, просмотрев файл.

Таблица 15.1. Перечень маркеров шаблона HTML-обертки

Имя маркера	Описание
`${application}`	Определяет идентификатор (`id`) встраиваемого SWF-файла, который может быть использован для ссылки на него в коде на JavaScript или иных языках для написания сценариев, обрабатываемых броузером.
`${bgcolor}`	Цвет фона HTML-файла. Как правило, Flex-приложение занимает все доступное пространство в окне броузера, но фон может быть виден при изменении размеров окна броузера или в том случае, когда приложение не занимает все доступное пространство. Вы также можете установить цвет фона с помощью тега метаданных SWF, о чем было сказано в главе 14 «Визуальное оформление приложения».

Имя маркера	Описание
`${height}`	Высота приложения, определяемая свойством `height` тега `<mx:Application/>`.
`${swf}`	Определяет полный путь к скомпилированному файлу приложения с расширением *.swf*.
`${title}`	Название HTML-страницы, отображаемое в строке заголовка броузера, по умолчанию – имя приложения, например `PhotoGallery`.
`${version_major}`	Требуемый старший номер версии Flash Player, например 9. Этот маркер используется только в обертках с определением версии Flash Player, как указано в настройках компиляции HTML-обертки.
`${version_minor}`	Требуемый младший номер версии Flash Player, как указано в настройках компиляции HTML-обертки.
`${version_revision}`	Требуемый номер редакции версии Flash Player, как указано в настройках компиляции HTML-обертки.
`${width}`	Ширина приложения, определяемая свойством `width` тега `<mx:Application/>`.

Версия Flash Player

Следующая настройка в списке опций Flex Compiler определяет, нужно ли добавить в HTML-обертку код, отвечающий за проверку версии плагина Flash Player. Поскольку использование различных версий компонентов Flex, а также написанный вами код на ActionScript может потребовать определенной версии Flash Player для воспроизведения, можно оставить настройки по умолчанию. Опция Use Express Install позволит приложению осуществлять запрос на обновление проигрывателя, если необходимая версия в системе пользователя отсутствует. Для этого в HTML-обертку добавляется дополнительный код, а в выходную папку также будет помещен специальный SWF-файл с именем *playerProductInstall.swf*.

Навигация с использованием журнала посещений

Последняя настройка в рассматриваемом диалоговом окне – Enable integration with browser navigation. Ее включение добавляет в выходную папку необходимые файлы (из папки history вашего проекта) для управления журналом посещений и позволит использовать кнопки Back и Forward броузера для навигации по приложению (об этом подробнее

говорилось в главе 11). Приложение PhotoGallery не использует журнал посещений, поэтому в нашем случае эту опцию можно оставить выключенной.

Экспорт чистовой версии приложения

Настройка приложения завершена; можно приступать к созданию чистовой версии. Для этого необходимо выделить проект на панели Flex Navigator и выбрать меню Project→Export Release Build. При этом откроется диалоговое окно, изображенное на рис. 15.2.

Поскольку проект уже был выбран во Flex Navigator, первая настройка в данном окне определена. Опция Application позволяет выбрать приложение для экспорта, если в вашем проекте их несколько. Проект PhotoGallery, как и остальные проекты, созданные вами по ходу прочтения этой книги, содержит только одно приложение, поэтому в этой настройке нет необходимости.

Примечание

В один проект может входить несколько приложений. Довольно удобно хранить их в одном проекте, если они используют одни и те же дополнительные ресурсы, компоненты и/или настройки.

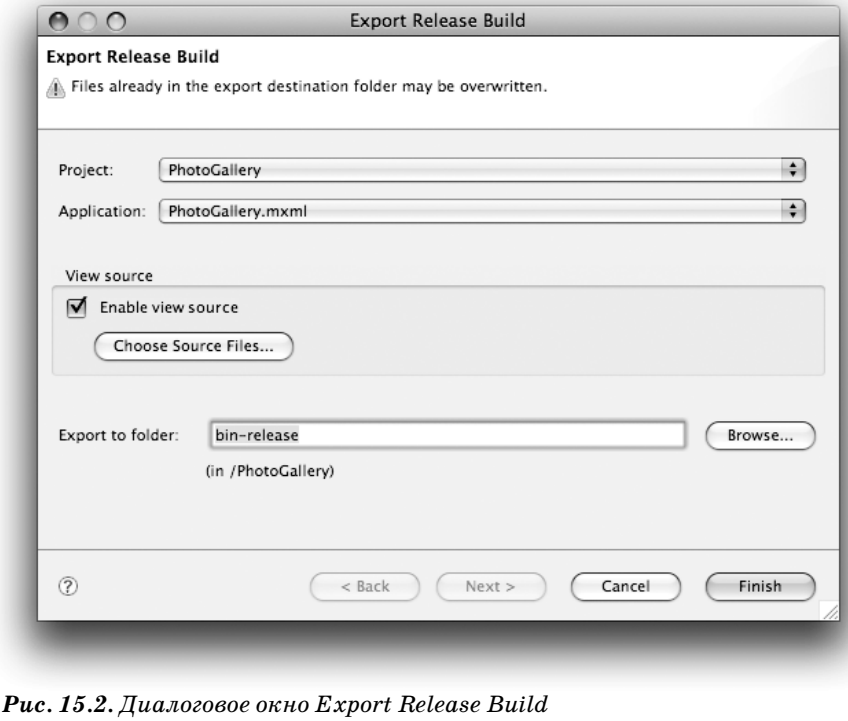

Рис. 15.2. *Диалоговое окно Export Release Build*

Открытие исходного кода

Следующий раздел диалогового окна Export Release Build называется View Source. При установке флажка напротив Enable view source в окончательную версию приложения будет включена папка с веб-страницей, содержащей его исходный код. На ней будет располагаться список входящих в приложение файлов, область для просмотра исходного кода (как в примере на рис. 15.5), а также ссылка для скачивания zip-архива с исходным кодом. Если вы хотите дать другим пользователям возможность видеть исходный код вашего приложения, включите эту опцию. С помощью этого открытого кода они смогут изучать Flex. Кроме того, опытные программисты, получившие доступ к коду приложения, могут заметить возможные ошибки и указать вам на их причину.

Если вы решите открыть исходный код вашего приложения, можно выбрать, доступ к каким файлам будет возможен, а к каким – нет. Для этого щелкните по кнопке Choose Source Files. Откроется диалоговое окно, изображенное на рис. 15.3, в котором вы увидите древовидный список файлов исходного кода вашего приложения. В данном примере я предпочел открыть только код, расположенный в папке src, в отличие от файлов с кодом HTML-обертки (в папке html-template) и пустой папки libs. Название выходной папки, в которую будет помещен исходный код, также можно изменить с помощью соответствующего поля.

