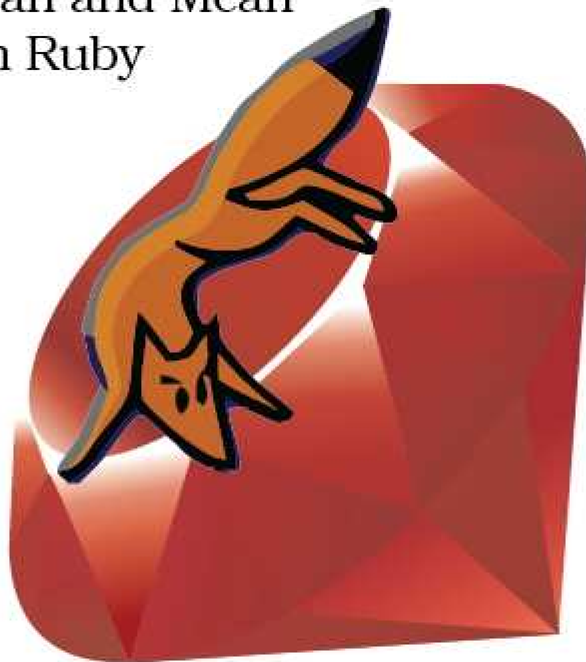
The Pragmatic Programmers

# FXRuby

## Create Lean and Mean GUIs with Ruby



*Lyle Johnson*

The Facets of Ruby Series

## Что читатели говорят о FXRuby

Изучение программированию GUI должно быть легким, но это обычно трудно. Читая эту книгу, я понял, в отличие от этого, что причина это обычно трудно то, что это не забава. Ориентированный на результаты подход к обучению делает изучение FXRuby интересным, и поэтому легким. Эта книга правильно написанное учебное руководство об одном из для Ruby наиболее стабильном графическом инструментарии из его наиболее авторитетного источника.

Чад Фоулер
CTO, InfoEther
Founding Co-director, Ruby Central

FXRuby - богатый, зрелый инструментарий GUI, который Лайл поддерживает и документирует очень хорошо в течение многих лет. С добавлением этой превосходной книги, этот инструментарий становится намного больше применимым.

Хэл Фалтон
Author, The Ruby Way

Мне заплатили, чтобы разработать приложение GUI, используя Ruby в 2003, и я выбрал основанный на FOX/FXRuby как правильный инструментарий из-за исключительного качество привязки и высокого уровеня поддержки Лайлом. Мое единственное сожаление? То, что у меня не было этой книги! Вы открываете на Ваш рабочий стол и онлайновые ссылки загружаются в Вашем браузере, ничто не должно мешать Вам создать удивительное настольное приложение используя Ruby.

Натаниэль Тэлботт
Founder and Developer, Terralien, Inc.

Глубокие знания Лайла в FXRuby гарантируют, что эта привлекательная книга подготовит Вас для создания межплатформенныех GUI за очень небольшое количество времени вообще.

Остин Зиглер
Software Designer and Developer

"FXRuby: Создание простых GUI с Ruby" это правильно написанный текст прямо из первых уст: книга о FXRuby от автора FXRuby. Вы не можете сделать лучше, если, конечно, библиотека не написала книгу непосредственно.

Джереми Макэналли
Developer/technical writer, ENTP

Эта книга - превосходное введение в программирование FXRuby. Лайл сделал хорошую работу, что бы Вы начав с основ шли дальше к более продвинутым темам в правильном темпе.

Даниэль Бергер
Software Engineer, Qwest, Inc

FXRuby
Create Lean and Mean GUIs with Ruby
Lyle Johnson

## ПРЕДИСЛОВИЕ

Инструментарий FOX - библиотека для разработки пользовательских интерфейсов разрабатывается больше десяти лет. FOX стартовал как мой хобби проект, названный "Свободные Объекты для X" (FOX), потому что моей начальной средой была система XWindows.

Одним из ранних применений FOX было в CFD Research Corporation, где Лайл и я работали. Разработчики пользовательского интерфейса в компании были приятно удивленны простым кодированием для разметки их интерфейсов, привыкнув к Мотиву, где при размещении единственной кнопки часто требуется дюжина строк кода. Та же самая задача требует только одной строки кода в FOX. Поддержанная этим успехом, библиотека FOX быстро прошла через много изменений; библиотека была портирована на Microsoft Windows, и была добавлена поддержка 3-D программирования. Все ключевые компоненты создания приложения GUI были введены в FOX.

FOX теперь достиг точки, где разработчики могут записать код и быть уверенный, что это будет компилироваться и работать на многочисленных платформах, от PC, запускающих Windows до машин Unix от Sun и IBM. FOX продолжает расти. За прошлые несколько лет, фокус был на интернационализации и локализации, так же как и поддержки мультипроцессоров.

FOX Toolkit написан на C++, и пока небыло привязок к другим языкам, Вы должны были программировать в C++, чтобы использовать FOX. Теперь, с созданием библиотеки FXRuby, возможности FOX Toolkit стали доступными в языке программирования Ruby.

В этой книге Вы изучите, как создать основанный на FOX графический пользовательский интерфейс с помощью Ruby. В части №1, Вы напишете своё первое маленькое FXRuby приложение, начав с подробных инструкций, как получить FXRuby и установить его в Вашей среде программирования Ruby. Вы пройдёте через нескольких итераций к функциональному приложению, это покажет Вам много важных возможностей программирования в FXRuby.

Во Второй части, даётся в большее количество деталей по событийно-управляемому программированию и как соединить пользовательский интерфейс с исполняемым кодом Ruby. Идя дальше к доступным элементам управления и виджетам, Вы изучите, как использовать компоновщики, для размещения Ваших элементов пользовательского интерфейса (это очень полезная глава, потому что автоматическое размещение - чуждое понятие даже для многих закаленных программистов Windows).

После того, как Вы прочитаете эту книгу, Вы будете в состоянии разрабатывать великолепные пользовательские интерфейсы для Ваших программ на Ruby!

Jeroen van der Zijp (Principal FOX Toolkit Developer)
January 2008

## Признание

Я хотел написать книгу о разработке на FXRuby долгое время. Когда я решил, что наконец готов сделать это, я понял, что хочу работать с Pragmatic Programmers, чтобы сделать эту работу. Большое спасибо Дэйву и Энди которые дали мне эту возможность.

Очевидно, FXRuby не существовал бы, если бы не было FOX Toolkit. Я благодарен моему другу и прежней коллеге Джероен ван дер Зиджп в разрешении мне играть незначительную роль в разработке FOX все эти годы и за все, что я изучил у них в процессе работы.

This book could easily have run off the rails if it weren’t for the hard work and dedication of my editor, Susannah Davidson Pfalzer. Susannah, thanks so much for your attention to detail and your expert guidance as we worked through all of those revisions. The result is so much better than it would have been without your help.

One of the realities of working on a book like this for months at a time is that you get way too close to the text to be objective about it, and you become unable to spot its flaws. For that reason, I owe many thanks to the book’s reviewers: Dan Berger, Joey Gibson, Chris Hulan, Sander Jansen, Chris Johnson, Joel VanderWerf, and Austin Ziegler. Their comments and suggestions were invaluable. Thanks are likewise due to the numerous beta book readers who took the time to point out problems with the early releases of the book.

Finally, thanks to my wife, Denise, for her support and encouragement and for putting up with a frequently distracted husband over the past nine months. We are so going to the beach now that this is done.

Lyle Johnson
January 30, 2008
lyle@lylejohnson.name

# Глава 1

## ВВЕДЕНИЕ

FXRuby - это библиотека для разработки мощных и сложных кроссплатформенных графических интерфейсов пользователей (GUI) для Ваших приложений на Ruby. Он основан на Инструментарии FOX, популярной библиотеке C++ с открытым исходным кодом разработанной Джероеном ван дер Зиджпом. Что это означает для Вас как разработчика приложений - то, что Вы в состоянии написать код на языке программирования Ruby, который Вы уже знаете и любите, в то время как одновременно получаете преимущество в производительности и функциональность полнофункционального, чрезвычайно оптимизированного инструментария C++.

Хотя у FOX нет того же самого уровня узнаваемости имени как у некоторых других инструментариев GUI, он стал доступен с 1997 и всё ещё находится в непрерывной разработке. FXRuby разрабатывался с конца 2000, и первый общедоступный выпуск был в январе 2001. Я был ведущим разработчиком в течение того всего времени, со многим сообществом добровольцев, вносящих патчи по ходу дела. Это - хитрое суждение предположить размер пользовательского сообщества для проекта с открытым исходным кодом, но согласно статистике RubyForge было около 45 000 загрузок FXRuby с начала проекта (и почти 18 000 перед этим, когда это было размещено в SourceForge). На вопросы, отправленные на пользовательский список рассылки FXRuby часто отвечает один Джероен ван дер Зиджп (разработчик FOX), или кто-то из числа старых членов сообщества FXRuby.

## 1.1 Что в этой книге?

Цель этой книги состоит в том, чтобы дать Вам преимущество при разработке GUI приложения с Ruby и FXRuby через комбинацию упражнений учебного руководства и фокусной технической информации.

Это не всесторонняя книга по программированию FXRuby, и это не ссылочное руководство. Почти полный справочник доступен, и он не включён в стандартную поставку FXRuby. Что это книга делает, так это обучает Вас начальным концептуальным препятствиям и снабжает Вас практической информацией для создания Вами своего собственного приложения.

## 1.2 Для кого эта книга?

Эта книга для разработчиков программного обеспечения, которые хотят изучить, как разработать приложение с GUI, используя язык программирования Ruby. Если Вы новичок в Ruby, Вы должны сами изучить его в то время как мы выделим определенные методы программирования Ruby по ходу изложения. Эта книга не предназначена, чтобы учить Вас, как программировать в Ruby. Вам не обязательно быть гуру Ruby, но важно, чтобы Вы были довольны программированием в Ruby, и понимали объектно-ориентированное программирование в общем, прежде, чем продолжить.

Однако не обязательно иметь предшествующий опыт программирования GUI, чтобы читать эту книгу. Поскольку новые темы представлены, мы не будем торопиться, чтобы объяснить, как они вписываются в общую картину и как они касаются вещей, которые Вы встретите в других контекстах. Если у Вас действительно есть некоторый предыдущий опыт по созданию GUI, Вы будете в состоянии использовать эту книгу, чтобы быстро идентифицировать общие черты и различия между этим и другими инструментариями GUI, в которые Вы использовали в прошлом. Независимо от Вашего уровня опыта эта книга поможет Вам не только получить начальный базу но и изучить фундаментальные вещи, который Вы должны понять так, чтобы смогли идти дальше к разработке мощных пользовательских интерфейсов для Ваших приложений.

## 1.3 Как читать эту книгу

Первая часть этой книги начинается с инструкций по установке и затем переходит к расширенному примеру, в котором мы последовательно создадим законченное приложение FXRuby. Это - место, чтобы начать, для тех кто хочет научиться программированию на FXRuby с нуля. Фактически, большинство людей любит создавать приложение по книге.

Если Вам лень набирать примеры на клавиатуре, Вы можете обмануть всех и загрузить исходный код (сжатый архив tar или файл zip) здесь: сайт http://www.pragprog.com/titles/fxruby имеет links для downloads.

Во второй части книги мы повторно рассмотрим некоторые из тем которые мы объясняли при разработке приложения в качестве примера, и мы детально рассмотрим как всё работает. Мы также рассмотрим некоторые дополнительные темы, которые не вошли в пример, но которые все еще важны для Вас, чтобы знать их.

По пути Вы будете видеть различные соглашения, которые мы приняли.

**Код**

Большинство фрагментов кода представлены в законченном рабочем виде, которые Вы можете загрузить.

**hello.rb** (http://media.pragprog.com/titles/fxruby/code/hello.rb)

```
require 'fox16'
app = Fox::FXApp.new
main= Fox::FXMainWindow.new(app, "Hello, World!", :width => 200, :height => 100)
app.create
main.show(Fox::PLACEMENT_SCREEN)
app.run
```

## 1.4 Где получить Помощь

Лучшее место, чтобы получить справку на FXRuby (кроме этой книги, конечно) списки рассылки и различные источники онлайн-документации.

### Mailing Lists

Два различных списка рассылки выделены FXRuby. Announcements list – список для низкого трафика, это прежде всего используется,чтобы уведомить пользователей о новых выпусков FXRuby, в то время users list – список более высокого трафика где происходит общее обсуждение FXRuby программистами. Вы можете найти инструкции по тому, как подписаться на эти списки, так же как и на архивы списка рассылки, на странице проекта RubyForge для FXRuby (http://rubyforge.org/mail/?group_id=300).

В дополнение к спискам FXRuby Вы можете счесть ценным подписаться на регулярный FOX users mailing list. Многие из проблем, которые Вы встретите разрабатывая приложения на FXRuby являются теми же самыми как те, к которым обращаются разработчики, работающие с библиотекой FOX на приложения GUI C++. Для инструкции по тому, как подписаться на пользовательский список рассылки FOX и на архивы этого списка, см. страницу проекта SourceForge для FOX (http://sourceforge.net/mail/?group_id=3372).

### Online Documentation

Несмотря на слухи, есть довольно много онлайн документации и для FOX и для FXRuby, если Вы знаете, где смотреть.

## FOX Documentation Page

В страницах документации на веб-сайте FOX есть много статей по всесторонней информации о темах, таких как менеджеры расположения, значки и изображения, шрифты, и drag and drop (http://www.fox-toolkit.org/doc.html). Эти статьи, имеют много ужасных технических деталей и конечно нацелены на пользователей библиотеки C++, таким образом, они являются не совсем подходящими для начинающих пользователей FXRuby. Как только Вы закончите эту книгу, Вы можете вернуться к этим статьям, чтобы получить более глубокое понимание некоторых механизмов программирования в FOX.

## FOX Community Wiki

FOX Community (http://www.fox-toolkit.net/) - это wiki, написанная разработчиками FOX и для разработчиков FOX. Там находится расширенный список FAQ, учебные руководства и другие виды документации. Много примеров кода приспособлено для разработчиков на C++, которые используют FOX в своих приложениях, но большинство информация там также относится к разработке приложений FXRuby.

## FXRuby User's Guide

Руководство пользователя FXRuby (http://www.fxruby.org/doc/book.html) - действительно мешанина информации о FXRuby, но это действительно обеспечивает довольно исчерпывающую информацию об установке FXRuby. Это также обеспечивает учебные руководства по работе с буфером обмена и как интегрировать drag and drop в Ваши приложения FXRuby.

## API Documentation

Как Вы (вероятно), знали прежде, чем купили эту книгу, это не reference manual. Документация по API для FXRuby всеобъемлюща и в свободном доступе, таким образом нет никакого смысла в попытке копировать этот материал здесь. Для просмотра последней и самой точной документации по API, зайдите на веб-сайт (http://www.fxruby.org/doc/api/). Если Вы устанавливали FXRuby через RubyGems, у Вас должна быть локальная копия документация. Просмотреть документацию HTML можно через RDoc, сгенерированный при установке gem, сначала запустите gem_server:

```
$ gem_server

[2007-05-09 17:18:04] INFO   WEBrick 1.3.1
[2007-05-09 17:18:04] INFO   ruby 1.8.6 (2007-03-13) [i686-darwin8.8.1]
[2007-05-09 17:18:04] INFO   WEBrick::HTTPServer #start: pid=427 port=8808
```

Теперь, укажите в веб-браузере адрес http://localhost:8808/. Просмотрите путем прокрутки listing of installed gems, пока Вы не найдёте запись для FXRuby, и затем щелкните по ссылке [rdoc], чтобы просмотреть документацию.

Другой изящный прием для поиска информации о классах FXRuby или его методов это использование инструмента командной строки ri:

```
$ ri Fox::FXCheckButton#checked?

------------------------------------ Fox::FXCheckButton#checked?
      checked?()
------------------------------------
return +true+ if this check button is in the checked state.
```

ri - это очень удобная команда и конечно применима для любой библиотеки Ruby, которую Вы установили, включая ядро и стандартные библиотечные классы и методы. Если Вы устанавливали FXRuby используя RubyGems, это должно было автоматически генерировать и установить ri документацию для FXRuby. Если Вы устанавливали FXRuby непосредственно из source tarball, или через некоторые другие средства, Вы, возможно, должны генерировать и установить ri документацию самостоятельно прежде, чем Вы сможете успешно использовать команду ri для поиска документации по FXRuby.

Если по некоторым причинам ri должным образом не установлен на Вашей системе, сделайте себе одолжение и получите его и используйте в своей работе!

## 1.5 A Word About Versions

Обсуждение и примеры в этой книге основаны на FXRuby 1.6, это текущий выпуск во время написания этой книги.

Вообще говоря, в Ваших интересах использовать последнюю доступную версию FOX и FXRuby, потому что у тех версий будут последние исправления ошибок и улучшения. Отметьте, однако, что старшее число версии для данного выпуска FXRuby указывает на номер основной версии из выпуска FOX, с которым это является совместимым; например, FXRuby 1.6 предназначен для использования с FOX 1.6. Это важно потому что последний выпуск FOX часто тегируется как нестабильный выпуск или выпуск "разработки", и эти версии могут не работать с последним выпуском FXRuby.

Теперь давайте начнем!

# Часть I
# Создание приложения FXRuby

## Глава 2
## Первый старт с FXRuby

Эта глава - Ваш быстрый старт к разработке приложений FXRuby. Мы потратим несколько страниц что бы рассмотреть FXRuby и показать как он работает с FOX перед тем как перейти к инструкциям по установке FXRuby на некоторых из большинства популярных операционных систем. Мы расскажем как создать простое приложение “Привет, Мир!”, таким образом, Вы увидите типовую структуру приложения FXRuby и убедитесь, что программное обеспечение должным образом установлено.

FXRuby упакован как модуль расширения для Ruby. Это означает, что это - библиотека C++, которую интерпретатор Ruby загружает во время выполнения, включая набор новых классов Ruby и констант. Рисунок 2.1 иллюстрирует отношения между Вашим кодом программы (написанного на Ruby), расширением FXRuby, библиотекой FOX и операционной системой. FXRuby похож на любую другую “чистую Ruby” библиотеку которую Вы могли бы использовать; различие в том, что исходный код этой библиотеки написан не на Ruby. FXRuby представляет всю функциональность библиотеки FOX, но это не больше чем только простая "обертка" вокруг API. FXRuby использует расширенные возможности языка Ruby для предоставления высокоуровнего интерфейса FOX. Например, очень утомительно писать весь код на C++ для отображения пользовательского интерфейса в традиционных приложениях FOX. В FXRuby, Вы в состоянии соединить блок Ruby непосредственно с виджетом посредством нескольких строк кода.

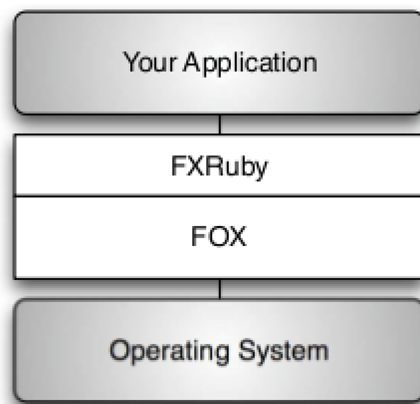



Figure 2.1: Relationship between the operating system, FOX, FXRuby, and your Ruby application

Когда я сначала начинал работать над FXRuby, не было особого разнообразия продуктов межплатформенной разработки GUI для Ruby, кроме встроенной поддержки Tk. Сегодня, ситуация изменилась. Если Вы ищете межплатформенное GUI, очень хорошо поддерживается привязка Ruby для GTK + и QT, и для других популярных GUI, таких как wxWidgets и FLTK. Такой широкий выбор побуждает кого-то отправить вопрос на список рассылки Ruby с вопросом какой GUI является лучшим выбором?

Точно так же как, вопросы какай редактор лучше, операционная система, или язык программирования. Вопрос какой GUI является "лучшим", зависит от того что Вы ищете. Вместо того, чтобы пытаться склонить Вас к определенному выбору, я поощряю Вас, по крайней мере, экспериментировать с разными продуктами, что бы Вы сами решили, что могло бы подойти для Ваших потребностей.

Для начала есть много вещей, которые Вы можете сделать с FOX и FXRuby. Если Вы хотите соединить простой фронтэнд GUI с командной строкой, FXRuby, конечно, отвечает всем требованиям. Так как FOX обеспечивает поддержку всех стандартных видов элементов пользовательского интерфейса, таких как labels, кнопки, и текстовые поля, это - также лучший выбор для того, чтобы быстро разработать основанное на формах GUI. Расширенная функциональность FOX отличает его от некоторых из его конкурентов. Обширная поддержка вывода и манипулирование данными изображений делает это идеалом для того, чтобы разработать визуально богатые пользовательские интерфейсы, и благодаря встроенной поддержки OpenGL, FOX также стал популярным выбором для приложений, требующих 3-D функциональности в визуализации.

Другая характеристика, которую важно рассмотреть, это то, что GUI может использовать легкие или тяжелые виджеты в зависимости от Ваших предпочтений. FOX использует легкие (или non-native) виджеты. Это означает что, основанное на FOX приложение полагается только на основные базовые возможности платформы, на которой оно работает, чтобы создать пользовательский интерфейс, вместо того чтобы обеспечивать обёртку классами и методами вокруг существующих платформенных виджетов. У этого подхода есть несколько преимуществ:

1. Так FOX определяет поведение виджетов, которые создаются, вместо того, чтобы полагаться на поведение родных виджетов, то поведение является непротиворечивым для разных платформ.

2. Так как FOX рисует свои собственные виджеты, Ваше приложение будет смотрется одинаково, независимо от платформы на которой это работает на (Некоторые люди считают это недостатком использования легких виджетов).

3. Так как FOX был разработан с нуля, чтобы быть объектно-ориентированным и расширяемым, у Вас есть намного больше гибкости с точки зрения подклассов существующих виджетов FOX, чтобы создавать Ваши собственные специализированные виджеты. Эта большая гибкость будет потеряна, если Вы используете библиотеки GUI, которые являются оберткой вокруг некоторого другого наследуемого инструментария.

4. Так как FOX сокращает количество уровней кода, через которые Вы должны пройти, основанные на FOX приложения имеют тенденцию быть более производительными и отзывчивыми.

Наконец, не последний вопрос как определенная библиотека GUI лицензируется. Например, некоторые библиотеки GUI требуют, чтобы Вы купили коммерческую лицензию для разработки, если Вы хотите использовать их, чтобы разработать собственные приложения (с закрытым исходным кодом). FOX и FXRuby – оба лицензируются в соответствии с LGPL (http://www.gnu.org/licenses/lgpl.html), которая разрешает использование этих библиотек и в свободном и в проприетарном (коммерческие) приложении.

Теперь, давайте начнем с установки FXRuby и затем используем его для создания простой программы “Привет, Мир!”.

## 2.1 Установка FXRuby

Установка FXRuby является немного более стимулирующей чем установка другой Ruby библиотеки, потому что она написана в C++ и должна поэтому быть скомпилирована в совместно используемую библиотеку, которую интерпретатор Ruby может загрузить во время выполнения. Это далее усложняется фактом, что есть несколько зависимостей, включая библиотеку FOX, на которой FXRuby базируется и как правило несколько сторонних библиотек для поддержки для различных графических форматов.

Хорошие новости - это, если Вы устанавливаете FXRuby на Windows или MacOS X, установка является довольно безболезненной. Если Вы устанавливаете FXRuby на

Linux, у Вас будет немного больше работы, но шаги довольно легки и Вы можете рассчитывать на поддержку от FOX и FXRuby сообщества при любых проблемах с установкой, которые могут возникнуть.

Следующие разделы освещают некоторые основные шаги по установке FXRuby на наиболее распространенных операционных системах. Для некоторых исключительных ситуаций, мы обратимся к онлайн-документации для FOX и FXRuby, где есть самая полная и актуальная информация о проблемах установки:

1. Для всесторонних инструкций по установке библиотеки FOX см. инструкции по установке в веб-сайте FOX http://www.fox-toolkit.org/install.html.

2. Для всесторонних инструкций по установке FXRuby см. в FXRuby User's Guide http://www.fxruby.org/doc/build.html

## Установка в Windows

Если Вы использовали One-Click Installer for Ruby on Windows http://rubyinstaller.rubyforge.org/wiki/wiki.pl, Вы уже имеете версию установленного FXRuby. Однако, недостаток версии FXRuby в данной установке: «одним кликом мышки» это то, что она иногда отстает от последней выпущенной версии. Вы должны обновится командой gemupdate:

```
C:\> gem update fxruby
```

Если Вы установили Ruby некоторыми другими средствами, Вы должны скомпилировать и FOX и FXRuby вручную. Если Вы используете Unix подобную среду для Windows, такую как Cygwin или MinGW, Вы должны следовать инструкциям в Разделе 2.1, "Установка в Linux", в следующем разделе, для выполнения этой задачи. Если Вы используете Microsoft (или некоторого другого поставщика) средства разработки, Вы должны обратиться к онлайновой документации упомянутой в начале этой главы.

## Установка в Linux

Получение FOX и FXRuby для Linux, может стать процессом отнимающим много времени. Вам может повезти: многие свежие дистрибутивы Linux включают пакеты FOX и/или FXRuby. Когда это имеет место, Я строго рекомендую, чтобы Вы использовали эти пакеты, чтобы избежать неизбежной головной боли по поиску зависимостей и установки их вручную. Например, если Вы пользуетесь Ubuntu Linux и включили компоненту "universe" в репозитарии, Вы можете установить FOX непосредственно установив пакет libfox-1.6-dev:

```
$ sudo apt-get install libfox-1.6-dev
```

Так как Ubuntu Linux не обеспечивает пакет для FXRuby по умолчанию, Вы будете нуждаться в устанавке его через gem, как описано позже в этом разделе.

Если Вы используете дистрибутив Linux, который еще не включает FOX или FXRuby как стандартный пакет установки, Вы должны будете искать сторонние пакеты или (худший случай) создавать их из исходного кода. В этом случае, сначала загрузите последний релиз FOX 1.6 с сайта FOX: http://www.fox-toolkit.org/download.html

Имя файла fox-1.6.29.tar.gz. Используйте команду tar, для распаковки:

```
$ tar xzf fox-1.6.29.tar.gz
```

Это действие создаст каталог, названный fox-1.6.29. Войдите в эту директорю и затем использйте последовательность configure, make, make install для установки FOX:

```
$ cd fox-1.6.29
$ ./configure
«output of "configure" command»
$ make
«output of "make" command»
$ sudo make install
«output of "make install" command»
```

Теперь, когда Вы скомпилировали и установили FOX, Вы готовы установить FXRuby. Самый простой способ это использовать команду для gem install:

```
$ sudo gem install fxruby --remote
Bulk updating Gem source index for: http://gems.rubyforge.org
Building native extensions. This could take a while...
```

Как указано в сообщении, этот процесс может занять продолжительное время.

## 2.2 Instant Gratification

Теперь, когда у Вас есть установленный **FXRuby**, мы перейдём к более интересной теме. Мы начнём с простого FXRuby приложения в этом разделе и затем пойдём дальше к более сложному примеру в следующих главах, которые научат Вас тому, как структурировать реальные приложения **FXRuby**.

"Hello, World!"

В проверенной временем традиции книг по программированию, мы начнем с версии FXRuby "Привет, Мир!". Позвольте себе начать с абсолютно пустого минимума и удостоверьтесь, что это работает.

http://media.pragprog.com/titles/fxruby/code/hello.rb

```
require 'fox16'
```

---

**Установка Среды RubyGems**

Если Вы установили FXRuby используя RubyGems, пример программы в этой книге, возможно, не заработает должным образом, если Вы не сказали Ruby, чтобы он автоматически загрузил runtime RubyGems и использовал библиотеки сохранённые в репозитарии RubyGems. Есть обсуждение различных вариантов этого в RubyGems Users Guide в http://rubygems.org/read/chapter/3; я лично предпочитаю устанавливать RUBY-OPT environment variable как описано в этом обсуждении.

Помните, если Вы используете Ruby 1.9.0 или выше, RubyGems runtime полностью интегрирован с интерпретатором Ruby, таким образом, эти предосторожности не обязательны.

---

Уже чувствует себя хорошо, не так ли? Эта строка импортирует модуль FOX и всё его содержимое в интерпретатор Ruby. Имя функции "fox16", потому что мы хотим использовать версию 1.6 FXRuby, а не одну из более ранних версий.

Теперь, только короткая программа, ну и юмор у меня: сохраните этот файл как hello.rb, и выполните это теперь:

```
$ ruby hello.rb
```

Ruby что-то делает в течение нескольких секунд и затем спокойно возвращаются в command prompt, Так и должно быть. Это – все, что программа должна сделать, если FXRuby установлен правильно. Если с другой стороны Вы видите одно или больше сообщений об ошибках, остановитесь тут же и выясните что случилось (См.

http://www.fxruby.org/doc/build.html для получения детальной информации). Одна типичная проблема, которая неожиданно возникает во время выполнения, имеет отношение к установке среды RubyGems.

Затем, создайте экземпляр класса **FXApp**, который определен в модуле **FOX**:

```
app = Fox::FXApp.new
```

**FXApp** короток для "приложения". Объект приложения ответственен за цикл событий (event loop), так же выполняет большую работу за сценой программы в **FXRuby**. Это - связующее звено, которое скрепляет все. Пока, тем не менее, достаточно знать, что каждая программа **FXRuby**, которую Вы пишете, будет нуждаться в создании объекта **FXApp**.

Приложение нуждается в главном окне, так что давайте добавим его далее:

```
main = Fox::FXMainWindow.new(app, "Hello, World!",
:width => 200, :height => 100)
```

Теперь Вы видите один из примеров использования объекта **FXApp**. Передавая его как первый параметр **FXMainWindow.new()**, Вы говорите что Ваше приложение ответственно за основное окно. Второй параметр - заголовок главного окна и будет выведен на экран в строке заголовка окна. Вы также определяете начальную ширину и высота главного окна, в пикселях. Есть много параметров, которые Вы могли бы определить в главном окне, но пока хватит этого.

Далее, добавьте вызов в метод **create()**. Это гарантирует что все ресурсы для Вашего приложения будут созданы. Мы обсудим это более детально позже. Пока, только знайте, что это Вы должны будете делать всегда в любом приложении **FXRuby**:

```
app.create
```

Затем, вызовите **show()** в главном окне с параметром **PLACEMENT_SCREEN**, для отрисовки на экране, когда программа начнёт работать:

```
main.show(Fox::PLACEMENT_SCREEN)
```

Параметр размещения **PLACEMENT_SCREEN** - это запрос для центрирования окна на экране, когда оно показывается в первый раз.

В документации API для класса **FXTopWindow** (базовый класс для FXMainWindow) перечислены все параметры, которые Вы можете передать в метод **show()**.

Наконец, вызовите **run()** для объекта **FXApp** что бы запустить цикл основного приложения. Ваша готовая программа должна быть похожей на это:

```
require 'fox16'
app = Fox::FXApp.new
main = Fox::FXMainWindow.new(app, "Hello, World!" ,
:width => 200, :height => 100)
app.create
main.show(Fox::PLACEMENT_SCREEN)
app.run
```

Теперь Вы можете выполнить программу, как любую типичную программу Ruby:

```
$ ruby hello.rb
```

Ваш результат должен быть таким как окно, показанное в рисунке 2.2, который является снимком экрана программы, работающей в Windows.

Figure 2.2: “Hello, World!” on Windows

## Idiomatic FXRuby Programs

Если я собирался записать новую программу FXRuby с нуля, это не вполне то, что я хотел бы видеть. Есть несколько идиом, которые довольно распространены в программах FXRuby, и всей остальной части примеров, которые Вы увидите в этой книге. Прежде всего, распространено включать модуль FOX в глобальном пространстве имен Ruby так, чтобы Вы не использовали полностью определенные имена для классов FXRuby и констант повсюду в Вашей программе.

С этим изменением hello.rb становится немного проще для чтения:

```
require 'fox16'
include Fox

app = FXApp.new
main = FXMainWindow.new(app, "Hello, World!", :width => 200, :height => 100)
app.create
main.show(PLACEMENT_SCREEN)
app.run
```

Вообще говоря, эта практика могла привести к столкновениям между именами определенными в модуле FOX и именами, определенными в других модулях, но на практике я никогда не встречался с этой проблемой.

Другое изменение, которое Вы можете произвести, реорганизовать главное окно приложения как подкласс FXMainWindow:

```
require 'fox16'
include Fox

class HelloWindow < FXMainWindow
      def initialize(app)
            super(app, "Hello, World!", :width => 200, :height => 100)
      end
      def create
            super
            show(PLACEMENT_SCREEN)
      end
end

app = FXApp.new
HelloWindow.new(app)
app.create
app.run
```

Займите минуту или две, чтобы сравнить эту итерацию с предыдущей, и удостоверьтесь, что Вы понимаете сущность изменения. Отметьте, что все, что вы делали для настройки главного окна было перемещено в подкласс HelloWindow, включая вызов show() после того, как он был создан.

Мы увидим в последующих примерах программы, что это удобно, чтобы фокусировать управление приложением пользовательского главного окна как класса.

Как заключительная модификация, переместите конструкция HelloWindow и FXApp в блок запуска:

```
require 'fox16'
include Fox

class HelloWindow < FXMainWindow
      def initialize(app)
            super(app, "Hello, World!", :width => 200, :height => 100)
      end

      def create
            super
            show(PLACEMENT_SCREEN)
      end
end
if __FILE__ == $0
      FXApp.new do |app|
            HelloWindow.new(app)
            app.create
            app.run
      end
end
```

Я также использовал этот шаг, чтобы показать как работает блочная форма конструктора FXApp. Это то, что Вы можете сделать с любым классом FXRuby, когда Вы хотите сделать некоторую дополнительную инициализацию. Ни одно из этих изменений не меняет основную работу программы, но они служат, чтобы демонстрировать типичную структуру программ FXRuby.

Возможно, не похоже на это, но мы охватили многие вопросы в этой главе. Мы устанавливали FXRuby и гарантировали, что он работает должным образом. Мы также разработали простую, но функциональную программу, чтобы познакомится с основами, которым будет следовать каждое приложение FXRuby. В процессе нескольких изменений кода мы видели, что классы, которые обеспечивает FXRuby, могут быть разделены на подклассы и настроены точно так же как любой другой класс Ruby. Теперь мы готовы взяться за разработку более сложного проекта, который мы будем создавать в следующих главах.