Если вы решите открыть доступ к исходным файлам, в теге `<mx:Application/>` приложения *PhotoGallery.mxml* появится новый атрибут `viewSourceURL`, указывающий путь к папке с исходным кодом. Данный

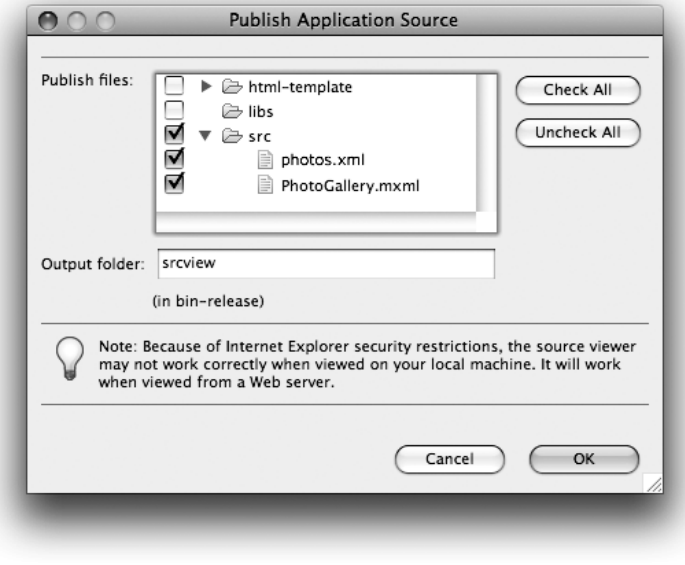

Рис. 15.3. *Диалоговое окно Publish Application Source*

атрибут создает новый пункт в контекстном меню (View Source), появляющемся при щелчке правой кнопкой мыши по приложению, воспроизводимому в броузере (рис. 15.4). При выборе данного пункта откроется область просмотра, как показано на рис. 15.5.

Внимание

Если политика вашей компании не позволяет открывать исходный код разрабатываемых приложений, удостоверьтесь, что данная возможность отключена. Кроме того, если ранее опция Enable view source была включена, проверьте, что

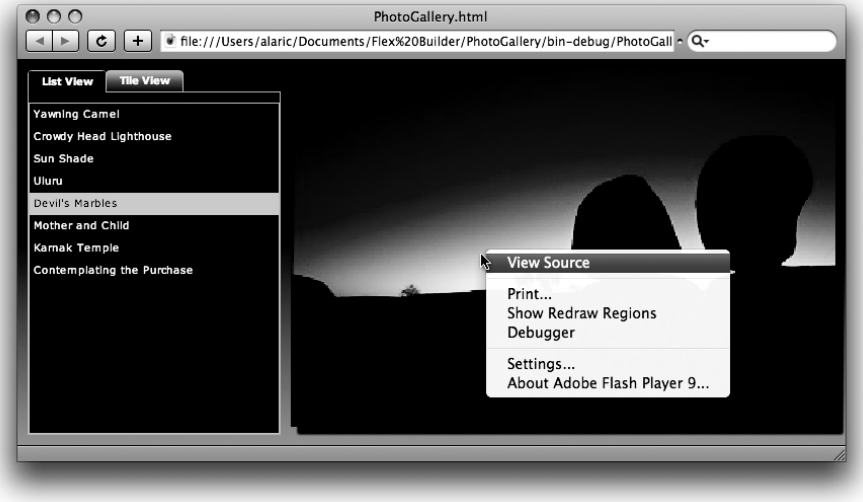

Рис. 15.4. *Приложение PhotoGallery: контекстное меню*

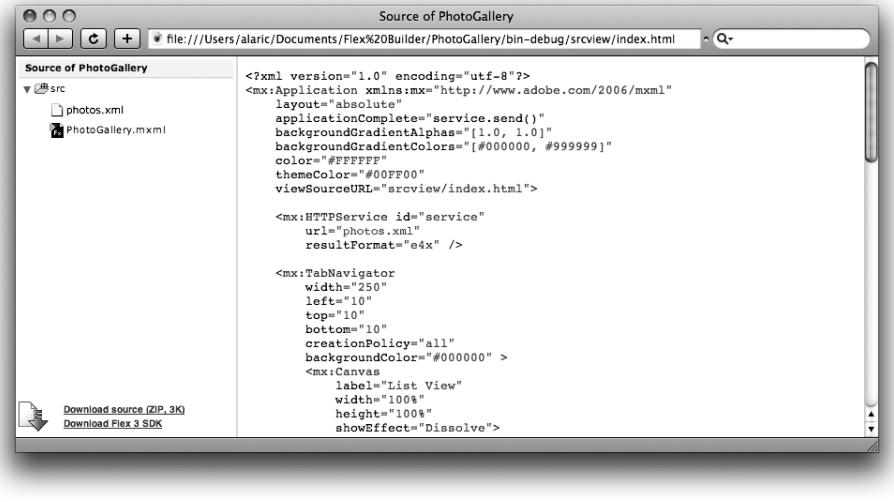

Рис. 15.5. *Приложение PhotoGallery: область просмотра исходного кода*

в выходной папке не осталось исходных файлов. Даже при отключении данной настройки загруженные на сервер папки с исходным кодом будут доступны сторонним пользователям.

Папка для экспорта

Последняя опция, доступная в диалоговом окне Export Release Build, называется Export to folder. С ее помощью можно изменить расположение и имя папки, в которой будут находится файлы чистовой версии. Лучше всего оставить настройки по умолчанию, если только вы не хотите присвоить папке новое имя или поместить файлы в определенную папку в вашей системе или на сетевом диске. Пока что оставьте папку, указанную по умолчанию (bin-release), и нажмите кнопку Finish. Окончательная версия приложения будет находиться в созданной внутри проекта `PhotoGallery` новой папке bin-release, как показано на рис. 15.6. Данная папка содержит только файлы, необходимые для запуска приложения в броузере.

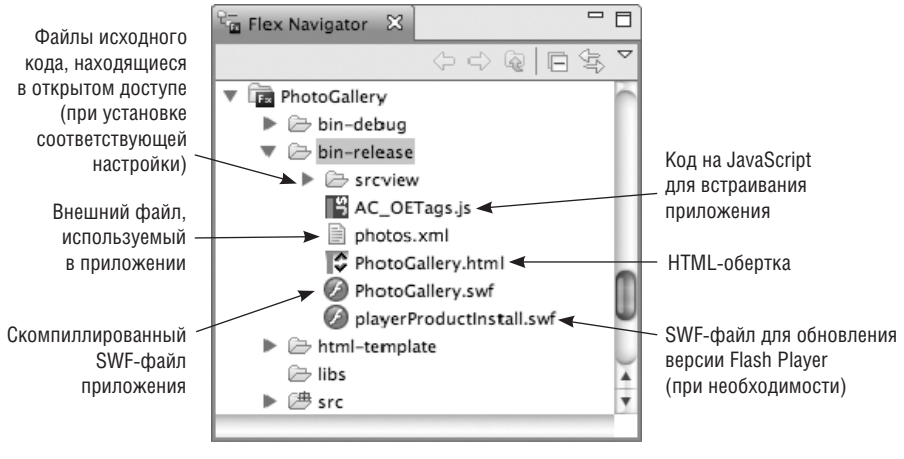

Рис. 15.6. *Структура папки с чистовой версией приложения PhotoGallery*

Как сократить время загрузки приложения

Flex-приложения обладают на удивление малым размером, учитывая их широкие возможности. Конечный размер приложения зависит от нескольких факторов, таких как объем кода и количество используемых компонентов. Поскольку в скомпилированный SWF-файл входит весь код, необходимый для запуска, его размер прямо пропорционален количеству используемых компонентов. Поскольку веб-приложения отображаются через броузер и должны быть загружены через сеть,

имеет смысл по возможности сократить их размер, чтобы не заставлять пользователей долго ждать их загрузки.