# Глава 3

# The Picture Book Application

Теперь, когда Вы установили FXRuby и получили работающую начальную тестовую программу, пора идти дальше к чему-то более стимулирующему. В сдедующих нескольких главах, мы собираемся разработать приложение менеджера библиотеки фотографий, Picture Book, используя FXRuby.

Одной из наиболее трудных задач при писании этой книги было выбрать пример приложения, который я мог бы использовать, чтобы продемонстрировать разработку на FXRuby. Я не большой поклонник изобретения колеса и уже есть много прекрасных приложений для организации фотоальбома. Цель этого примера не создание лучшего из всего, что когда-либо существовало, а научиться как использовать инструменты, которые предоставляет FXRuby, чтобы создать более сложное GUI приложение.

## 3.1 What Picture Book Does

Как отмечено во введении в эту главу, мы стремимся к приложению, которое будет включать много функций, которые Вы хотели бы вставить в Ваши собственные приложения, при сохранении полного контроля над контекстом приложения. Одна из самых важных вещей, которую Вы освоите, это то, как объединить мощные менеджеры расположения (компоновщики) FOX, такие как **FXMatrix**, **FXSplitter**, и **FXSwitcher** и создать сложное форматирование. Вы также научитесь использовать встроенные виджеты, такие как **FXList** и **FXImageFrame**, чтобы создать настроенные, специализированные представления. Вы освоите приемы для того, чтобы использовать FOX's image manipulation и возможности отображения. К тому времени, когда мы закончим приложение, Вы освоите детали и принципы разработки приложений в FXRuby.

Давайте примем некоторые решения относительно основной функциональности приложения Picture Book. Мы желаем создать программу для организации набора существующих цифровых фотографий, сохраненных на диск в один или более именованные альбомы. Я воображаю пользовательский интерфейс примерно как на рисунке 3.1. Когда программа запускается, Вы должны видеть список существующих альбомов с левой стороны главного окна, и если Вы выбираете один из тех альбомов, область на правой стороне должна вывести на экран все фотографии в этом альбоме.

Давайте предусмотрим, что пользователь должен быть в состоянии создать новые альбомы и добавлять фотографии к этим альбомам. Мы предусмотрим еще некоторые расширенные функции, такие как редактирование фотографий и совместное использование Я надеюсь, что когда Вы закончите читать эту книгу, у Вас появятся некоторые идеи о том как реализовать и другие функции.

Одно решение, которое мы должны будем принять, имеет отношение к тому как фотографии будут храниться. Первый способ оставить импортированные фотографии там где они находятся изначально и только сохранять ссылки на их расположение на диске. Преимущество этого подхода состоит в том, что Вы можете хранить свои фотоальбомы на переносных устройствах, таких как внешние жесткие диски или DVD. Второй способ - это фактически сделать копии импортированных фотографий и спрятать их в расположении, известном только приложению.

Последний способ (делающий копии импортированных фотографий) представляет некоторую сложность, независимо от того, является ли это лучшим или худшим выбором. Мы пойдём более простым путём и будем использовать пути к существующим файлам фотографий на диске.

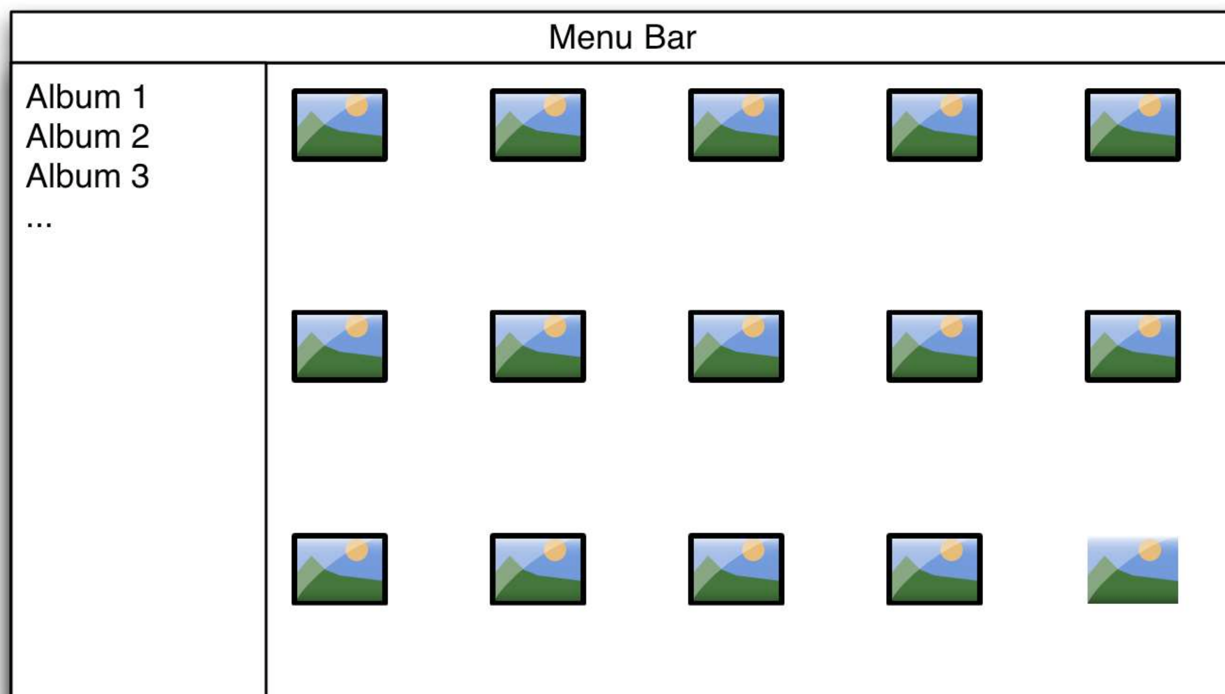



Figure 3.1: User interface concept for Picture Book application

## 3.2 Application Data

Теперь, когда мы изобразили схематически некоторые из предварительных требований к приложению, мы должны рассмотреть структуру данных. Мы собираемся (свободно) принять Model-View-Controller2 (MVC) стиль архитектуры для приложения Picture Book, которая просто означает проблемно-ориентированные данные (а именно, фотографии, альбомы, и списки альбомов), представлены одним набором классов в то время как элементы пользовательского интерфейса (фотография, альбом, и представления списка альбомов) представлены различным набором классов. Этот подход решает много проблем с которыми разработчики программного обеспечения сталкиваются когда данные приложения и пользовательский интерфейс слишком сильно связаны. Мы будем использовать немного измененную версию традиционного образца MVC, когда компоненты пользовательского интерфейса управляют и представлением и управлением.

Мы представим классы модели (M. в MVC) здесь, так как они довольно строгие и они не будут изменяться очень во время разработки приложения. Мы будем говорить больше о запуске классов представления в следующей главе.

Давайте начнём с просмотра единственной фотографии. Мы знаем, что для этого нам надо иметь ссылку на файл на диске, таким образом, мы должны сохранить путь к файлу. Может быть потом мы захотим сделать с фотографией нечто большее, но позвольте нам начать пока с этого.

Http://media.pragprog.com/titles/fxruby/code/picturebook_a/photo.rb

```
class Photo
      attr_reader :path

      def initialize(path)
            @path = path
      end
end
```

Что мы хотим сказать об альбоме? У него должен быть заголовок, такой как “Каникулы 2007,” и он должен содержать набор фотографий. Мы будем нуждаться в методах, чтобы добавлять фотографии к альбому и выполнять итерации по фотографиям в альбоме. Мы, возможно, должны сказать больше об этом больше, но пока представим первую редакцию класса Альбома:

Http://media.pragprog.com/titles/fxruby/code/picturebook_a/album.rb

```
class Album
	attr_reader :title

	def initialize(title)
		@title = title
		@photos = []
	end

	def add_photo(photo)
		@photos << photo
	end

	def each_photo
		@photos.each { |photo| yield photo }
	end
end
```

Наконец, мы нуждаемся в классе для того, чтобы управлять списком альбомов. После нашего образца классов для Photo и Album, мы собираемся начать с действительно основного класса AlbumList и затем будем добавлять к нему всё необходимое. Наша начальная реализация имеет методы для добавления и удаления альбомов, также как и итерации по альбомам в списке:

http://media.pragprog.com/titles/fxruby/code/picturebook_a/album_list.rb

```
class AlbumList
	def initialize
		@albums = []
	end

	def add_album(album)
		@albums << album
	end

	def remove_album(album)
		@albums.delete(album)
	end

	def each_album
		@albums.each { |album| yield album }
	end
end
```

Теперь, когда мы разработали предварительные реализации трех классов, мы можем идти дальше к созданию пользовательского интерфейса непосредственно. Отметьте, что не необходимо полностью определить классы модели прежде Вы начинаете разрабатывать пользовательский интерфейс, особенно если Вы принимаете итерационный подход, как мы для этого приложения.

## 3.3 Let's Code

Теперь, когда мы имеем общее представление о том, что мы хотим, чтобы программа сделала и какие данные мы собираемся использовать в качестве модели, Вам вероятно, не терпится поработать над первой итерацией пользовательского интерфейса. Мы теперь стоим перед вопросом с чего и как начать. Что делать дальше?

Нет правильного ответа на этот вопрос. В течение долгого времени, в процессе знакомства с разработкой приложений FXRuby, Вы будете совершенствовать знания и навык. Вы должны погрузиться в новое приложение с нуля и быстро создать свою функциональность так, как Вы предпочитаете это делать. Лично мне, однако, привычней начинать с самого простого решения и затем на основе его идти к заключительной цели. По этой причине мы начнём, создавая версию Picture Book с самой простой вещи: выведем на экран единственную фотографию.

# Глава 4

# Take 1: Display a Single Photo

Мы собираемся начать разрабатывать приложение так просто насколько возможно, чтобы быстро получить что-нибудь и увидеть некоторые результаты. Первая задача, состоит в том, чтобы вывести на экран единственную фотографию. Чтобы сделать это, мы собираемся создать наш первый класс представления, PhotoView, как подкласс существующего виджета FXRuby. Вы увидите, что классы представления должным образом инициализированы и расположены в корректном месте в главном окне. Мы увидим FOX's image display capabilities посредством класса **FXJPGImage**.

## 4.1 Get Something Running

К концу нашей программы “Привет, Мир!” созданной в Главе 1, мы установили на что похоже основное приложение FXRuby, так что давайте создадим подобную структуру для приложения Picture Book. Начните и определите класс PictureBook как подкласс **FXMainWindow**. Ваш код должен напоминать следующее:

```
require 'fox16'
include Fox
class PictureBook < FXMainWindow
	def initialize(app)
		super(app, "Picture Book" , :width => 600, :height => 400)
	end

	def create
		super
		show(PLACEMENT_SCREEN)
	end
end

if __FILE__ == $0
	FXApp.new do |app|
		PictureBook.new(app)
		app.create
		app.run
	end
end
```

Сохраните этот файл как picturebook.rb, и затем выполните его, чтобы удостовериться что все до сих пор работает:

```
$ ruby picturebook.rb
```

Вы должны видеть пустое главное окно с именем приложения «Picture Book», в строке заголовка. Даже при том, что мы не ожидаем от программы что нибудь большего в этой точке, это предоставляет нам некоторую уверенность в том, что наша рабочая среда установлена должным образом. Теперь давайте идти дальше к чему-то немного более интересному.

## 4.2 Create the View

Теперь, когда главное окно на месте, далее создадим представление для единственной фотографии. Мы собираемся изучить, как создать пользовательский класс представления как подкласс одного из встроенных виджетов FXRUBY и посмотрим, как поместить этот виджет в главном окне.

Есть много различных виджетов в библиотеке **FXRuby**, которые способны отображать изображения, но для нашего примера мы будем использовать **FXImageFrame**. Виджет **FXImageFrame** - простой виджет чья единственная цель вывести на экран объект **FXImage**. У него действительно нет никакой другой функции чем эта. Вашим первым побуждением могло бы быть использование этого фрейма непосредственно для отображения, но мы будем действовать иначе разделив **FXImageFrame** на подклассы, что обеспечивает нас немного большей гибкостью с точки зрения обеспечения специализированной функциональности.

Создайте новый документ в своем редакторе, и установите определение для класса PhotoView:

Http://media.pragprog.com/titles/fxruby/code/picturebook_a2/photo_view.rb

```
class PhotoView < FXImageFrame
	def initialize(p, photo)
		# We'll add code here soon...
	end
end
```

Смотрите на *initialize()* метод для PhotoView. Так как PhotoView подкласс от **FXImageFrame**, самая первая вещь которую мы должны сделать внутри метода *initialize()* PhotoView – вызов метода *initialize()* базового класса.

Наш метод *initialize()* для класса PhotoView будет использовать *super()*, чтобы вызвать реализацию *initialize()* из **FXImageFrame**. Этот важный шаг необходимо помнить всякий раз, когда Вы разделяете класс FXRuby на подклассы, чтобы пользоваться ими: убедитесь, что вызвали метод *initialize()* от базового класса Вашей переопределенной версии. Некоторые языки программирования, как C++ и Java, будут автоматически вызывать конструктор базового класса по умолчанию для Вас; Ruby не из этих языков!

Теперь, если Вы посмотрите документацию API для класса **FXImageFrame** (http://www.fxruby.org/doc/api/classes/Fox/FXImageFrame.html) то увидите, что первые два параметра метода *initialize()* необходимые параметры – нет никаких значений по умолчанию для них. Первый параметр - родительский (контейнерный) виджет для фрейма изображения, и второй ссылка на изображение, которое он выводит на экран. Можно пока не волноваться насчёт других параметров, что мы могли бы передать в *initialize()*; мы примем их значения по умолчанию.

Условно, первый параметр метода *initialize()* виджета - родительский виджет, так что передаём в первом аргументе родителя PhotoView. Мы можем это сделать через *super()*. И так как цель PhotoView состоит в том, чтобы вывести на экран фотографию, мы действительно хотели бы передать фото как второй параметр. Мы не можем указать второй параметр через *super()*, потому что класс **FXImageFrame** не знает что-либо о нашем классе Photo. Фактически, согласно документации API, метод **FXImageFrame.new()** ожидает объект **FXImage** вместо этого. Так, как мы получим один из этих объектов **FXImage**?

Не торопитесь. Как оказывается, мы можем только передать *nil* для изображения в фрейме. Единственное последствие этого решения – фрейм изображения не будет ничего выводить на экран. Мы исправим эту проблему в следующей итерации. Пока, измените метод **initialize()** для PhotoView так, чтобы это было похоже на это:

```
class PhotoView < FXImageFrame
	def initialize(p, photo)
		super(p, nil)
	end
end
```

Теперь мы должны привязать это в к нашему главному окну. Возвратитесь к picturebook.rb, измените метод *initialize()* для PictureBook, чтобы создать

объект Photo, соответствующий некоторой фотографии, имеющейся у Вас. Добавьте PhotoView для той фотографии. Я использую shoe.jpg, которая является изображением обуви, которую моя племянница оставила в прошлый раз, когда она навещала нас, но любой JPEG, который Вам нравится, должен работать. Ваш метод *initialize()* для PictureBook должен выглядеть примерно так:

```
def initialize(app)
	super(app, "Picture Book" , :width => 600, :height => 400)
	photo = Photo.new("shoe.jpg" )
	photo_view = PhotoView.new(self, photo)
end
```

Передавая в **self** как первый параметр в вызове в *PhotoView.new*, мы говорим что объект PictureBook (главное окно нашего приложения) родитель для PhotoView.

```
require 'fox16'
include Fox

require 'photo'
require 'photo_view'

class PictureBook < FXMainWindow
	def initialize(app)
		super(app, "Picture Book" , :width => 600, :height => 400)
		photo = Photo.new("shoe.jpg" )
		photo_view = PhotoView.new(self, photo)
	end

	def create
		super
		show(PLACEMENT_SCREEN)
	end
end

if __FILE__ == $0
	FXApp.new do |app|
		PictureBook.new(app)
		app.create
		app.run
	end
end
```

Выполните программу и посмотрите, что она Вам покажет:

```
$ ruby picturebook.rb
```

Вы будете все еще видеть пустое главное окно; это потому что у фрейма изображения еще нет FXImage, чтобы вывести на экран. Потребуется время, чтобы исправить эту проблему.

## 4.3 Construct an Image from a File

FXRuby поддерживает отображение на экран много различных видов данных, включая все главные форматы, такие как BMP, GIF, JPEG, PNG и TIFF. Мы обсудим эту функциональность более подробно в **Главе 11**, «**Создание Визуально Богатых Пользовательских Интерфейсов**», на странице 142. Пока, мы попытаемся изучить, как использовать встроенный класс **FXRUBY FXJPGImage** для создания экранного изображения непосредственно из файла JPEG на диске и затем присвоить изображение экземпляру нашего класса PhotoView.

Изображение представлено экземпляром класса **FXImage** или, что более обычно, одним из его подклассов, таких как **FXJPGImage**. Давайте запишем некоторый код, чтобы загрузить данные изображения из файла на диске и затем создать объект **FXJPGImage** из него. Возвратитесь в своем редакторе к классу PhotoView, и добавьте следующий метод:

Http://media.pragprog.com/titles/fxruby/code/picturebook_a/photo_view.rb

```
def load_image(path)
	File.open(path, "rb" ) do |io|
		self.image = FXJPGImage.new(app, io.read)
	end
end
```

Первая строка *load_image()* использует форму "транзакции" *open()* для гарантии, что файл будет закрыт должным образом, когда мы закончим с ним. Мы передаем путь как первый параметр для open(); это - только строка, содержащая путь к файлу на диске, что-то вроде shoe.jpg. Второй параметр *open()* говорит что мы открываем файл для чтения и что файл содержит двоичные данные. На некоторых операционных системах Вы можете безопасно не указывать спецификатор и файл загрузится должным образом, но на одной системе (а именно, Windows) я столкнулся с проблемами, когда я опустил его. Для своей безопасности, всегда используйте и r и b, когда имеете дело с файлами изображений.