Размер типичного Flex-приложения, в которое входят несколько стандартных компонентов вроде Button, List и пара контейнеров, составляет приблизительно 300 килобайт. Однако размер приложения не обязательно должен зависеть от количества используемых компонентов. Благодаря возможности кэширования стандартных компонентов среды Flex становится возможным хранить их на локальном компьютере пользователя, что избавляет от необходимости повторной загрузки компонентов каждый раз при обращении пользователя к Flex-приложению, использующему эти компоненты.

Примечание

Flex также позволяет разбивать приложение на модули, т. е. части, загружаемые по отдельности. Такую функцию уместно использовать при разработке объемных приложений, состоящих из большого количества частей или видов. Если эта возможность кажется вам удачным решением стоящей перед вами задачи, поищите в документации Flex информацию о компоненте Module.

Код оболочки Flex хранится в отдельном файле. При первой загрузке приложения он загружается на локальный компьютер пользователя и сохраняется в кэше. В следующий раз при загрузке того же самого приложения или другого Flex-приложения, поддерживающего функцию *кэширования оболочки*, будут использоваться компоненты оболочки, уже сохраненные на компьютере пользователя, без необходимости загружать их вновь.

Примечание

Механизм работы большинства броузеров подразумевает кэширование загружаемых Flex-приложений, расположенных на веб-страницах, с помощью самого броузера. Это означает, что загружаемое не в первый раз приложение будет скачано мгновенно, вне зависимости от того, используется ли в нем функция кэширования оболочки. Кэширование оболочки удобно использовать при работе с несколькими приложениями или приложениями, удаленными из кэша броузера.

На практике применение в приложении функции кэширования оболочки означает первоначальную загрузку файла большего размера. При этом следующие загрузки будут проходить быстрее. Таблицы 15.2 и 15.3 наглядно демонстрируют различия в объеме файла приложения `PhotoGallery` при использовании функции кэширования оболочки и без нее. В первом случае первоначальная загрузка составит 639 Кбайт, что значительно превышает размер загрузки при отсутствии кэширования. Однако во все последующие разы будет загружаться лишь 101 Кбайт данных. Если вы не используете кэширование оболочки, размер загружаемых данных всегда будет равен 318 Кбайт.

Примечание

Кэширование оболочки сохраняет только стандартные компоненты Flex – при использовании компонентов, предлагаемых сторонними разработчиками, данная функция работать не будет.

Таблица 15.2. Размер приложения PhotoGallery при первоначальной загрузке

	С использованием кэширования оболочки	Без использования кэширования оболочки
PhotoGallery.swf	101 Кб	318 Кб
SWF-файл внешней оболочки (загружается один раз)	538 Кб	–
Итого	**639 Кб**	**318 Кб**

Таблица 15.3. Размер приложения PhotoGallery при каждой последующей загрузке

	С использованием кэширования оболочки	Без использования кэширования оболочки
PhotoGallery.swf	101 Кб	318 Кб
SWF-файл внешней оболочки (загружается один раз)	– (сохранен в кэше)	–
Итого	**101 Кб**	**318 Кб**

Если вы сочтете, что в вашем приложении следует использовать функцию кэширования оболочки, ее можно включить, выделив проект на панели Flex Navigator и выбрав Project→Properties. В открывшемся диалоговом окне Properties перейдите в раздел Flex Build Path (рис. 15.7).

При этом откроется диалоговое окно Flex Build Path, состоящее из двух разделов, переход между которыми осуществляется с помощью переключателя: Source path, где можно изменить пути к папкам с исходным кодом и с чистовой версией, и Library path, где можно указать внешние библиотеки, которые требуется подключить к проекту.

По умолчанию Flex-приложения используют Flex SDK 3, который стоит первым пунктом в списке Build path libraries в разделе Library path. Необходимо изменить настройку Framework linkage, поскольку по умолчанию устанавливается значение Merged into code, что означает компилирование Flex SDK в конечный SWF-файл приложения. Установите значение Runtime shared library (RSL) для экспорта компонентов Flex в отдельные SWF-файлы.

Это означает, что размер конечного файла приложения с расширением *.swf* сократится, но для его функционирования будет загружено два отдельных SWF-файла большего размера, которые затем будут сохранены

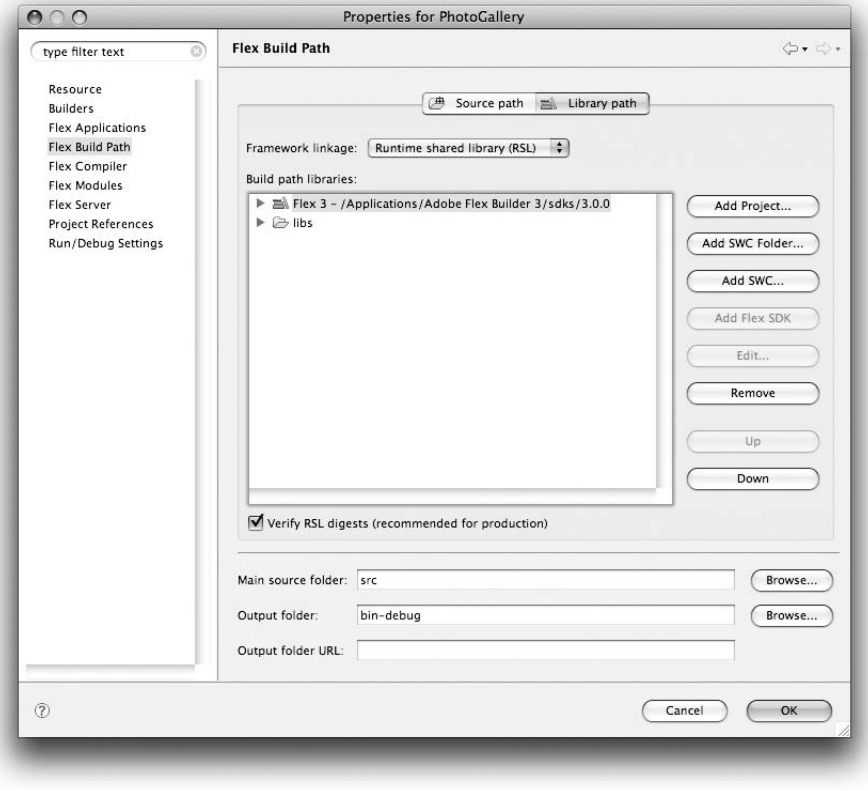

Рис. 15.7. *Раздел Flex Build Path диалогового окна Properties для проекта PhotoGallery*

в кэше. В нашем примере в окончательную версию приложения `PhotoGallery` войдут два дополнительных (и по сути одинаковых) файла. Первый из них – неподписанная версия, используемая предыдущими версиями Flash Player, второй – подписанная версия для текущей версии Flash Player. На рис. 15.8 изображена структура окончательной версии приложения `PhotoGallery` с использованием функции кэширования оболочки.