В блоке мы читаем содержимое файла и создаем экземпляр **FXJPGImage**. **FXJPGImage** - подкласс **FXImage** знает, как вывести на экран изображение JPEG.

Так, наш метод *load_image()* открывает именованный файл JPEG, читает его содержимое, и создает объект **FXJPGImage** соответствующий нашей фотографии. Очевидно, что если путь обращается к GIF или некоторому другому типу файла изображения, это перестанет работать. Чтобы не усложнять, наша программа собирается ограничиться непосредственно отображением изображений JPEG, но для некоторых идей, как обеспечить поддержку других типов изображений, см. **Главу 11**, «**Создание Визуально Богатых Пользовательских Интерфейсов**», на странице 142.

Осторожный читатель, можете задаваться вопросом о параметре **app**, который должен быть первым аргументом **FXJPGImage.new()**. В другом месте в нашей программе мы создаем объект **FXApp** и передаём его в метод **PictureBook.new()**, когда мы создаем главное окно, но как этот метод экземпляра спускается в низ в класс **PhotoView**?

Оказывается, что каждый класс, представляющий виджет **FXRuby**, наследует метод экземпляра **app()**, который возвращает ссылку на объект приложения. По историческим причинам все еще необходимо передать ссылку к объекту приложения в некоторых методах (таких как **FXJPGImage.new()**), даже при том, что практически Вы можете создать только один экземпляр **FXApp** для приложения **FXRuby**.

Теперь рассмотрите более внимательно среднюю часть *load_image()*, где мы фактически создаём объект **FXJPGImage**. Мы присваиваем недавно созданный объект к *self.image*. Что делает **self** в этом контексте, предполагая что *load_image()* является методом экземпляра для класса **PhotoView**? Правильно, **self** является только ссылкой на объект **PhotoView**. Теперь, наш класс **PhotoView** не определяет атрибут с именем *image*, но его базовый класс определяет. Таким образом, это - как мы говорим, представление фотографии, которая должна отобразится на экране.

Прежде, чем мы забудем это, давайте добавим вызов *load_image()* из метода *initialize()* **PhotoView**. Ваш класс **PhotoView** должен теперь быть похожим на это:

Http://media.pragprog.com/titles/fxruby/code/picturebook_a/photo_view.rb

```
class PhotoView < FXImageFrame
	def initialize(p, photo)
		super(p, nil)
		load_image(photo.path)
	end

	def load_image(path)
		File.open(path, "rb" ) do |io|
			self.image = FXJPGImage.new(app, io.read)
		end
	end
end
```

Наконец, мы в состоянии увидеть что-то действительно интересное. Запустите программу и смотрите то, что происходит:

```
$ ruby picturebook.rb
```

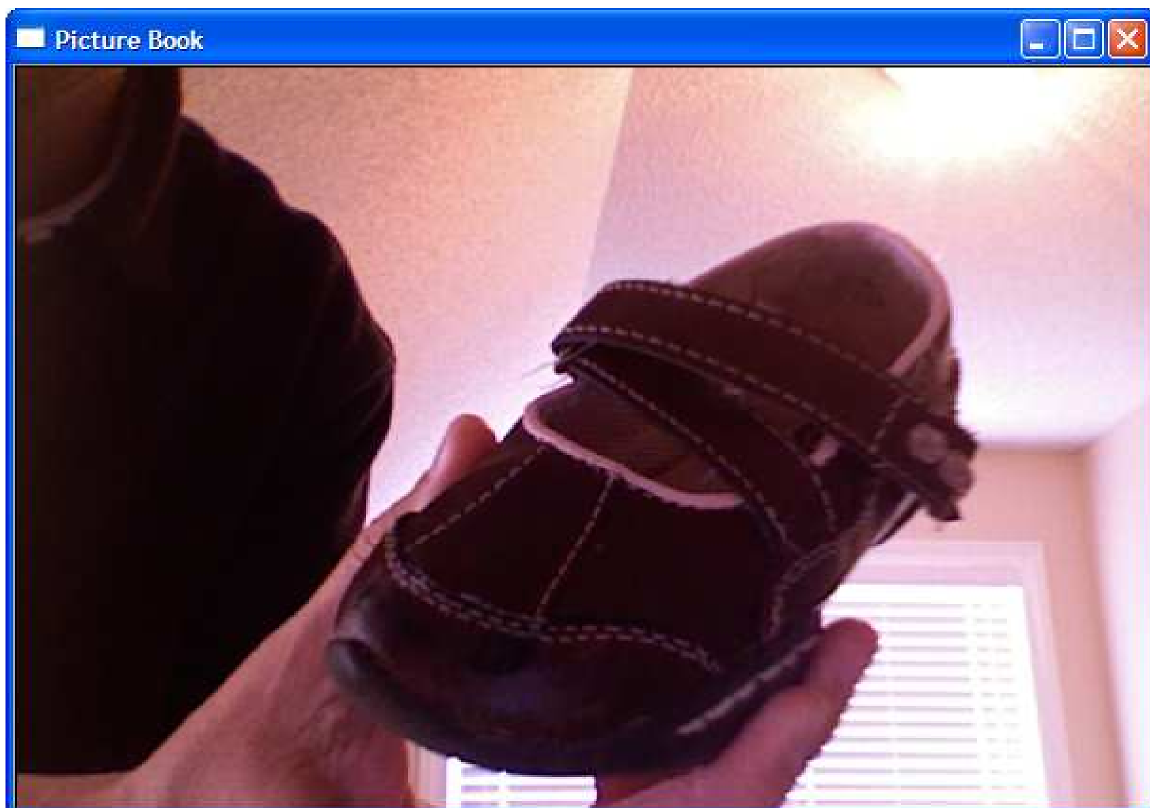



Figure 4.1: Picture Book, displaying a single image

Рисунок 4.1 показывает как это выглядит под Windows. Помните, что если Ваша фотография является слишком большой, чтобы поместиться в окне, Вы увидите только верхнюю левую часть её. Не волнуйтесь, мы собираемся это скоро исправить!

Мы достигли значительного прогресса в этой главе. Picture Book отображает фотографию, и таким образом, с минимумом кода, мы уже имеем базовую функциональность приложения. Мы видим, что встроенные виджеты, такие как **FXImageFrame** могут быть разделены на подклассы как любой другой класс Ruby, чтобы обеспечить пользовательское поведение. Мы также изучили, как использовать класс **FXJPGImage**, чтобы создать представление в оперативной памяти фотографии JPEG на диске, в одной строке кода. Есть еще много работы, потому что мы в состоянии вывести на экран только одну фотографию, и мы используем непосредственно прописанный путь к этой фотографии. В следующей главе мы предпримем некоторых шаги, чтобы изменить к лучшему эту ситуацию.

# Глава 5

# Take 2: Display an Entire Album

Возможно у Вас есть только эта особенная фотография которую Вы желаете рассматривать. Если так, поздравляем! Вы закончили и может идти дальше к разработке некоторого другого приложения **FXRuby**. Если это не так, у Вас, вероятно, есть крупная библиотека фотографий. Следовательно мы должны обновить приложение так, чтобы мы могли вывести на экран весь альбом фотографий.

Мы замедлили наш путь в разработку **FXRuby** в предыдущей главе, но теперь пора набрать темп. Мы собираемся рассмотреть многое в этой главе и освоить многие вопросы **FXRuby**. Например, добиться понимания того, как работают менеджеры расположения в построении пользовательского интерфейса, если Вы хотите разрабатывать что-либо кроме тривиальных пользовательских интерфейсов с **FXRuby**. Вы получите введение в менеджеры расположения, когда мы изучим, как использовать **FXMatrix** и **FXScrollWindow**, чтобы заняться некоторыми проблемами расположения в нашем приложении. В предыдущей главе мы видели как легко мы могли создать объект изображения из файла и затем вывести его на экран. В этой главе мы научимся немного большему по манипулированию изображениями в **FOX**, когда мы создадим миниатюры фотографий альбома. Мы также собираемся изучить, как добавить к нашему приложению меню с выпадающими подменю и как реализовать действия, связанные с командами меню. К тому времени, когда Вы завершите это обновление Picture Book, Вы будет намного лучшее понимать, как создавать серьёзные приложения в **FXRuby**.

## 5.1 Add Album View

**FOX** предоставляет несколько специальных виджетов, известных как менеджеры расположения (компоновщики). Цель менеджера по расположению состоит в том, чтобы автоматически определить размеры и размещение его дочерних окон, согласно некоторой политике расположения, которая уникальна для этого компоновщика. Мы обсудим некоторые из них более подробно в Главе 12, «**Управление Расположением**», на странице 159. В этом разделе мы получим введение в менеджер расположения **FXMatrix**.

Чтобы вывести на экран все фотографии в альбоме, мы нуждаемся в некотором классе представления который способен управлять многими экземплярами PhotoView. Есть много путей сделать это. Для этого примера мы будем использовать менеджер расположения **FXMatrix**, который размечает дочерние окна в строках и столбцах. Так, наш класс AlbumView получен из **FXMatrix**:

Http://media.pragprog.com/titles/fxruby/code/picturebook_b/album_view.rb

```
class AlbumView < FXMatrix
    attr_reader :album
    def initialize(p, album)
        super(p, :opts => LAYOUT_FILL)
        @album = album
    end
end
```

Первым параметром для *initialize()*, является родительский виджет для представления альбома, и второй параметр - ссылка на объект Album. Как мы изучали в предыдущей главе, мы должны убедиться, что вызвали метод *initialize()* базового класса, когда мы разделяем виджет на подклассы в **FXRuby**. Если посмотреть документацию для класса **FXMatrix** (http://www.fxruby.org/doc/api/classes/Fox/FXMatrix.html), единственный необходимый параметр для метода *initialize()* базового класса – родительский виджет, таким образом,

мы должны убедиться, что передали этот параметр в вызов *super()*. Мы также укажем расположение в **LAYOUT_FILL**, которое говорит матрице быть жадной и занимать столько места, сколько возможно.

Затем, мы хотим выполнить итерации по всем фотографиям в альбоме и добавить их к представлению. Добавьте следующую строку кода в конец метода *initialize()* для класса AlbumView:

Http://media.pragprog.com/titles/fxruby/code/picturebook_a/album_view.rb

```
@album.each_photo { |photo| add_photo(photo) }
```

Метод add_photo() для класса AlbumView похож на это:

```
def add_photo(photo)
	PhotoView.new(self, photo)
end
```

Отметьте, что, когда мы создаем объект PhotoView, мы снова передаем в **self** как первый параметр PhotoView.new. На сей раз, тем не менее, **self** не ссылается на главное окно, не так ли? Нет, теперь мы создаем photo views как дочерние элементы окна AlbumView.

Говоря об этом, мы должны изменить метод *initialize()* для PictureBook так, чтобы это создало Album и AlbumView вместо PhotoView:

Http://media.pragprog.com/titles/fxruby/code/picturebook_b/picturebook.rb

```
def initialize(app)
	super(app, "Picture Book" , :width => 600, :height => 400)
	@album = Album.new("My Photos" )
	@album.add_photo(Photo.new("shoe.jpg" ))
	@album.add_photo(Photo.new("oscar.jpg" ))
	@album_view = AlbumView.new(self, @album)
end
```

Не забудьте добавить, необходимые операторы **require** в album_view.rb и picturebook.rb, чтобы определения классов Album, AlbumView, Photo и PhotoView были доступны. Ваша копия album_view.rb должна быть такой:

Http://media.pragprog.com/titles/fxruby/code/picturebook_b/album_view.rb

```
require 'photo_view'

class AlbumView < FXMatrix
	attr_reader :album
	def initialize(p, album)
		super(p, :opts => LAYOUT_FILL)
		@album = album
		@album.each_photo { |photo| add_photo(photo) }
	end

	def add_photo(photo)
		PhotoView.new(self, photo)
	end
end
```

И вот обновленная версия picturebook.rb:

Http://media.pragprog.com/titles/fxruby/code/picturebook_b/picturebook.rb

```
require 'fox16'
include Fox

require 'album'
require 'album_view'
require 'photo'

class PictureBook < FXMainWindow
	def initialize(app)
		super(app, "Picture Book" , :width => 600, :height => 400)
		@album = Album.new("My Photos" )
		@album.add_photo(Photo.new("shoe.jpg" ))
		@album.add_photo(Photo.new("oscar.jpg" ))
		@album_view = AlbumView.new(self, @album)
	end

	def create
		super
		show(PLACEMENT_SCREEN)
	end
end

if __FILE__ == $0
	FXApp.new do |app|
		PictureBook.new(app)
		app.create
		app.run
	end
end
```

Выполните программу:

**$ ruby picturebook.rb**

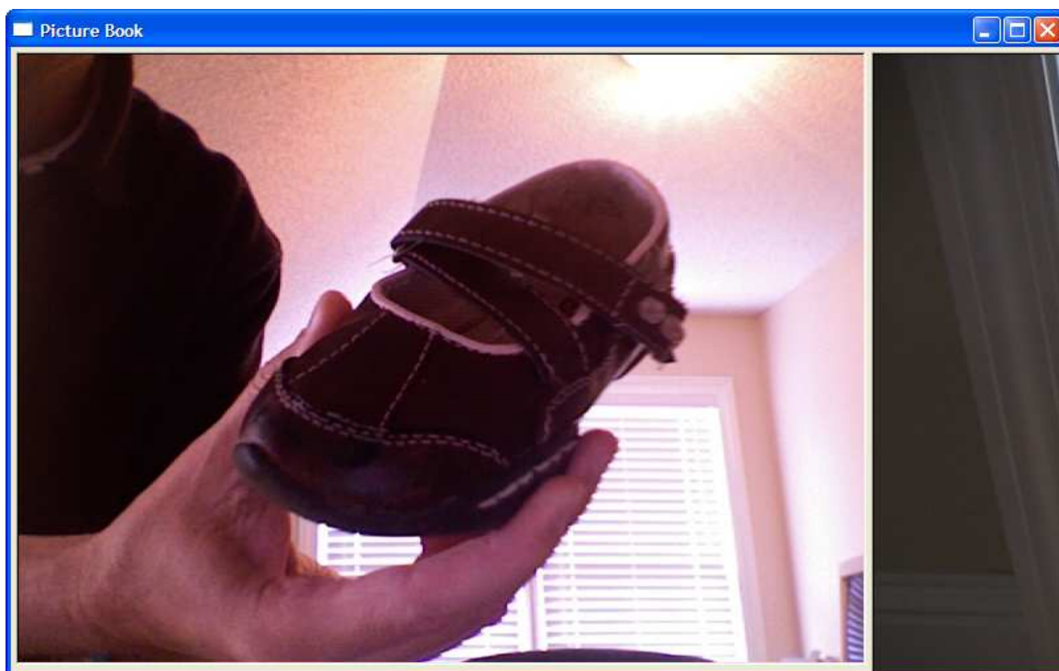



Figure 5.1: Picture Book, displaying an album

Рисунок 5.1, показывает то, что это выполняется на моей машине, но похоже, что есть проблема. В зависимости от размеров фотографий, которые Вы пытаетесь вывести на экран, Вы можете видеть это также. Так как общий размер альбома

увеличился, мы исчерпали пространство для вывода на экран фотографий, и некоторые из них частично (или полностью) отсечены. Изменение размеров окна вручную позволит Вам увидеть немного больше чем по умолчанию, но это, очевидно, не выход. Мы собираемся произвести несколько изменений, чтобы решить эту проблему, и для начала необходимо немного уменьшить масштаб изображений.

## 5.2 Display Images as Thumbnails

В дополнение к простому отображению изображений **FOX** поддерживает множество различных эффектов манипулирования изображением. В этом разделе мы будем изучать, как использовать метод *scale()* от API **FXImage**, чтобы уменьшить размер наших импортированных фотографий.

Класс **FXJPGImage**, который мы используем, чтобы представить изображения JPEG, является подклассом **FXImage**, и **FXImage** обеспечивает много действительно полезных APIs для того, чтобы управлять изображениями. Чтобы решить эту проблему, мы будем использовать метод *scale()*, чтобы уменьшить изображения от естественного размера так, чтобы это было более удобно в представлении альбома. Так как класс PhotoView ответственен за отображение фотографий, все изменения для этой итерации будут изолированы в этом классе.