Выбор хостинга

Чтобы заявить о существовании вашего приложения всему миру, его окончательную версию нужно разместить на сервере. Если у вас нет своего места на сервере для хостинга (к примеру, предоставляемого вашей компанией или учебным заведением), можно зарегистрироваться в одном из бесплатных сервисов, предоставляющих такие услуги, – скажем, Yahoo! Geocities (*http://geocities.yahoo.com*). Однако если вы настроены достаточно серьезно, стоит задуматься о приобретении хос-

Рис. 15.8. *Структура папки с чистовой версией приложения PhotoGallery с использованием функции кэширования оболочки*

тинга, предоставляющего вам более полный контроль. Выбор хостинга – дело нелегкое, поскольку необходимо взвесить все аспекты предлагаемых услуг, чтобы деньги не были потрачены впустую. К счастью, услуги хостинга дешевеют с каждым днем.

Примечание

Если ваше Flex-приложение предполагает использование серверных технологий, таких как ColdFusion или PHP, необходимо удостовериться, что выбранный вами хостинг предоставляет такую возможность.

Когда хостинг или веб-сервер уже выбран, можно скопировать все файлы, входящие в состав окончательной версии приложения, в соответствующий каталог на сервере. URL-адрес для доступа к вашему приложению зависит от доменного имени и структуры каталогов вашего хоста. К примеру, окончательная версия приложения `PhotoGallery` размещена на моем сайте и доступна по адресу *www.greenlike.com/flex/learning/projects/photogallery/PhotoGallery.html*, потому что она расположена в каталоге flex/learning/projects/photogallery моего сервера, имя хоста которого *greenlike.com*. Результат можно увидеть на рис. 15.9 или перейдя по указанной ссылке.

Примечание

Имя экспортируемого HTML-файла можно изменить на более простое. Если переименовать файл в *index.html*, большинство веб-серверов будут осуществлять загрузку данного файла при обращении к содержащему его каталогу. К примеру, если бы я изменил название файла *PhotoGallery.html* на *index.html*, приложение было бы доступно по адресу *www.greenlike.com/flex/learning/projects/photogallery/*, а не *www.greenlike.com/flex/learning/projects/photogallery/PhotoGallery.html*.

Рис. 15.9. *Веб-приложение PhotoGallery, отображаемое в броузере*

Оптимизация производительности

Важным этапом разработки приложения является проверка его производительности. Об этом следует задумываться уже в процессе написания кода, однако считается, что лучше завершить основной процесс разработки перед тем, как начать оптимизацию вашего приложения.

> *Не стоит уделять внимания мелким улучшениям в течение примерно 97% времени, потраченного на разработку; преждевременная оптимизация – корень всех зол.*
>
> Дональд Кнут (Donald Knuth)

Если вы заинтересованы в увеличении производительности вашего приложения, во Flex Builder у вас под рукой всегда есть инструмент Profiler. Он доступен через меню Run→Profile или с помощью кнопки Profiler на панели инструментов (рис. 15.10). Он поможет распознать «слабые места» приложения, выполняя замеры времени выполнения различных частей вашего приложения (например, вызова метода). Это позволит вам определить, какие части оказывают наиболее сильное влияние на время выполнения приложения. Эта информация будет весьма полезна при дальнейшей оптимизации приложения.

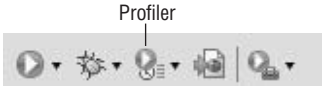

Рис. 15.10. *Кнопка Profiler на панели инструментов*

Настольные приложения

Возможности Flex не ограничиваются созданием веб-приложений – это также отличный инструмент создания «традиционных» настольных приложений. С помощью Adobe AIR можно создавать приложения, которые можно загрузить и установить на локальном пользовательском компьютере и использовать в любое время вне зависимости от наличия подключения к сети Интернет. Некоторые приложения (хотя и далеко не все) имеет смысл использовать именно в качестве настольных. Другие прекрасно подходят как для работы в среде броузера, так и для использования в качестве настольных. Выбор остается за вами, поскольку можно с легкостью переносить приложения с одной платформы на другую или даже создать код, работающий на любой платформе.

При создании нового Flex-проекта перед вами встает выбор между разработкой веб-приложения или приложения для настольного использо-

Тестирование пользователями (или друзьями)

Лучший способ проверить, удобно ли работать с вашим приложением, т. е. дружественен ли ее интерфейс для пользователя, – спросить мнение других людей. Наблюдение за работой других пользователей с вашим приложением поможет сделать важные выводы о том, что необходимо исправить для совершенствования всех аспектов взаимодействия с пользователем. Поскольку вы постоянно работаете с приложением в процессе его разработки, вы можете не замечать некоторых его недостатков. Но человек, впервые видящий ваше приложение, с большой долей вероятности будет использовать его как-то по-другому.

Если вы не имеете возможности воспользоваться услугами специалистов по взаимодействию с пользователями, на помощь могут прийти ваши друзья. Покажите ваше приложение паре знакомых и посмотрите, как они будут с ним работать. Не объясняйте им, как пользоваться приложением и не отвечайте на вопросы – ваша задача – всего лишь поудобнее откинуться на спинку кресла и наблюдать, как они будут сиять от удовольствия – или же кипеть от раздражения. Помните, что вы не сможете помочь своими разъяснениями каждому потенциальному пользователю вашего приложения, поэтому не стоит давать вашим приятелям никаких привилегий в виде дополнительной информации или подсказок.

Спокойно выслушайте все комментарии, даже если вы не совсем с ними согласны, – любой отзыв может содержать ценную информацию, которая пригодится вам для совершенствования ваших приложений.

вания. В самом начале разработки у вас может еще не быть четкого представления о том, как будет представлен результат, или же вы хотите развернуть обе возможности. Не стоит сильно ломать голову, ведь создать новый проект – проще простого.

Примечание

Чем разнообразнее круг людей, тестирующих ваше приложение, тем лучше – в противном случае есть вероятность упустить много важных аспектов использования приложения.

Вы уже давно не создавали новых Flex-проектов; на этот раз мы создадим проект, ориентированный на настольное применение. В следующем разделе подробно описано, как преобразовать веб-приложение `ContactManager` в настольное, создав новый проект с соответствующими настройками и скопировав в него часть имеющегося кода.

Создание проекта Adobe AIR

Создайте новый Flex-проект с именем `AddressBook`, используя команду New→Flex Project. В разделе Application type диалогового окна New Flex Project укажите Desktop application (runs in Adobe AIR), как показано на рис. 15.11.

После нажатия на кнопку Finish будет создан новый проект Adobe AIR. Заметьте, что для его отображения на панели Flex Navigator используется другая иконка; кроме того, он содержит XML-файл с именем *AddressBook-app.xml*. В нем содержатся настройки для настольного приложения, касающиеся названия в строке заголовка, и т. д. – но об этом чуть позже.

Открыв основной файл приложения, вы также заметите некоторые различия: вместо привычного корневого тега `<mx:Application/>` в нем используется тег `<mx:WindowedApplication/>`. Компонент WindowedApplication подобен Application, но обладает дополнительными функциями, необходимыми для работы Flex-приложения в качестве настольного. Код файла *AddressBook.mxml* будет выглядеть следующим образом:

```
<mx:WindowedApplication
  xmlns:mx="http://www.adobe.com/2006/mxml"
  layout="absolute">

</mx:WindowedApplication>
```

Итак, проект Adobe AIR создан; остается лишь скопировать код из приложения `ContactManager` в приложение `AddressBook`. Для этого скопируйте файлы, расположенные в папке с исходным кодом проекта `ContactManager`, и поместите их в папку с исходным кодом проекта `AddressBook`, за исключением основного файла приложения (*ContactManager.mxml*). Теперь файл таблицы стилей, внешний XML-файл и компонент ContactViewer будут доступны и приложению AIR.

Рис. 15.11. *Новый Flex-проект, запускаемый с помощью Adobe AIR*

Следующий шаг – копирование MXML-кода из файла *ContactManager.mxml* в файл *AddressBook.mxml*. Это можно сделать путем копирования кода, расположенного между открывающим и закрывающим тегами `<mx:Application/>` файла *ContactManager.mxml*, и его вставки между открывающим и закрывающим тегами `<mx:WindowedApplication/>` файла *AddressBook.mxml*. Атрибуты тега `<mx:Application/>` можно скопировать отдельно и поместить их внутри тега `<mx:WindowedApplication/>`.

Вместо этого можно также скопировать и вставить в файл *AddressBook.mxml* содержимое файла *ContactManager.mxml* полностью, а затем заменить тег `<mx:Application/>` на тег `<mx:WindowedApplication/>`.

Примечание

Для правильного функционирования приложения AIR необходимо использовать корневой тег `<mx:WindowedApplication/>`.

Код файла *AddressBook.mxml* теперь будет выглядеть примерно следующим образом:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication
  xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:view="com.oreilly.view.*"
  layout="absolute"
  applicationComplete="contactsService.send()"
  viewSourceURL="srcview/index.html">

<mx:Style source="styles.css"/>

  <mx:HTTPService id="contactsService"
    resultFormat="e4x"
    url="contacts.xml" />

  <mx:Parallel id="fadeAndResize">
    <mx:Dissolve id="dissolve"/>
    <mx:Resize id="fastResize" duration="300"/>
  </mx:Parallel>

  <mx:DataGrid id="contactsDataGrid"
    dataProvider="{contactsService.lastResult.contact}"
    selectedIndex="0"
    left="10"
    top="10"
    bottom="10"
    width="300"
    change="contactViewer.currentState = ''">
    <mx:columns>
      <mx:DataGridColumn headerText="First"
        dataField="firstName"/>
      <mx:DataGridColumn headerText="Last"
          dataField="lastName"/>
    </mx:columns>
  </mx:DataGrid>

  <view:ContactViewer id="contactViewer"
    contact="{contactsDataGrid.selectedItem}"
    x="318"
    y="10"
    resizeEffect="{fadeAndResize}"
    horizontalScrollPolicy="off"
    verticalScrollPolicy="off">
  </view:ContactViewer>

</mx:WindowedApplication>
```

Таким образом, вы создали приложение Adobe AIR. Его можно запустить так же, как и любое другое Flex-приложение, с помощью Flex Builder. Для этого выберите меню Run→Run AddressBook или щелкните по кнопке Run, расположенной на панели инструментов. При этом будет запущено приложение AddressBook, и вы увидите «родное» окно вашей операционной системы, как показано на рис. 15.12 и 15.13. Кроме того, если вы работаете на платформе Mac OS X, в меню Dock появит-

ся соответствующая иконка, а в системе Windows – соответствующий пункт меню на панели задач.

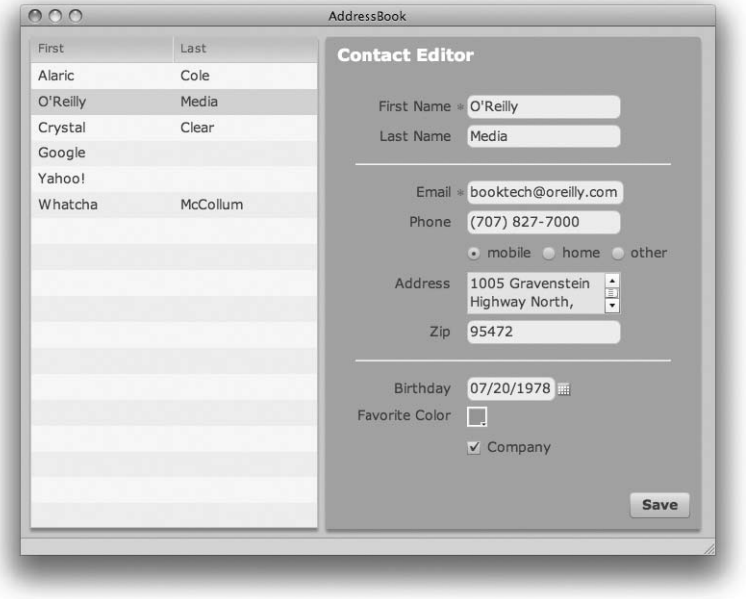

Рис. 15.12. *Настольное приложение AddressBook, запускаемое под Mac OS X*

Рис. 15.13. *Настольное приложение AddressBook, запускаемое под Windows Vista*

Настройка приложения

В меню Dock или на панели задач отображается иконка приложения, используемая AIR по умолчанию. Чтобы заменить ее на собственную, откройте файл *AddressBook-app.xml*. Он содержит шаблон файла дескриптора, который, подобно файлам в папке html-template в составе Flex-приложений для использования в среде веб, позволяет изменять настройки внешнего вида настольного приложения – например, текст, отображаемый в строке заголовка, оформление окна приложения и вид соответствующей иконки на панели задач или в меню Dock.

Описание доступных настроек можно увидеть в виде комментариев данного XML-файла. Сами настройки тоже заключены в комментарии; их можно раскомментировать, чтобы внести необходимые изменения. К примеру, можно заменить иконку, используемую по умолчанию, на свою собственную. Для этого необходимо создать изображение размером 128 на 128 пикселов (или загрузить мое изображение, расположенное по адресу *www.greenlike.com/flex/learning/projects/addressbook/address_icon.png*). Затем измените следующий код в файле дескриптора:

```
<!-- The icon the system uses for the application. For at least one
     resolution, specify the path to a PNG file included
     in the AIR package. Optional.
     Иконка приложения, используемая системой. Для изменения по крайней
     мере одной из возможных иконок укажите путь к файлу в формате PNG,
     входящему в пакет AIR (необязательно)-->
<!-- <icon>
        <image16x16></image16x16>
        <image32x32></image32x32>
        <image48x48></image48x48>
        <image128x128></image128x128>
</icon> -->
```

на