Мы хотим, чтобы получающееся изображение входило в ограничивающий прямоугольник, при поддержке его исходного формата изображения. В настоящее время давайте принимать то, что размерности ограничивающего прямоугольника фиксированы и определены константами класса **MAX_WIDTH** и **MAX_HEIGHT**:

Http://media.pragprog.com/titles/fxruby/code/picturebook_b/photo_view.rb

```
MAX_WIDTH = 200
MAX_HEIGHT = 200
```

Уменьшенная ширина миниатюры изображения будет меньше исходной ширины или **MAX_WIDTH**. Точно так же уменьшенная высота миниатюры будет меньше своей исходной высоты или **MAX_HEIGHT**. Позвольте нам добавить некоторые методы, чтобы вычислить масштабируемую ширину и высоту миниатюр:

http://media.pragprog.com/titles/fxruby/code/picturebook_b/photo_view.rb

```
def scaled_width(width)
	[width, MAX_WIDTH].min
end

def scaled_height(height)
	[height, MAX_HEIGHT].min
end
```

Теперь мы можем записать код, который фактически выполняет масштабирование. Давайте выполним это как *scale_to_thumbnail()*:

http://media.pragprog.com/titles/fxruby/code/picturebook_b/photo_view.rb

```
def scale_to_thumbnail
	aspect_ratio = image.width.to_f/image.height
	if image.width > image.height
		image.scale(
			scaled_width(image.width),
			scaled_width(image.width)/aspect_ratio,
			1
		)
```

```
		else
			image.scale(
				aspect_ratio*scaled_height(image.height),
				scaled_height(image.height),
				1
			)
		end
	end
```

Формат изображения это просто отношение ширины изображения к его высоте, но мы должны рассмотреть два случая. Если изображение более широко, чем это высоко, тогда мы хотим уменьшить ширину изображения так, чтобы оно соответствовало ограничивающему прямоугольнику и затем корректирует высоту соответственно. В другом случае, если изображение более высоко чем широко, высота этого изображения является важной размерностью.

Наконец, мы можем добавить вызов нашего нового метода *scale_to_thumbnail()* в конец *load_image()*. Так, чтобы Вы могли видеть все эти изменения в контексте, вот полный список для нашей новой и улучшенной версии PhotoView:

http://media.pragprog.com/titles/fxruby/code/picturebook_b/photo_view.rb

```
class PhotoView < FXImageFrame

	MAX_WIDTH = 200
	MAX_HEIGHT = 200

	def initialize(p, photo)
		super(p, nil)
		load_image(photo.path)
	end

	def load_image(path)
		File.open(path, "rb" ) do |io|
			self.image = FXJPGImage.new(app, io.read)
			scale_to_thumbnail
		end
	end

	def scaled_width(width)
		[width, MAX_WIDTH].min
	end

	def scaled_height(height)
		[height, MAX_HEIGHT].min
	end

	def scale_to_thumbnail
		aspect_ratio = image.width.to_f/image.height
		if image.width > image.height
			image.scale(
				scaled_width(image.width),
				scaled_width(image.width)/aspect_ratio,
				1
			)
		else
			image.scale(
				aspect_ratio*scaled_height(image.height),
				scaled_height(image.height),
				1
			)
		end
```

```
    end
end
```

Если Вы снова запустите программу, Вы должны увидеть обе фотографии, приблизительно одинакового размера (рисунок 5.2). Это выглядит несколько лучше, хотя это позор что у нас есть только две фотографии. Мы знаем, конечно, что могли бы программно добавить даже больше фотографий, но это не идеальное решение. Мы должны дать пользователю некоторые средства выбора файлов JPEG с диска для создания альбома в интерактивном режиме. Давайте добавим эту функциональность.

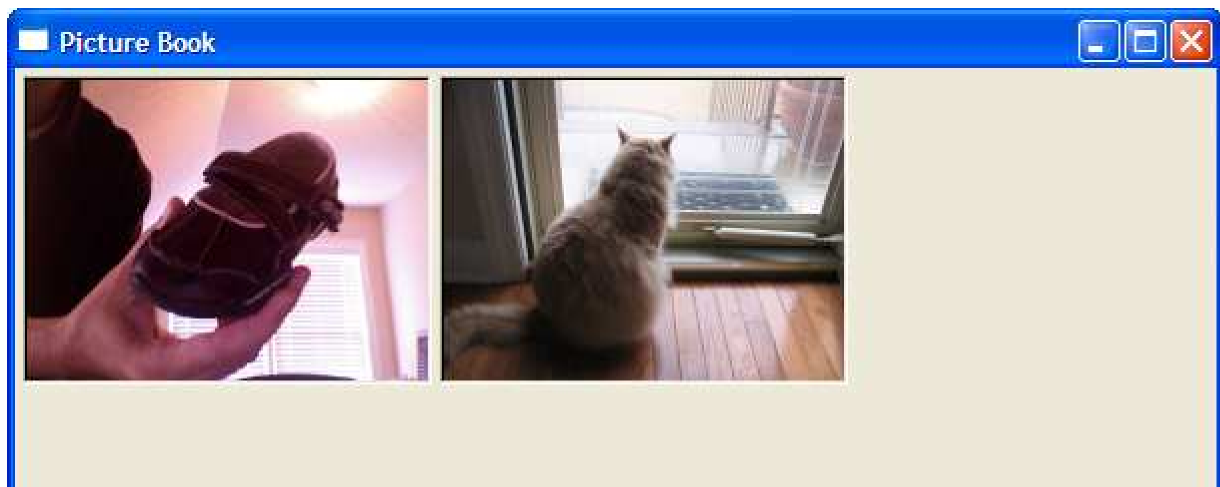



Figure 5.2: Displaying image as thumbnails

## 5.3 Import Photos from Files

До сих пор мы вручную создавали Альбом и добавляли объекты фотографий в него. Это, очевидно, не то что нам нужно; мы нуждаемся в диалоговом окне выбора файла для выбора одной или более фотографий и создания альбома из этого списка. В этой итерации мы будем учиться как классы **FXMenuBar**, **FXMenuPane**, **FXMenuTitle** и **FXMenuCommand** могут быть использованы для оснащения приложения панелью меню с выпадающими подменю. Мы увидим как использовать метод *connect()* для соединения виджетов, таких как кнопки **FXMenuCommand** с блоками кода **Ruby**. Наконец, чтобы предоставить пользователю средство выбрать файлы для импортирования в альбом, мы выведем на экран диалоговое окно **FXFileDialog** и затем получим имена выбранных файлов из него.

Когда пользователи начинают искать команду, чтобы импортировать фотографии, первое место где они будут искать, будет меню *File*. Давайте добавим его. Сохраним метод *initialize()* для класса PictureBook чистым насколько возможно, поэтому давайте поместим весь код, имеющий отношение к построению строки меню в новом методе экземпляра с названием *add_menu_bar()*. Первая вещь, которую должен сделать этот метод, создание экземпляра **FXMenuBar** как дочернего элемента главного окна:

http://media.pragprog.com/titles/fxruby/code/picturebook_c/picturebook.rb

```
def add_menu_bar
    menu_bar = FXMenuBar.new(self, LAYOUT_SIDE_TOP|LAYOUT_FILL_X)
end
```

Так как Вы - старый профессионал если добрались до этой темы, значит Вы уже смотрели документацию API для класса **FXMenuBar** и заметили, что есть фактически две перегрузки для метода *initialize()*. Мы будем использовать версию, которая создает "nonfloatable" menu bar, и она имеет два необходимых параметра: родительское окно и опции. Как старый профессионал, Вы также уже знаете, что **self,** куда мы размещаем menu bar, соотносится с окном PictureBook, так как это - метод экземпляра для класса PictureBook. **LAYOUT_SIDE_TOP** и **LAYOUT_FILL_X** говорят **FXRuby** помещать строку меню наверху главного окна и простираться настолько широко, насколько возможно.

Затем, мы создаем окно **FXMenuPane** как дочерний элемент **FXMenuBar**:

http://media.pragprog.com/titles/fxruby/code/picturebook_c/picturebook.rb

```
file_menu = FXMenuPane.new(self)
```

Область меню будет содержать все команды для меню *File*. Эта область - своего рода всплывающее окно, что означает, что оно появляется на короткое время. Оно появляется при вызове. Вы взаимодействуете с ним, выбирая команду меню, и затем оно исчезает. Вы вызываете область меню, щелкая по виджету **FXMenuTitle**, связанному с той областью меню.

http://media.pragprog.com/titles/fxruby/code/picturebook_c/picturebook.rb

```
FXMenuTitle.new(menu_bar, "File" , :popupMenu => file_menu)
```

Этот немного хитро. **FXMenuTitle** - дочерний элемент **FXMenuBar**, но он должен знать, какую область меню надо вывести на экран при активации. Таким образом, мы передаем это как параметр *:popupMenu*. Теперь мы имеем menu bar, так же как меню *File*, таким образом, пора добавить нашу первую команду:

http://media.pragprog.com/titles/fxruby/code/picturebook_c/picturebook.rb

```
import_cmd = FXMenuCommand.new(file_menu, "Import..." )
import_cmd.connect(SEL_COMMAND) do
	# ...
end
```

Мы создаем объект **FXMenuCommand** как дочерний элемент области меню. Вызывая *connect()* из import_cmd, мы связываем блок кода **Ruby** с этой командой. Когда пользователь выбирает команду *Import...* из меню *File*, мы хотим вывести на экран диалоговое окно выбора файла. Для того, чтобы сделать это, добавим блок *connect()*:

http://media.pragprog.com/titles/fxruby/code/picturebook_c/picturebook.rb

```
dialog = FXFileDialog.new(self, "Import Photos" )
dialog.selectMode = SELECTFILE_MULTIPLE
dialog.patternList = ["JPEG Images (*.jpg, *.jpeg)" ]
if dialog.execute != 0
	import_photos(dialog.filenames)
end
```

Мы начинаем, создавая **FXFileDialog** как дочерний элемент главного окна, с заголовком Import Photos. Затем, мы устанавливаем режим выбора файла для этого диалогового окна в **SELECTFILE_MULTIPLE**, что означает разрешение выбрать любое число существующих файлов для импорта. Мы также устанавливаем список для диалогового окна так, чтобы на экран выводились только имена файлов имеющие расширение .jpg или .jpeg, так как они - единственные файлы которыми мы интересуемся. Наконец, мы вызываем *execute()*, чтобы вывести на экран диалоговое окно и ожидаем чтобы пользователь выбрал некоторые файлы.

Метод **execute()** для диалогового окна возвращает код завершения 0 или 1, в зависимости от того, щелкнул ли пользователь *Cancel*, чтобы закрыть диалоговое окно или *OK*, чтобы принять выбранные файлы. Если пользователь нажал *Cancel*, мы не должны ничего больше делать. Иначе, мы должны вызвать пока еще несуществующий метод *import_photos()* чтобы импортировать выбранные фотографии в наш альбом. Давайте добавим тот метод в класс **PictureBook**:

http://media.pragprog.com/titles/fxruby/code/picturebook_c/picturebook.rb

```
def import_photos(filenames)
	filenames.each do |filename|
		photo = Photo.new(filename)
		@album.add_photo(photo)
		@album_view.add_photo(photo)
	end
	@album_view.create
end
```

Метод *import_photos()* выполняет итерации по именам файлов, собранным **FXFileDialog** и добавляет новую фотографию к AlbumView для каждой из их. Отметьте, что теперь импорт фотографий в альбом стал намного удобнее и Вы можете удалить неудобные вызовы *add_photo()*, что у нас были в методе *initialize()*.

Вы, возможно, заметили вызов метода *create()* в заключительной части *import_photos()*. Я надеюсь Вы заметили, потому что, если Вы пропустите его, Вы не увидите свои недавно импортированные фотографии.

Мы вызывали метод *create()* прежде, в первой главе когда мы создавали наш “Привет, Мир!”. Тогда, мы отмечали, что вызов *create()* гарантирует что все серверные ресурсы для приложения будут созданы, и пока мы оставим всё в этом виде. Мы собираемся говорить об этой теме очень подробно позже, в Разделе 7.7, на странице 95.

Но вернёмся к нашей задаче. Пока мы здесь, почему бы не добавить команду *Exit* к меню *File*? Добавьте эти строки к методу *add_menu_bar()*, прямо после кода, который устанавливает команду *Import...*:

http://media.pragprog.com/titles/fxruby/code/picturebook_c/picturebook.rb

```
exit_cmd = FXMenuCommand.new(file_menu, "Exit" )
exit_cmd.connect(SEL_COMMAND) do
	exit
end
```

Чтобы завершить эту итерацию, мы должны добавить вызов *add_menu_bar()* из метода *initialize()*. Ваш класс PictureBook должен быть похожим на это:

http://media.pragprog.com/titles/fxruby/code/picturebook_c/picturebook.rb

```
require 'fox16'
include Fox

require 'album'
require 'album_view'
require 'photo'

class PictureBook < FXMainWindow
	def initialize(app)
		super(app, "Picture Book" , :width => 600, :height => 400)
		add_menu_bar
		@album = Album.new("My Photos" )
		@album_view = AlbumView.new(self, @album)
	end

	def add_menu_bar
		menu_bar = FXMenuBar.new(self, LAYOUT_SIDE_TOP|LAYOUT_FILL_X)
		file_menu = FXMenuPane.new(self)
		FXMenuTitle.new(menu_bar, "File" , :popupMenu => file_menu)
		import_cmd = FXMenuCommand.new(file_menu, "Import..." )
		import_cmd.connect(SEL_COMMAND) do
			dialog = FXFileDialog.new(self, "Import Photos" )
			dialog.selectMode = SELECTFILE_MULTIPLE
			dialog.patternList = ["JPEG Images (*.jpg, *.jpeg)" ]
			if dialog.execute != 0
				import_photos(dialog.filenames)
			end
		end
		exit_cmd = FXMenuCommand.new(file_menu, "Exit" )
		exit_cmd.connect(SEL_COMMAND) do
			exit
		end
	end

	def import_photos(filenames)
		filenames.each do |filename|
			photo = Photo.new(filename)
			@album.add_photo(photo)
			@album_view.add_photo(photo)
		end
		@album_view.create
	end

	def create
		super
		show(PLACEMENT_SCREEN)
	end
end

if __FILE__ == $0
	FXApp.new do |app|
		PictureBook.new(app)
		app.create
		app.run
	end
end
```

Выполните программу. Представление альбома должно быть пустым, когда программа запускается, но Вы должны быть в состоянии использовать *Импорт...* из меню *File* для выбора некоторых фотографий и добавления их к альбому. Рисунок 5.3, на следующей странице, показывает на что это похоже в Windows, после того, как я импортировал несколько фотографий в свой альбом.

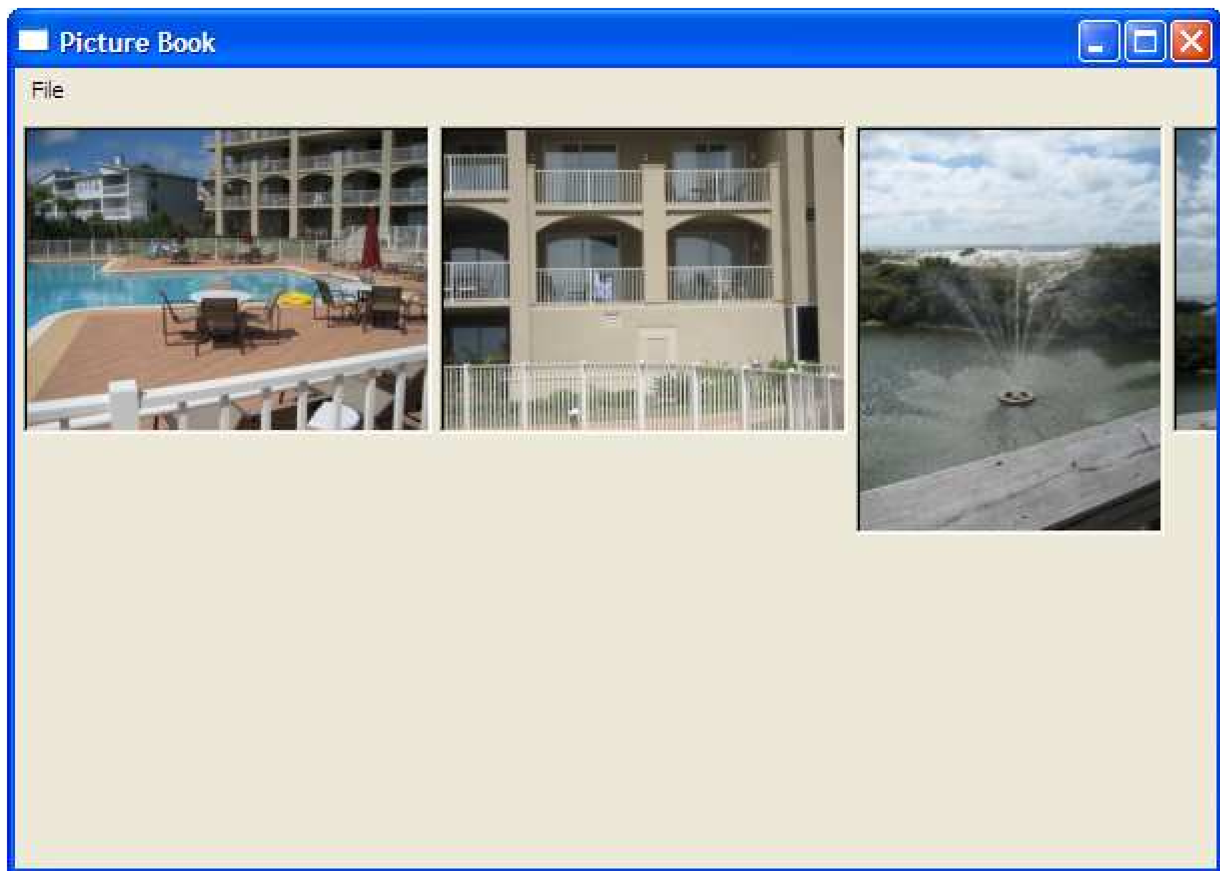



Figure 5.3: Now with menus

Если Вы внимательно изучите самый правый край окна, то увидите, что последняя фотография отсечена. А это даже не последняя фотография которую я импортировал; похоже другие фото не влезают в экран! Это похоже на то, что наше окно переполнено. Мы должны произвести небольшое изменение в класс AlbumView, чтобы поработать с этой проблемой.