```
<!-- The icon the system uses for the application. For at least one
     resolution, specify the path to a PNG file included
     in the AIR package. Optional.
     Иконка приложения, используемая системой. Для изменения по крайней
     мере одной из возможных иконок укажите путь к файлу в формате PNG,
     входящему в пакет AIR (необязательно)-->
<icon>
    <image128x128>address_icon.png</image128x128>
</icon>
```

Это позволит использовать свойство, определяющее иконку (поскольку теперь оно не интерпретируется как комментарий), и создает для вашего приложения иконку размером 128 на 128 пикселов. Уменьшенные версии иконки будут созданы автоматически путем масштабирования большого изображения, но еще лучше создать отдельные изображения размерами 48 на 48, 32 на 32 и 16 на 16 пикселов.

Примечание

Ширину и высоту окна вашего приложения AIR можно задать в файле дескриптора, а, кроме того, определить их значения по умолчанию с помощью свойств `width` и `height` тега WindowedApplication.

Экспорт программы установки

Для экспорта программы установки вашего настольного приложения используется та же команда, что и для экспорта веб-приложения – Project→Export Release Build. При этом также откроется диалоговое окно Export Release Build (рис. 15.14).

Однако в отличие от экспорта веб-приложения, в этом случае в данном диалоговом окне необходимо указать дополнительную информацию. Во-первых, поскольку приложение AIR устанавливается в качестве настольного, оно требует цифровой подписи. Нажмите на кнопку Next, чтобы перейти к разделу Digital Signature диалогового окна Export Release Build (рис. 15.15). Цифровая подпись – средство обеспечения безопасности, используемое для гарантии того, что вы являетесь создателем данного приложения и что оно не было изменено кем-то другим с момента подписания. Для обеспечения должного функционирования приложения выберите первую опцию – Export and sign an AIR file with digital certificate. Это позволит защитить приложение с помощью цифрового сертификата, который можно получить на одном из специализиро-

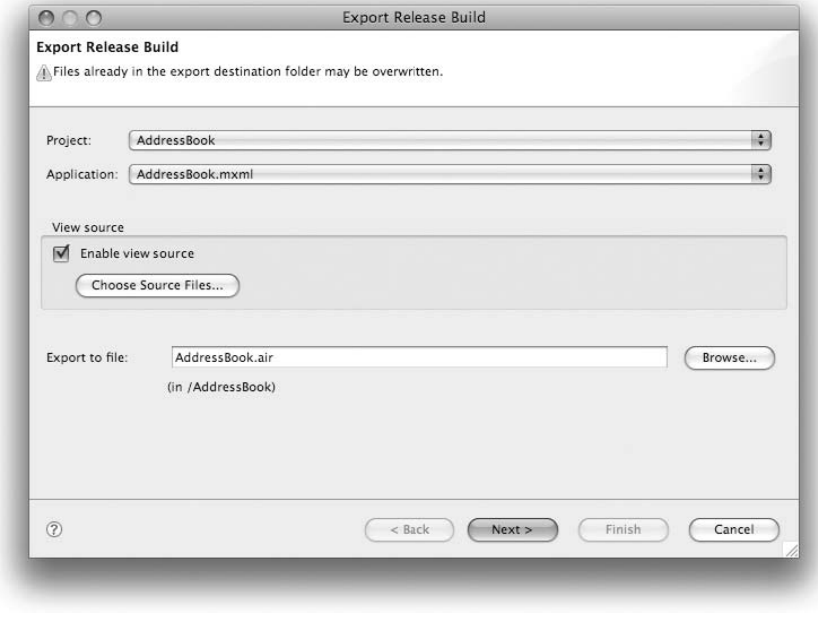

Рис. 15.14. *Экспорт приложения AIR: шаг первый*

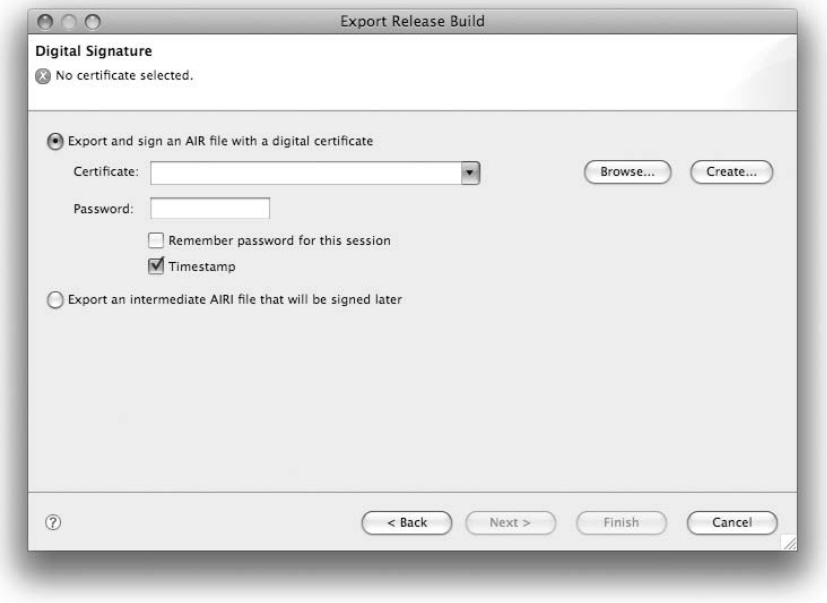

Рис. 15.15. *Экспорт приложения AIR: шаг второй*

ванных сайтов или создать самостоятельно. Если у вас нет сертификата безопасности, можно создать его, нажав на кнопку Create.

При этом откроется диалоговое окно, изображенное на рис. 15.16. С его помощью можно создать сертификат для вашего приложения – для этого достаточно указать имя владельца сертификата, пароль, а также имя самого сертификата. Я назвал сертификат *addressbook.certificate*, хотя на самом деле это не имеет большого значения. Запомните свой пароль, его нужно будет вводить каждый раз при использовании создаваемого сертификата.

После создания сертификата вернитесь к диалоговому окну Export Release Build, в котором снова нужно будет ввести пароль, выбранный вами для сертификата (рис. 15.17). Затем вы перейдете к последнему этапу процесса создания программы установки вашего приложения.

Он предполагает выбор файлов, которые необходимо включить в состав приложения (рис. 15.18). Необходимо удостовериться, что выбраны все файлы, необходимые для работы приложения, и исключить все лишнее. Скомпилированный файл приложения в формате *.swf* и файл дескриптора должны быть включены в обязательном порядке, поскольку без них функционирование приложения невозможно. Для обеспечения надлежащей работы приложения `AddressBook` потребуются также файлы *contacts.xml* и *addressbook_icon.png* (или иной файл с изображением иконки). Если код приложения будет размещен в открытом доступе, будет добавлена также папка, содержащая файлы с исходным кодом.

Рис. 15.16. *Экспорт приложения AIR: шаг третий*

Цифровые сертификаты

Цифровой сертификат – средство, гарантирующее, что настольное приложение не было изменено с момента его **подписания**. Он также используется для обеспечения безопасности и проверки подлинности личности создателя.

Если приложение AIR заверено собственным[1] цифровым сертификатом, информация о личности создателя не сможет пройти проверку на подлинность. Только отсутствие изменений в приложении с момента подписания может быть гарантировано, но не идентификация владельца подписи. Поэтому вместо имени создателя в диалоговом окне установки программы будет отображаться слово UNKNOWN (рис. 15.21).