## 5.4 Dynamically Reconfigure the Album View

Конфигурация по умолчанию для менеджера расположения **FXMatrix**, который мы используем, это просто помещать все фотографии в единственную строку. Мы предпочли бы, чтобы он поместил так много фотографий, сколько влезет в строку и затем использовал дополнительные строки чтобы вывести на экран оставшиеся фотографии. Чтобы сделать это, мы должны произвести несколько изменений.

Первое изменение, которое мы должны произвести, имеет отношение к алгоритму компоновки который использует менеджер размещения **FXMatrix**. Матрица может быть сконфигурирована размечать дочерние элементы с любым постоянным числом строк (значение по умолчанию) или с постоянным числом столбцов. Так как мы хотим фиксировать число столбцов и позволить числу строк изменяться, мы должны передать опцию **MATRIX_BY_COLUMNS** к списку опций для представление альбома. Наша измененная версия *initialize()* для класса AlbumView похожа на это:

http://media.pragprog.com/titles/fxruby/code/picturebook_d/album_view.rb

```
def initialize(p, album)
	super(p, :opts => LAYOUT_FILL|MATRIX_BY_COLUMNS)
	@album = album
	@album.each_photo { |photo| add_photo(photo) }
end
```

Затем, мы должны определить, сколько столбцов матрица должна вывести на экран.

Проблема состоит в том, что число зависит от того, сколько пространства нам надо и сколько столбцов фотографий мы можем сделать. Например, если окно альбома было 800 пикселей и фотографии были каждая 200 пикселей в ширину, мы могли бы разместить приблизительно четыре фотографии в каждой строке. Однако, если окно было изменено так что оно стало более узким или более широким, мы должны были бы всё пересчитать.

Учитывая факт, что требуемое число столбцов зависит от текущей ширины альбома мы определим метод представление альбома *layout()*. Всякий раз, когда размер окна изменяется FOX гарантирует, что метод *layout()* будет вызван, чтобы он мог обновить позиции и размеры его дочерних окон (подробно это рассмотрено на стр.159, в главе 12). Мы собираемся использовать это, чтобы повторно вычислить число столбцов для матрицы прежде, чем будет выполнено расположение. Добавьте следующий метод к классу AlbumView:

http://media.pragprog.com/titles/fxruby/code/picturebook_d/album_view.rb

```
def layout
	self.numColumns = [width/PhotoView::MAX_WIDTH, 1].max
	super
end
```

Вторая строка нашей переопределенной версии *layout()* использует **super**, чтобы вызвать версию базового класса *layout()*. Как обычно это - шаг, который мы не хотим пропустить. Но давайте сосредоточимся на первой строке, где мы фактически присваиваем число столбцов.

Начиная с выражения на правой стороне присвоения, width относится к ширине окна представления альбома в пикселях. Это можно рассматривать будто мы обращаемся к переменной, но мы фактически обращаемся к методу класса **FXMatrix**, который возвращает ширину окна. Если Вы ищете метод *width()* в документации по API, Вы найдете, что это фактически определено в классе **FXWindow**, от которого получен **FXMatrix** и многие другие классы **FXRuby**.

**PhotoView::MAX_WIDTH** ссылается на возможный максимум ширины фотографии. Помните, дочерние окна для представления альбома это только экземпляры PhotoView. Мы делим полную ширину на максимально возможную ширину фотографии и присваиваем результат атрибуту матрицы **numColumns** (Снова, хотя я именую **numColumns** как атрибут, мы действительно только вызываем метод экземпляра **FXMatrix**, названный **numColumns = ()**).

Наконец, если пользователь уменьшает окно альбома так, что оно становится фактически более узким чем **PhotoView::MAX_WIDTH**, мы нуждаемся к защитите от обнуления числа столбцов. Мы создаём массив с двумя элементами, содержащим номер 1 и наше вычисление для **numColumns**, мы можем тогда вызвать метод массива *max()* и возвратить большее из двух значений. Это гарантирует, что мы получим по крайней мере один столбец.

Давайте посмотрим, окупилась ли наша работа. Если Вы запустите приложение и импортируете набор фотографий, Вы должны теперь увидеть что когда строка в альбоме заполняется целиком, добавляется новая строка. Рисунок 5.4, на следующей странице, показывает это. Вы можете также изменить размеры главного окна, чтобы сделать его более узким или более широким, и видите, что число столбцов в альбоме изменяется динамически в зависимости от того, сколько пространства доступно.

К сожалению, это изменение решает только часть нашей проблемы. Прежде, чем мы сделали это изменение, фотографии, которые не влезали в окно альбома, были обрезаны с правого края окна. Теперь, если Вы импортируете слишком много фотографий, Вы увидите, что некоторые не выровнены по нижней части окна. Чтобы рассмотреть эту проблему, мы должны рассмотреть еще один компоновщик.

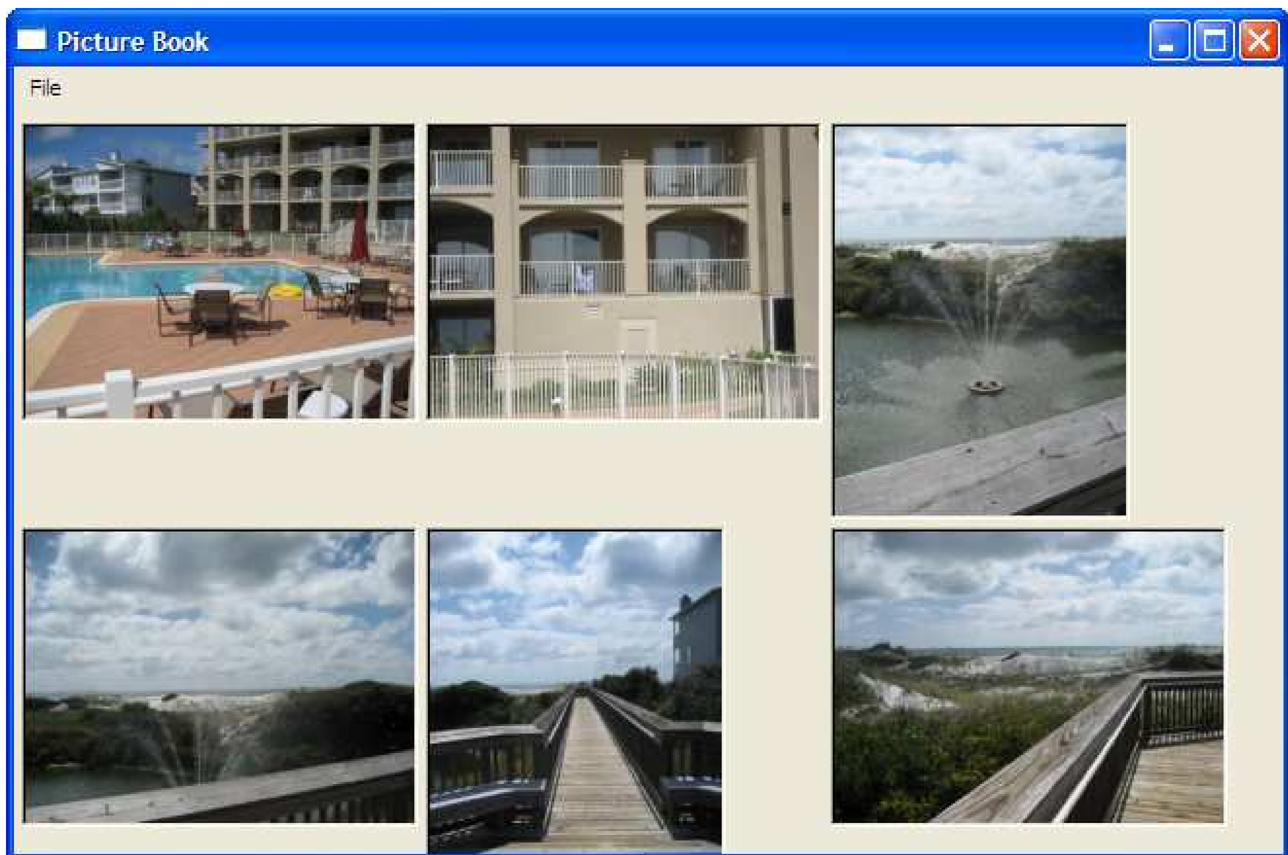



Figure 5.4: Making the album resizable

## 5.5 Make the Album View Scrollable

Мы получили введение по компоновщикам **FOX**, когда мы решили использовать менеджера расположения **FXMatrix** в качестве основания для нашего класса AlbumView. В этом разделе мы рассмотрим другой компоновщик, **FXScrollWindow**, и видим, как использовать его, чтобы решить нашу последнюю дилемму расположения.

Иногда, содержимое, которое Вы хотите вывести на экран в окне, просто слишком большое, чтобы соответствовать в просматриваемой области этого окна. Например, много цифровых фотоаппаратов теперь способны сделать фотографии такого высокого разрешения, что они не могут быть выведены на экран в их истинных размерностях на типичном компьютерном мониторе. Когда это имеет место, Вы можете использовать несколько опций. Одна опция должна уменьшить размерность содержимого так, чтобы оно соответствовало в окну. Однако надо помнить, что уменьшение ухудшит качество изображения, другая опция должна поместить содержимое в окно прокрутки.

**FOX** предусматривает эту последнюю опцию посредством классов **FXScrollArea** и **FXScrollWindow**. **FXScrollWindow** - подкласс **FXScrollArea**, и в большинстве случаев это - класс, который Вы будете использовать, когда будете создавать окна контента с возможностью прокрутки в Ваших приложениях. Мы собираемся применить эту технологию изменяя наш класс AlbumView, чтобы сделать его подклассом **FXScrollWindow**:

http://media.pragprog.com/titles/fxruby/code/picturebook_e/album_view.rb
```
class AlbumView < FXScrollWindow
      # ...
end
```
Следующий набор изменений, которые мы должны произвести, включает метод *initialize()* для AlbumView. Быстрая проверка документации API для **FXScrollWindow**

(http://www.fxruby.org/doc/api/classes/Fox/FXScrollWindow.html) показывает, что метод *initialize()*, имеет только один требуемый параметр, и как это имело место для *initialize()* метода **FXMatrix**, этот параметр - родительское окно. Мы собираемся произвести одно небольшое изменение к тому, как мы вызываем метод *initialize()* базового класса. Кроме этого, к родительскому окну мы собираемся передать подсказку расположения **LAYOUT_FILL** как одну из наших опций конструкции:

http://media.pragprog.com/titles/fxruby/code/picturebook_e/album_view.rb

```
def initialize(p, album)
	super(p, :opts => LAYOUT_FILL)
	# ...
end
```

Вы изучите все о менеджерах расположения и подсказках расположения в Главе 12, «*Managing Layouts*», на странице 159. Пока, достаточно знать что это подсказка расположения говорит **FXRuby**, что мы хотели бы, чтобы представление альбома простиралось в горизонтальном и вертикальном направлении.

Теперь, когда мы инициализировали часть базового класса представления альбома, можно сказать, что мы получили скроллинг. У **FXScrollWindow** есть единственное дочернее окно, которое определяется как окно его содержимого. Для нашего приложения экземпляр **FXMatrix** - это контент, который мы хотим прокручивать. Так, представление альбома - теперь **FXScrollWindow**, которое содержит **FXMatrix**, который содержит наборы экземпляров PhotoView:

http://media.pragprog.com/titles/fxruby/code/picturebook_e/album_view.rb

```
def initialize(p, album)
	super(p, :opts => LAYOUT_FILL)
	@album = album
	FXMatrix.new(self, :opts => LAYOUT_FILL|MATRIX_BY_COLUMNS)
	@album.each_photo { |photo| add_photo(photo) }
end
```

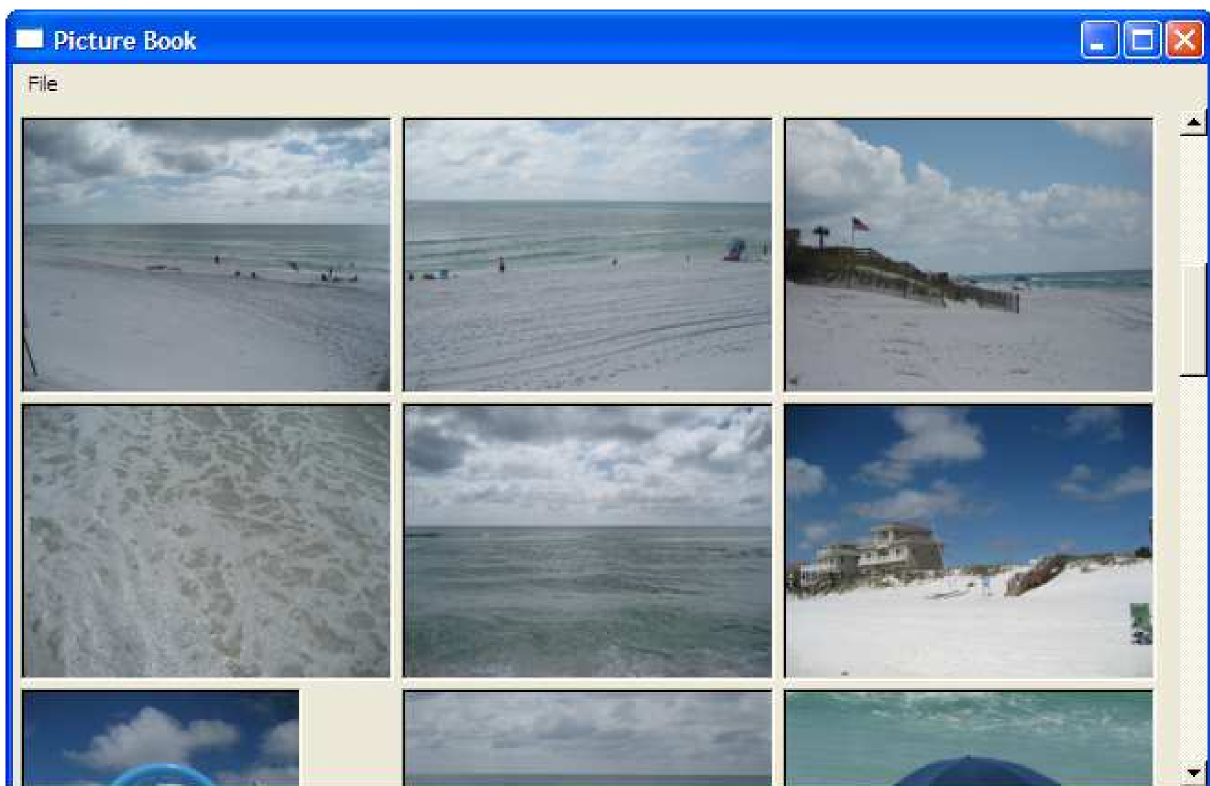



Figure 5.5: Making the album scrollable

Бросьте взгляд на это, и удостоверьтесь, что Вы понимаете то, что делаете прежде чем читать дальше. Что означает, например, третья строка, когда мы передаем **self** как параметр для **FXMatrix.new**? На что ссылается **self** в этом контексте? Вы начинаете понимать что это средство для окна, подобного окну **FXMatrix**, чтобы стать дочерним окном другого окна - **FXScrollWindow**?

Финальные изменения к классу AlbumView для этой итерации имеют место в методах *add_photo()* и *layout()*. Вот на что новая и улучшенная версия *add_photo()* похожа:

http://media.pragprog.com/titles/fxruby/code/picturebook_e/album_view.rb

```
def add_photo(photo)
	PhotoView.new(contentWindow, photo)
end
```

Различие между этой версией *add_photo()* и предыдущей не слишком заметны. Вместо того, чтобы передать **self** в **PhotoView.new**, мы передаем в результате метод *contentWindow()* окна прокрутки. Как говорилось ранее, окном контента для **FXScrollWindow** является только первое (и только) дочернее окно, которое Вы добавляете к нему – в нашем случае виджет **FXMatrix**. Так, мы все еще создаем наши объекты PhotoView как дочерние элементы объекта **FXMatrix**, даже при том, что это не столь же очевидно, как это было прежде.

Мы должны сделать изменение в методе *layout()*, так как **self** больше не обращается к матричному виджету, а скорее окну прокрутки. Вот на что похожа новая версия *layout()*:

http://media.pragprog.com/titles/fxruby/code/picturebook_e/album_view.rb

```
def layout
	contentWindow.numColumns = [width/PhotoView::MAX_WIDTH, 1].max
	super
end
```

После того, как Вы сделали все модификации к классу AlbumView, это должно выглядеть примерно так:

http://media.pragprog.com/titles/fxruby/code/picturebook_e/album_view.rb

```
require 'photo_view'

class AlbumView < FXScrollWindow

	attr_reader :album

	def initialize(p, album)
		super(p, :opts => LAYOUT_FILL)
		@album = album
		FXMatrix.new(self, :opts => LAYOUT_FILL|MATRIX_BY_COLUMNS)
		@album.each_photo { |photo| add_photo(photo) }
	end

	def layout
		contentWindow.numColumns = [width/PhotoView::MAX_WIDTH, 1].max
		super
	end

	def add_photo(photo)
		PhotoView.new(contentWindow, photo)
	end
end
```

Теперь, когда Вы запускаете приложение, Вы должны видеть вертикальную полосу прокрутки на правой стороне окна всякий раз, когда альбом содержит больше фотографий чем может быть выведено на экран. Это должно выглядеть как на рисунке 5.5, на предыдущей странице. Здесь, я прокрутил албом вниз, чтобы я мог видеть некоторые из дополнительных фотографий в моем албоме.

Мы достигли определённых успехов в этой главе. Мы довели нашу программу от довольно элементарной программы просмотра единственного изображения до приложения способного загрузить любое число фотографий с диска и вывести на экран их как миниатюры в окне с возможностью прокрутки. Однако, есть множество проблем, которые мы всё ещё должны решить, так что давайте займёмся этим.

## Глава 6

# Часть 3: Manage Multiple Albums

Мы должны будем сделать некоторые серьезные структурные изменения в приложении, чтобы отследить многократные фотоальбомы.

Мы начнем, создавая представление, которое позволит нам выводить на экран перечисление имён альбомов в списке альбомов и это позволит пользователю переключаться назад и вперед между альбомами, чтобы просмотреть фотографии, которые они содержат. По пути мы произведем много других изменений, поскольку мы продолжаем к продвигаться к обеспечению полезной функциональности.

Мы начнём с введения в виджет **FXList**, когда мы используем его как основание нашего класса **AlbumListView**. Когда мы встретимся с некоторыми новыми проблемами расположения, мы собираемся обратиться к менеджерам размещения **FXSplitter** и **FXSwitcher**. Мы будем использовать **FXInputDialog** в качестве простого способа собрать информацию от пользователя. Возможно, самое важное, мы получим введение в мощный механизм обновления **GUI FOX**, чтобы автоматически обновить содержание представления альбома всякий раз, когда пользователь выбирает новый альбом из списка. Так что давайте возвратимся к работе.

## 6.1 Create the Album List View

Сначала мы должны продумать представление для списка альбомов и как мы могли бы реализовать это. В этом разделе мы будем учиться как использовать виджет **FXList** для отображения списков элементов, из которых пользователь может выбирать.

Как Вы увидите позже, в Главе 9, *«Sorting Data with List and Table Widgets»*, на странице 115, **FXRuby** обеспечивает много видов виджетов для того, чтобы иметь дело со списочными данными, таким образом, у нас есть много опций для выбора. Давайте сделаем наиболее очевидный выбор, которым является основной виджет **FXList**.

Создайте новый класс **AlbumListView** как подкласс **FXList**, и дайте методу *initialize()* параметры родительского окна, некоторых опции, и экземпляр **AlbumList**:

http://media.pragprog.com/titles/fxruby/code/picturebook_f/album_list_view.rb

```
class AlbumListView < FXList

	attr_reader :album_list

	def initialize(p, opts, album_list)
		super(p, :opts => opts)
		@album_list = album_list
	end
end
```

Теперь мы хотим скачкообразно двинуться к классу PictureBook, который реализует главное окно, и добавим некоторый код к методу *initialize()*, чтобы создать AlbumListView как дочерний элемент главного окна. Отметьте что начиная с главного окна уже есть несколько дочерних окон — представление альбома и строка меню — это list view будет "одноуровневым элементом(sibling)" к этим окнам:

http://media.pragprog.com/titles/fxruby/code/picturebook_f/picturebook.rb

```
def initialize(app)
	super(app, "Picture Book" , :width => 600, :height => 400)
	add_menu_bar
	@album = Album.new("My Photos" )
	@album_list = AlbumList.new
	@album_list.add_album(@album)
	@album_list_view = AlbumListView.new(self,
		LAYOUT_FILL_Y|LAYOUT_SIDE_LEFT, @album_list)
	@album_view = AlbumView.new(self, @album)
end
```

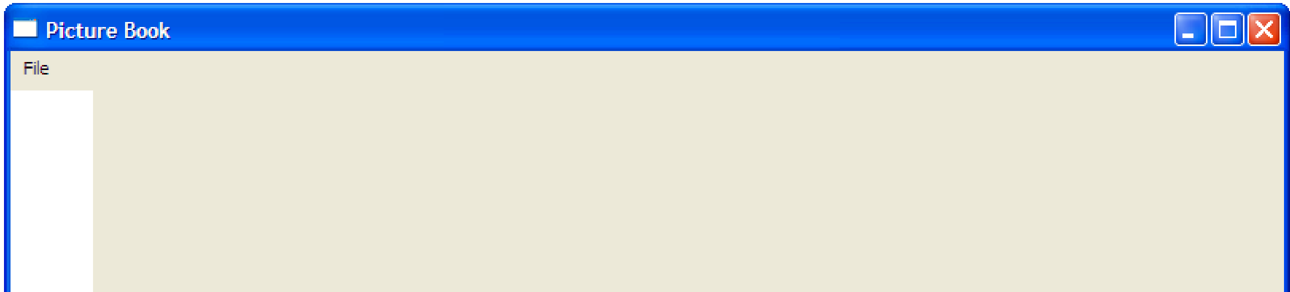



Figure 6.1: After adding list view

Не забывайте добавить **require** чтобы операторы импортировали определения для классов **AlbumList** и **AlbumListView**:

http://media.pragprog.com/titles/fxruby/code/picturebook_f/picturebook.rb

```
require 'album'
require 'album_list'
require 'album_list_view'
require 'album_view'
require 'photo'
```

Теперь давайте выполним программу (Рисунок 6.1). Ясно, что что-то изменилось, но что это за белая полоса вдоль левой стороны окна, которая, как предполагается, была списком?

Часть проблемы - то, что в нашем списке ничего фактически нет. Во-первых, мы добавим метод помощника для **AlbumListView**, который добавляет новый элемент списка для данного альбома:

http://media.pragprog.com/titles/fxruby/code/picturebook_f/album_list_view.rb

```
def add_album(album)
	appendItem(album.title)
end
```

Теперь, давайте добавим некоторый код в метод *initialize()* для **AlbumListView** так, чтобы он выполнял итерации по всем альбомам в списке альбомов и добавлял каждый из их:

http://media.pragprog.com/titles/fxruby/code/picturebook_f/album_list_view.rb

```
@album_list.each_album do |album|
	add_album(album)
end
```

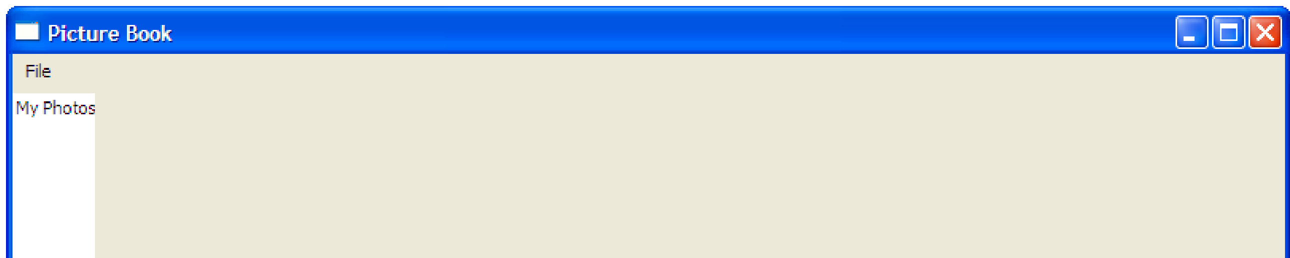



Figure 6.2: Barely wide enough

Теперь выполните программу снова. На сей раз, имя нашего единственного альбома (творчески названный “Мои фотографии”), должен появиться в списке. Проблема теперь (как показано в рисунке 6.2) то, что список недостаточно широк, чтобы вывести на экран все название альбома. Фактически, в зависимости от размера шрифта по умолчанию, который **FOX** выбирает на Вашем компьютере, список, может быть узким или непропорционально широким. Дело в том, что Вы не можете действительно сделать предположение о ширине по умолчанию пустого списка.

**FOX** действительно предоставляет возможности для того, чтобы установить фиксированную ширину виджета, таким образом, мы можем только выбрать некоторую произвольную ширину для списка альбомов, и это решило бы проблему – пока. Смотря вперед, тем не менее, мы могли бы добавить некоторые альбомы с еще более длинными именами, которые тогда потребовали бы нас изменить фиксированную ширину списка. Мы могли, возможно, сделать так, чтобы список динамически изменил свои размеры, становясь достаточно широким чтобы вместить самый широкий текст элемента списка, но это может создать визуальные ограничения. То, в чем мы нуждаемся, является расположением, которое позволит пользователю динамически изменить размеры ширины списка согласно его собственному предпочтению. Чтобы сделать это, мы собираемся использовать разделитель.

## 6.2 Use a Split View

**FXSplitter** - менеджер расположения, который Вы можете использовать для отображения окна изменяемого пользователем. **FXSplitter** - специальный вид менеджера расположения, который управляет двумя дочерними окнами. Когда это сконфигурировано как горизонтальный разделитель, два дочерние окна рядом; когда это - вертикальный разделитель, одно дочернее окно находится сверху, и другое находится в нижней части. Мы собираемся использовать горизонтальную разновидность **FXSplitter**.

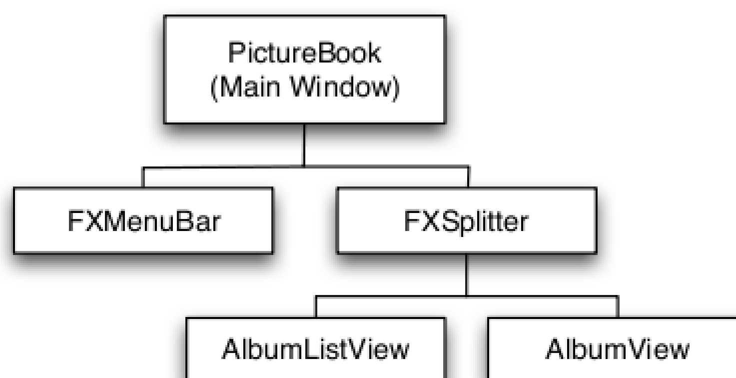



Figure 6.3: Layout hierarchy

Вот план: мы собираемся создать окно разделителя как дочерний элемент главного окна и затем делаем представление списка альбомов и представление альбома дочерними элементами разделителя. Смотрите на рисунок 6.3, чтобы видеть то, что имеет место.

http://media.pragprog.com/titles/fxruby/code/picturebook_f/picturebook.rb

```
splitter = FXSplitter.new(self,
        :opts => SPLITTER_HORIZONTAL|LAYOUT_FILL)
```

Затем, сделайте представление списка альбомов и album view windows как дочерние элементы разделителя:

http://media.pragprog.com/titles/fxruby/code/picturebook_g/picturebook.rb

```
@album_list_view = AlbumListView.new(splitter,
        LAYOUT_FILL, @album_list)
@album_view = AlbumView.new(splitter, @album)
```

Отметьте, что больше нет необходимости определять подсказку **LAYOUT_SIDE_LEFT** для представления списка альбомов. Так как мы добавляем список альбомов первым, он автоматически присваивается левой стороне разделения, а представление альбома размещается на правой стороне.

Вот как выглядит метод *initialize()* для класса PictureBook после этих изменений:

http://media.pragprog.com/titles/fxruby/code/picturebook_g/picturebook.rb

```
def initialize(app)
        super(app, "Picture Book" , :width => 600, :height => 400)
        add_menu_bar
        @album = Album.new("My Photos" )
        @album_list = AlbumList.new
        @album_list.add_album(@album)
        splitter = FXSplitter.new(self,
                :opts => SPLITTER_HORIZONTAL|LAYOUT_FILL)
        @album_list_view = AlbumListView.new(splitter,
                LAYOUT_FILL, @album_list)
        @album_view = AlbumView.new(splitter, @album)
end
```

Теперь, когда Вы выполняете программу, если мышь находится на краю между списком и представлением альбома, курсор мыши должен изменить свою форму на ряд вертикальных полосок со стрелками. Когда курсор принимает эту форму, Вы можете нажать левую кнопку мыши, чтобы "захватить" разделитель и перетащить границу, чтобы изменить размеры разделения, и затем отпустить кнопку мыши.

## 6.3 Switch Between Albums

К настоящему времени Вы должны довольно много понимать о менеджерах размещения и можете использовать их, чтобы решить различные проблемы. Мы узнали о менеджере расположения **FXMatrix**, когда мы создали представление альбома в предыдущем разделе, как использовать **FXSplitter**, чтобы управлять окнами, которые, возможно, должны быть изменены пользователем. В этом разделе мы собираемся узнать о еще одном менеджере расположения - **FXSwitcher**.

Когда пользователь выбирает новый альбом из списка альбомов, мы хотим чтобы альбом обновил себя так, чтобы были показаны фотографии нового выбранного альбома. Один способов сделать это - сначала удалить все экземпляры PhotoView связанные с ранее выбранным альбомом и повторно заполнить AlbumView, используя фотографии нового альбома. Это может, конечно, работать; **FOX** очень эффективен с точки зрения создания и уничтожении окон, и вероятно, что пользователь не заметил бы большую задержку переключения между альбомами, которые содержат относительно небольшое количество фотографий. Это не очень мудрое решение, и для больших альбомов время, требуемое для перезагрузки всех файлов изображений и затем создания объектов **FXJPGImage** для них было бы слишком большим.

Лучшее решение состоит в том, чтобы создать один экземпляр AlbumView для каждого альбома в библиотеке и затем использовать менеджер расположения **FOX FXSwitcher** для быстрого переключения назад и вперед между этими представлениями альбомов. Переключатель может содержать любое число дочерних окон, но выводит на экран только одно из них в одно время.

Мы собираемся изменить наше расположение еще раз так, чтобы правая область **FXSplitter** содержала **FXSwitcher**. Затем мы сделаем окно AlbumView дочерним элементом **FXSwitcher**. Рисунок 6.4 показывает пересмотренную родительско-дочерную иерархию. Первый шаг создаёт переключатель как дочерний элемент разделителя:

http://media.pragprog.com/titles/fxruby/code/picturebook_h/picturebook.rb

```
@switcher = FXSwitcher.new(splitter, :opts => LAYOUT_FILL)
```

Теперь измените первый параметр в вызове в AlbumView.new, чтобы сделать переключатель новым родителем представления альбома:

http://media.pragprog.com/titles/fxruby/code/picturebook_h/picturebook.rb

```
AlbumView.new(@switcher, @album)
```

Отметьте, что мы больше не должны хранить ссылку на единственный экземпляр AlbumView в @album_view. Если мы собираемся иметь дело с многократным альбомом, мы действительно должны начать думать с точки зрения операций для представления текущего альбома. Фактически, давайте создадим небольшую функцию для запроса в списке альбомов индекса выбранного альбома и затем возврата дочернего элемента переключателя с тем же самым индексом:

http://media.pragprog.com/titles/fxruby/code/picturebook_h/picturebook.rb

```
def current_album_view
	@switcher.childAtIndex(@switcher.current)
end
```

Мы можем тогда осуществить это, чтобы обеспечить метод current_album():

http://media.pragprog.com/titles/fxruby/code/picturebook_h/picturebook.rb

```
def current_album
	current_album_view.album
end
```

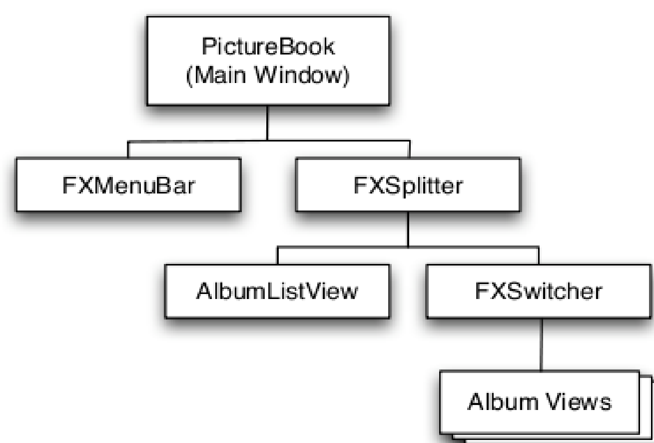



Figure 6.4: Using a switcher

Мы изменяем метод *import_photos()*, чтобы гарантировать что он всегда импортирует фотографии в настоящий момент выбранный альбом:

http://media.pragprog.com/titles/fxruby/code/picturebook_h/picturebook.rb

```
def import_photos(filenames)
	filenames.each do |filename|
		photo = Photo.new(filename)
		current_album.add_photo(photo)
		current_album_view.add_photo(photo)
	end
	current_album_view.create
end
```

Мы уверенно идём в направлении просмотра многократных фотоальбомов. Что делать чтобы мы наконец могли фактически добавить другой альбом или два к набору?

## 6.4 Add New Albums

В предыдущей главе Вы изучили, как использовать вид диалогового окна **FXFileDialog**, чтобы предоставить пользователю интерфейс к файловой системе для того, чтобы выбрать файлы. В этом разделе мы изучим, как использовать **FXInputDialog**, чтобы получить различный вид ввода от пользователя. Мы получим введение в механизм обновления **GUI FOX**, мощный и полезный способ синхронизации пользовательского интерфейса с приложением.