Подробные сведения о том, как получить сертификат из надежных источников, можно найти на следующих веб-сайтах:

Thawte: *www.thawte.com*

Verisign: *www.verisign.com*

Microsoft Authenticode: *http://msdn2.microsoft.com/en-us/library/ms537364.aspx*

[1] Англ. self-signed, буквально «самоподписанный». – *Прим. перев.*

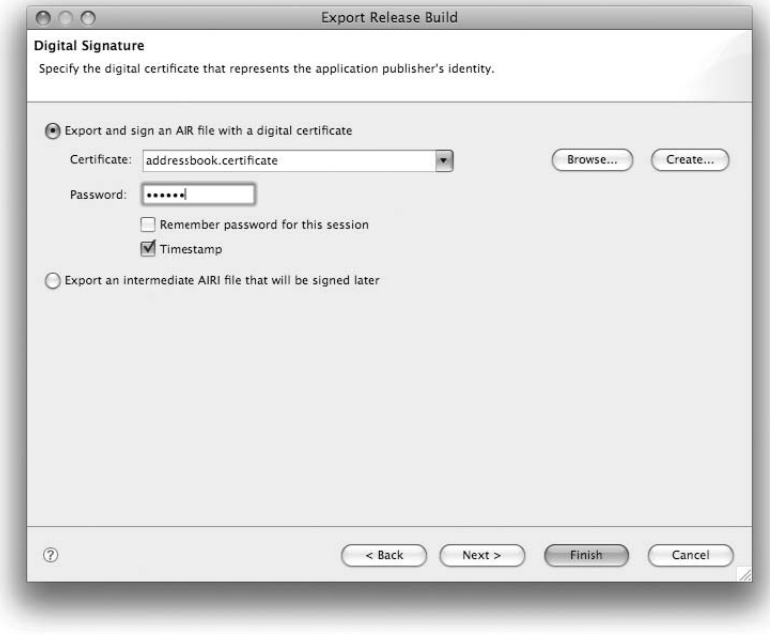

Рис. 15.17. *Экспорт приложения AIR: шаг четвертый*

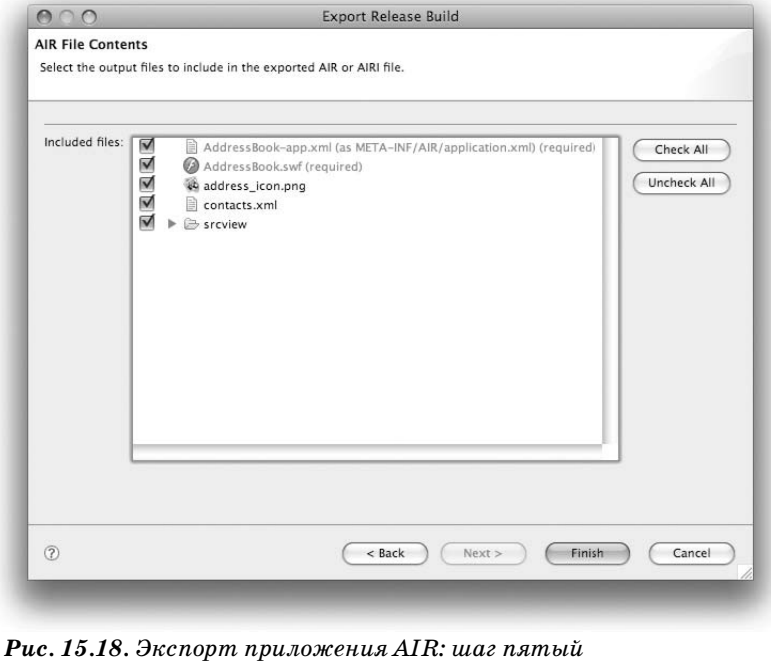

Рис. 15.18. *Экспорт приложения AIR: шаг пятый*

При нажатии на кнопку Finish в корневом каталоге вашего проекта появится новый файл с именем *AddressBook.air*. Этот файл предназначен для установки вашего приложения и его распространения (аналогично файлам с расширением *.dmg* в системе Mac или *.exe* в Windows).

Данный файл можно использовать для установки приложения в том случае, если в системе пользователя уже установлена система Adobe AIR. В противном случае от него не будет никакого толку. Это существенное ограничение, и вам наверняка захочется исправить такую ситуацию, усовершенствовав процесс установки вашего приложения. Оптимальным решением будет использование установочного значка (badge, «значок» или просто бейдж) для Adobe AIR. Он позволяет установить ваше приложение с помощью одного стандартного диалогового окна, размещенного на веб-странице. Пользователям, в чьей системе не установлена среда Adobe AIR, будет предложено загрузить и установить ее, не выходя из этого диалогового окна – таким образом, предлагается простое решение для пользователей, заинтересованных в использовании вашего приложения.

Гладкая установка

Для создания бейджа необходимо несколько файлов. К счастью, большую часть кода можно получить в готовом виде. Если при установке Flex Builder вы оставили настройки расположения по умолчанию, требуемые файлы можно будет найти в следующем месте:

Mac: /Mac/Applications/Adobe Flex Builder 3/sdks/3.0.0/samples/badge

Windows: C:\Program Files\Adobe Flex Builder 3\sdks\3.0.0\samples\badge

В папке badge находится несколько файлов, но для нас представляют интерес следующие из них:

AC_RunActiveContent.js: Файл со сценарием на JavaScript, используемый для автоматического обновления Flash Player.

badge.swf: Данный файл содержит код для автоматической загрузки Adobe AIR при ее отсутствии в системе пользователя.

default_badge.html: Основной HTML-файл, отображающий значок для установки Adobe AIR.

test.jpg: Графическое изображение, используемое для предварительного просмотра вида приложения на странице default_badge.html.

Остальные файлы, расположенные в этой папке, предназначены для создания своего собственного файла *badge.swf*; как правило, в них нет необходмости, тем более в нашем примере.

default_badge.html – это шаблон, который можно использовать для создания своего собственного бейджа на веб-странице. Его можно протестировать, загрузив с помощью броузера. Этот файл *.html* загрузит изображение *test.jpg*, и будет создана кнопка Install Now, нажатие на ко-

торую приведет к началу загрузки приложения – и среды Adobe AIR, если она еще не установлена.

Внимание

Обратите внимание на строчку `My%20Application` (имя приложения в формате, допустимом для URL). Замените ее на название своего приложения, а если в нем встречаются пробелы, замените их на `%20`.

Данный набор файлов – всего лишь шаблон, который необходимо изменить для использования со своими собственными приложениями. Используемое по умолчанию изображение следует заменить на снимок вашего приложения, чтобы дать пользователям представление о том, что они загружают. Кроме того, вам понадобится страница для установки вашего приложения в формате *.air*. Страница *default_badge.html* содержит ссылку на несуществующий файл myapp.air, которую нужно заменить названием файла вашего приложения (в данном случае *AddressBook.air*). Не забудьте также заменить все выражения My Application на имя вашего приложения, поскольку они будут отображены в сообщении под значком в том случае, если Adobe AIR не установлена в системе пользователя. (По умолчанию выводится сообщение «In order to run My Application, this installer will also set up Adobe AIR[1]»).

И, наконец, вы, возможно, захотите разместить код созданного значка на своем собственном веб-сайте. Впрочем, с тем же успехом можно просто разместить ссылку на страницу *default_badge.html*.

Примечание

Если вам кажется, что создание установочного значка – слишком трудоемкое занятие, спешу сообщить вам, что это способ создать кросс-платформенное приложение. При создании настольных приложений другими средствами вам пришлось бы по меньшей мере создать различные версии программы установки для Mac, Windows и Linux. Процесс создания бейджа относительно прост и в этом случае достаточно одной универсальной программы, работающей на базе любой платформы.

По окончании редактирования файла *default_badge.html* разместите все вышеперечисленные файлы, включая инсталляционный файл *AddressBook.air*, на вашем сервере. При загрузке данного HTML-файла в окне броузера отобразится бейдж, как показано на рис. 15.19. Я разместил этот файл на своем сайте по адресу *www.greenlike.com/flex/learning/projects/addressbook/install/default_badge.html*. Этапы процесса установки приложения отображены на рис. 15.20–15.24.

Adobe AIR наделяет ваше приложение дополнительными функциями, такими как возможность перетаскивания объектов с и на рабочий стол

[1] Для запуска приложения My Application программа установки также установит Adobe AIR.

Рис. 15.19. *Установочная коробочка, отображаемая в броузере*

с помощью мыши (drag and drop), хранение информации в локальной базе данных и т. д. Что особенно важно, все приобретенные вами навыки разработки Flex-приложений, работающих в броузере, можно применять и в процессе создания приложений AIR. Если вашей первостепенной задачей является создание именно настольных Flex-приложений, рекомендую вам обратиться к дополнительной литературе, посвященной вопросам разработки приложений AIR с использованием Flex, поскольку столь широкие возможности не могут быть полностью охвачены в данной книге.

Примечание

Множество примеров приложений AIR доступны по адресу *http://labs.adobe.com/technologies/air/samples*.

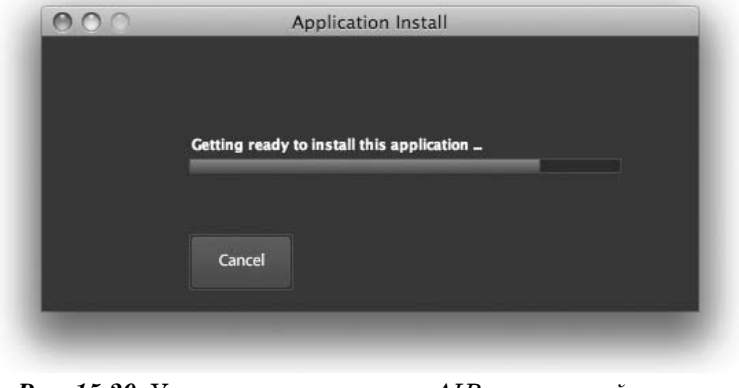

Рис. 15.20. *Установка приложения AIR: шаг первый*

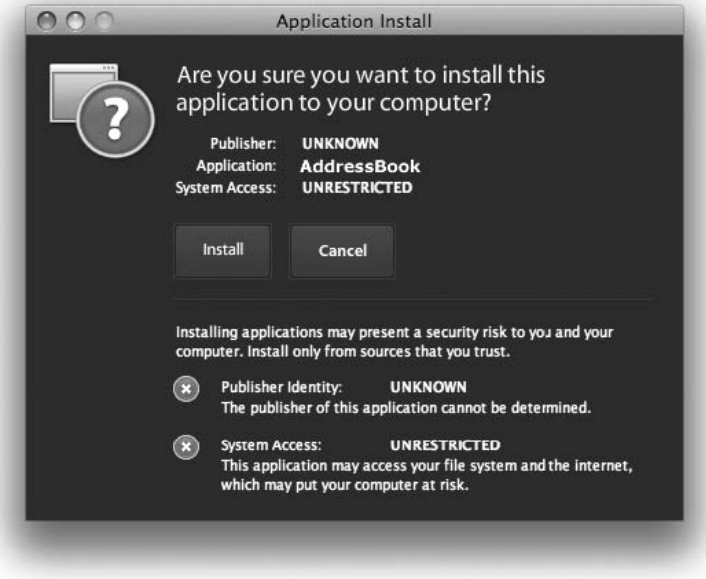

Рис. 15.21. *Установка приложения AIR: шаг второй*

Рис. 15.22. *Установка приложения AIR: шаг третий*

Рис. 15.23. *Установка приложения AIR: шаг четвертый*

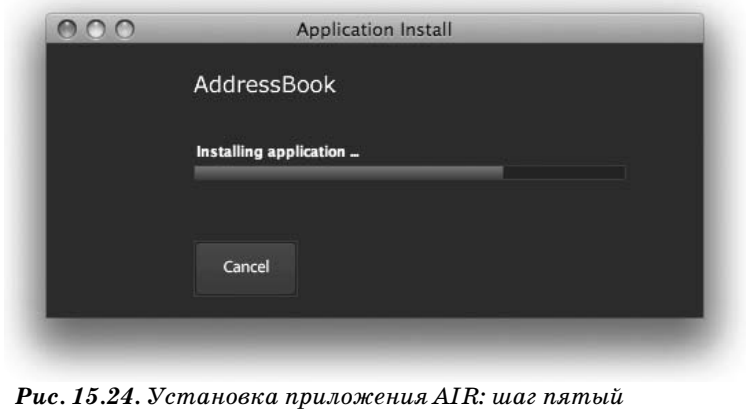

Рис. 15.24. *Установка приложения AIR: шаг пятый*

Секреты Flex-кухни

Вы прошли долгий путь изучения технологии Flex. При создании своих собственных приложений вам может потребоваться помощь. Можно быстро воспользоваться готовым решением. Ознакомьтесь с ресурсом Flex Cookbook по адресу *www.adobe.com/go/flex_cookbook*. Для его использования не нужно быть экспертом – можно просто-напросто использовать предлагаемые фрагменты кода для решения собственных задач (по крайней мере тех, что связаны с разработкой Flex-приложений). Кроме того, вы получите много новых знаний.

При необходимости вы всегда можете приобрести более полный справочник – книгу Flex 3: Cookbook (издательство O'Reilly).

Заключение

Пришло время поздравлений! Вы завершили последний этап изучения процесса создания насыщенных интернет-приложений с помощью технологии Adobe Flex. Познакомившись с основами данной технологии и принципами работы в среде разработки Flex Builder, вы перешли к изучению языков MXML и ActionScript и приобрели ценные навыки, необходимые для создания полноценных приложений, такие как умение подключать к ним данные из внешних источников и владение основными приемами проектирования гибкого пользовательского интерфейса. Вы умеете настраивать внешний вид приложений с помощью фильтров, переходов и таблиц стилей (CSS). В заключительной главе вы также узнали о том, как использовать ваши приложения, – как в веб-среде, так и в качестве обычных настольных приложений. Мне хочется верить, что книга вам понравилась, и в скором времени я надеюсь увидеть ваши отличные приложения, созданные с помощью этой функциональной – и очень увлекательной – технологии.

Алфавитный указатель

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

М

Н

О

П

Р

С

Т

У

Ф

Х

Ц

Ч

Ш

Э

Я