До этой точки мы имели дело с только единственным альбомом, даже хотя мы производили много изменений к коду, чтобы разместить многократные альбомы. Давайте направляться на финишную прямую, добавив новую команду в меню *File*. Найдите метод *add_menu_bar()*, и добавьте команду *New Album...*:

http://media.pragprog.com/titles/fxruby/code/picturebook_h/picturebook.rb

```
new_album_command = FXMenuCommand.new(file_menu, "New Album..." )
new_album_command.connect(SEL_COMMAND) do
	# ...
end
```

Все, что мы должны сделать, запросить у пользователя имя нового альбома, создать новый альбом этим именем, и добавить к списку альбомов его и представление списка альбомов. Мы собираемся использовать **FXInputDialog** класс:

http://media.pragprog.com/titles/fxruby/code/picturebook_h/picturebook.rb

```
album_title = FXInputDialog.getString("My Album" , self, "New Album" , "Name:" )
```

Если пользователь нажмет кнопку *Cancel*, метод *getString()* возвратит ноль. Иначе, *getString()* возвратит заголовок нового альбома, и мы можем использовать это, чтобы выполнить остальную часть команды:

http://media.pragprog.com/titles/fxruby/code/picturebook_h/picturebook.rb

```
if album_title
	album = Album.new(album_title)
	@album_list.add_album(album)
	@album_list_view.add_album(album)
	AlbumView.new(@switcher, album)
end
```

Конечный продукт должен быть похожим на это:

http://media.pragprog.com/titles/fxruby/code/picturebook_h/picturebook.rb

```
new_album_command = FXMenuCommand.new(file_menu, "New Album..." )
new_album_command.connect(SEL_COMMAND) do
	album_title =
		FXInputDialog.getString("My Album" , self, "New Album" , "Name:" )
	if album_title
		album = Album.new(album_title)
		@album_list.add_album(album)
		@album_list_view.add_album(album)
		AlbumView.new(@switcher, album)
	end
end
```

Если Вы выполняете программу в этой точке и пытаетесь добавить альбом, Вы должны видеть, что имя нового альбома обнаруживается в списке альбомов. Если Вы тогда щелкните по тому элементу в списке, Вы ожидаете увидеть новый, пустой альбом в представлении альбома. Но Вы будете разочарованы. Поэтому мы не сказали переключателю, что Вы выбрали новый элемент из список альбомов — до сих пор, там нет никакого соединения.

Во многих инструментариях GUI решение состоит в том, чтобы вернуться к виджету списка альбомов и записать некоторый код, который реагирует на новый список, обновляя переключатель. Вы можете сделать это в **FXRuby**, но я собираюсь показать Вам немного отличающийся подход. Мы будем вместо этого используйте механизм обновления **GUI FOX**, чтобы позволить переключателю обновляться непосредственно на основании выбора в списке.

Перейдите назад до метода *initialize()* для PictureBook, и добавьте этот блок после создания переключателя:

http://media.pragprog.com/titles/fxruby/code/picturebook_h/picturebook.rb

```
@switcher.connect(SEL_UPDATE) do
	@switcher.current = @album_list_view.currentItem
end
```

Определяя обработчик **SEL_UPDATE** для переключателя, мы говорим FOX как обновить состояние переключателя всякий раз, когда состояние приложения изменяется. В настоящий момент показанный элемент переключателя должен отразить выбранный элемент списка. Этот обработчик обновления вызывают для нас автоматически. Мы будем говорить о механизме обновления GUI более подробно в Разделе 7.4, *«Синхронизация Пользовательского интерфейса с Данными приложения»*, на странице 91.

Теперь нам нужно соединить все должным образом. Давайте сделаем небольшой эксперимент. Запустите программу, и импортируйте ряд фотографий в начальный альбом. Теперь, добавьте новый альбом к списку, и выберите тот альбом. Переключатель должен должным образом обновить себя и показать Вам новый (и пустой) альбом. Импортируйте некоторые фотографии в этот альбом, и затем подтвердите то, что Вы можете переключиться назад и вперед между двумя. Если Вы действительно чувствуете как сходите с ума, добавьте другой альбом или два. Возможно пригласите соседей посмотреть.

## 6.5 Serialize the Album List with YAML

В этом заключительном шаге мы изучим, как использовать YAML Ruby, библиотеку Ain’t Markup Language (YAML), чтобы сохранить данные приложения в файле и затем читать их когда программа запускается. Хотя это не имеет непосредственное

отношение к разработке **FXRuby**, мы видим что мы действительно должны произвести некоторые изменения в коде программы, чтобы гарантировать что пользовательский интерфейс обновлен должным образом после того, как новые данные альбома загружены.

Если Вы запускали приложение несколько раз, Вы вероятно, сильно раздражались. Каждый раз, когда программа запускается, Вы получаете пустой экран, и Вы должны воссоздать альбомы и повторно импортировать свои фотографии. Мы должны сделать некоторые усилия для того, чтобы сохранить список альбомов на диске, когда программа закрывается и затем перезагрузить эти альбомы, когда программа открывается.

Мы можем сделать это несколькими способами. Если Вы программировали на Java, Вашим первым инстинктом должна быть разработка XML-схемы. Это описывает отношения между списком альбомов, альбомами, и фотографиями, содержавшимися там. XML не был бы худшим выбором, это - хорошо понятый, удобочитаемый путь к сохранению структурированных данных. Стандартная библиотека Ruby даже включает REXML, большой модуль для чтения и записи XML-документов. Несмотря на популярность, однако, XML - не всегда лучшее решение для всех проблем хранения данных.

Другое решение - это хранить информацию в реляционной базе данных. Ruby оказывает превосходную поддержку для того, чтобы работать с базами данных если это имеет смысл для Вашего приложения, но похоже на симпатичного тяжеловеса решение для наших текущих потребностей. Стандартная библиотека Ruby обеспечивает поддержку двух легких схем сериализации, посредством модулей Marshal и YAML.

Я не большой поклонник данных которые не удобочитаемы, и так как модуль Ruby Marshal хранит данные в собственном двоичном формате, мы будем использовать файл YAML в качестве нашего персистентного хранилища для приложения.

Во-первых, это легко. Давайте запишем некоторый код, чтобы сохранить содержание списка альбомов в файле. Метод *store_album_list()* создает файл picturebook.yml и использует метод YAML.dump, чтобы записать список альбомов:

http://media.pragprog.com/titles/fxruby/code/picturebook_i/picturebook.rb

```
def store_album_list
    File.open("picturebook.yml" , "w" ) do |io|
        io.write(YAML.dump(@album_list))
    end
end
```

Мы хотим, чтобы это произошло, когда пользователь выбирает команду *Exit* из меню *File*. Давайте изменим обработчик *exit*, чтобы добавить вызов *store_album_list() перед вызовом exit():*

http://media.pragprog.com/titles/fxruby/code/picturebook_i/picturebook.rb

```
exit_cmd.connect(SEL_COMMAND) do
    store_album_list
    exit
end
```

Теперь, хитрая часть. Очевидно, ключевая вещь, которую мы должны сделать, попробовать загрузить сохраненный файл списка альбомов, если он существует. Если он не существует, то мы вернёмся назад к нашему поведению по умолчанию и создадим новый список с одним альбомом:

http://media.pragprog.com/titles/fxruby/code/picturebook_i/picturebook.rb

```
begin
	@album_list = YAML.load_file("picturebook.yml" )
rescue
	@album_list = AlbumList.new
	@album_list.add_album(Album.new("My Photos" ))
end
```

Не забудьте добавить оператор **require** в начало picturebook.rb для импорта библиотеки YAML:

http://media.pragprog.com/titles/fxruby/code/picturebook_i/picturebook.rb

```
require 'yaml'
```

Позвольте нам произвести код, связанный с заполнением **AlbumListView** из содержания списка альбомов в accesor метод:

http://media.pragprog.com/titles/fxruby/code/picturebook_i/album_list_view.rb

```
def album_list=(albums)
	@album_list = albums
	@album_list.each_album do |album|
		add_album(album)
	end
end
```

Всякий раз, когда новый экземпляр **AlbumList** присвоен **AlbumListVie**w, этот код вызовет *add_album()* для каждого альбома в списке. Теперь, когда этот accessor method на месте, мы не нуждаемся в передаче списка альбомов в метод *initialize()* представления списка, так что давайте удалим это из списка параметров:

http://media.pragprog.com/titles/fxruby/code/picturebook_i/album_list_view.rb

```
def initialize(p, opts)
	super(p, :opts => opts)
end
```

Мы также должны изменить строку в PictureBook, где мы фактически создаем **AlbumListView** и гарантируем, что мы больше не передаем album_list в *AlbumListView.new()*:

http://media.pragprog.com/titles/fxruby/code/picturebook_i/picturebook.rb

```
@album_list_view = AlbumListView.new(splitter, LAYOUT_FILL)
```

Не забывайте присваивать список альбомов представлению. Иначе, мы никогда не будем видеть содержание наших альбомов:

http://media.pragprog.com/titles/fxruby/code/picturebook_i/picturebook.rb

```
@album_list_view.album_list = @album_list
```

Теперь добавьте строку к методу *add_album()*, чтобы создать новый AlbumView в то же самое время когда мы добавляем элемент списка:

http://media.pragprog.com/titles/fxruby/code/picturebook_i/album_list_view.rb

```
def add_album(album)
	appendItem(album.title)
	AlbumView.new(@switcher, album)
end
```

Чтобы это работало, мы нуждаемся в способе сообщить **AlbumListView** о переключателе:

http://media.pragprog.com/titles/fxruby/code/picturebook_i/album_list_view.rb

```
def switcher=(sw)
	@switcher = sw
end
```

Вот то, на что похожа новая и улучшенная версия **AlbumListView**:

http://media.pragprog.com/titles/fxruby/code/picturebook_i/album_list_view.rb

```
class AlbumListView < FXList

	attr_accessor :album_list

	def initialize(p, opts)
		super(p, :opts => opts)
	end

	def album_list=(albums)
		@album_list = albums
		@album_list.each_album do |album|
			add_album(album)
		end
	end

	def switcher=(sw)
		@switcher = sw
	end

	def add_album(album)
		appendItem(album.title)
		AlbumView.new(@switcher, album)
	end
end
```

Последняя небольшая пара изменений имеет место в методе *initialize()* для PictureBook. Во-первых, мы устанавливаем переключатель для представления списка альбомов:

http://media.pragprog.com/titles/fxruby/code/picturebook_i/picturebook.rb

```
@album_list_view.switcher = @switcher
```

Наконец, начиная с присвоения списка альбомов к *album list view* триггеры вызывают *add_album()* для каждого альбома в списке и *add_album()* создает **AlbumView**, соответствующую недавно добавленным альбомам, мы можем (и должны) удалить строку из метода *initialize()* PictureBook, которая создаёт объект AlbumView по умолчанию.

Начиная с упорядочивания этих строк в *initialize()*, важно, чтобы окончательная версия метод была такой:

http://media.pragprog.com/titles/fxruby/code/picturebook_i/picturebook.rb

```
def initialize(app)
	super(app, "Picture Book" , :width => 600, :height => 400)
	add_menu_bar
	begin
		@album_list = YAML.load_file("picturebook.yml" )
	rescue
		@album_list = AlbumList.new
		@album_list.add_album(Album.new("My Photos" ))
	end
	splitter = FXSplitter.new(self,
		:opts => SPLITTER_HORIZONTAL|LAYOUT_FILL)
	@album_list_view = AlbumListView.new(splitter, LAYOUT_FILL)
	@switcher = FXSwitcher.new(splitter, :opts => LAYOUT_FILL)
	@switcher.connect(SEL_UPDATE) do
		@switcher.current = @album_list_view.currentItem
	end
	@album_list_view.switcher = @switcher
	@album_list_view.album_list = @album_list
end
```

В этой точке Вы должны быть в состоянии использовать Picture Book, чтобы создать альбомы и добавить фотографии к ним без страха, что вся Ваша работа будет потеряна когда Вы выходите из программы. Это то, куда мы собирались дойти и остановим работу над приложением, но прежде, чем мы перейдём к другим темам, позвольте нам занять минуту, чтобы подумать о некоторых дополнительных улучшениях что Вы могли бы сделать после того, как Вы закончили книгу.

## 6.6 So, What Now?

В прошлых трех главах мы существенно расширили функциональность приложения относительно немногими изменениями к основному коду. Поскольку Вы становитесь профессионалом в разработке на **FXRuby**, Вы должны думать, что это - действительно гибкий инструментарий для создания приложения этим способом.

Цель этого изложения не создать реальное приложение, поскольку это должно было объяснить Вам что надо делать, чтобы разработать приложения с FXRuby и рассказать Вам о большом количестве методов которые Вы будете использовать в своих собственных проектах. Однако Вы могли бы сделать некоторое число улучшений к этому приложению.

Например, Picture Book в настоящий момент поддерживает только импорт и отображение изображений JPEG. В Главе 11, «*Создание Визуально Богатых Пользовательских интерфейсов*», на странице 142 мы рассмотрим большое количество деталей о различных форматах изображения, которые поддерживает FOX. Вооружившись этим знанием, Вы должны быть способны расширить приложение так, чтобы можно было импортировать и выводить на экран изображения любого типа. Вы также получите некоторое представление о других видах манипуляций изображением, которые Вы хотели бы внести в свою версию приложения, чтобы добавить например, обрезку и вращение изображений.

Другое ограничение Picture Book в его текущем виде то, что размер миниатюры изображения привязан к 200 пиксельным квадратам. В Главе 8, «*Создание Простых Виджетов*», на странице 100, Вы получите сведения о некоторых других видах виджетов, которые Вы можете встроить в пользовательский интерфейс. Вы можете решить, что хотите использовать некоторые из этих инструментов, чтобы позволить пользователю редактировать размеры миниатюры, имя фотографии или альбома или некоторые другие виды данных.

Или рассмотрите список альбомов. Наша организационная структура для того списка немного негибка, мы не можем группировать подобные альбомы в папки — это большой плоский список. В Главе 9, *«Сортировка Данных в Списках и Табличных Виджетах»*, на странице 115, мы будем рассматривать (между прочим) Виджет **FXTreeList**, который Вы могли бы использовать для существенно более глубокого вложенного вида списка альбомов.

Вы не узнаете о каждом укромном уголке **FOX** и **FXRuby** из этой книги, но к тому времени, когда Вы закончите читать, Вы будете иметь достаточно твердую основу, чтобы исследовать больше расширенных функций, которые должен предложить инструментарий. Поскольку Ваш путь пролегает через остальную часть книги, я надеюсь, что Вы будете возвращаться к этому приложению как к своего рода испытательному стенду для того, чтобы попробовать новые вещи которые Вы узнаете о **GUI** с **FXRuby**.

**Часть II**

# FXRuby Fundamentals

**Глава 7**

## FXRuby Under the Hood

Теперь, когда мы проложили себе путь посредством создания всего приложения от начала до конца пора порыть немного глубже в **FXRuby**. Хотя Вы можете, конечно, обойтись поверхностным пониманием библиотеки **FXRuby**, затратьте время и изучите как работает **FXRuby** под капотом. Это поможет Вам написать более гибкие, удобные в сопровождении, и эффективные приложения.

В то время как мы создавали приложение Picture Book, мы учились как использовать метод *connect()*, чтобы связать несколько пользовательских действий, такие как щелчки мышью, с блоками кода **Ruby**. Эта функциональность основана на мощной системе управления событиями обмена сообщениями, и в этой главе мы узнаем больше, как использовать в своих интересах эту систему, чтобы обработать много видов событий приложения. **Fox** использует эту систему как основу для автоматического обновления **GUI** при синхронизации с данными приложения. Мы рассмотрим более внимательное, как это работает со множеством виджетов **FXRuby**, как использовать цели данных в качестве высокоуровневой альтернативы механизму непосредственного обновления **GUI**.

Создатели **FOX** гордится им как одним из самых быстрых и дружественных инструментариев **GUI**, и хотя необязательно понимать как **FOX** оптимизирован для достижения производительности, мы исследуем некоторые вещи. **FOX** гарантирует, что Ваши приложения остаются быстрыми и быстро реагирующими из пользовательского интерфейса. **FOX** также делает намного более явное различие между клиентскими и серверными представлениями объектов пользовательского интерфейса, чем другие инструментарии **GUI**, таким образом, мы будем учиться как управление различными представлениями помогает Вам сделать своё приложение менее ресурсоёмким.

Figure 7.1: Objects send messages to other objects

К концу этой главы у Вас будет намного больше всестороннего понимания того, как FXRuby работает под капотом, и Вы получите прочную основу, если займётесь остальной частью этой книги.

### 7.1 Event-Driven Programming

Много сценариев и программ, которые Вы пишете в **Ruby**, следуют определённым предсказуемым правилам. Типичная программа могла бы открыть файл, читать некоторые данные, выполнить вычисления данных, и сообщите о результатах. Программа могла бы содержать немного условной логики, которая заставляет ее делать переходы и другие разные вещи в зависимости от входных данных.

Программы **FXRuby** событийно-управляемы и ведут себя несколько по-другому. После некоторой инициализации программа **FXRuby** вводит то, что известно как цикл

события: программа ожидает события, она отвечает на это событие, и затем продолжает ожидать следующего события. Чаще всего это пользователь, который генерирует событие, является ли это щелчком кнопки мыши, вводом некоторого текста в текстовом поле, или некоторым другим действием. В другое время это может быть операционная система или работа с окнами, которые генерирует события, такой как, появление сигнала. Другие **GUI** реализуют событийно-управляемое программирование во множеством способов, но FXRuby моделирует событие как один объект, отправитель отправляет сообщение к другому объекту - цели (см. рисунок 7.1).

### Типы сообщения, Идентификаторы и Данные

Каждое сообщение, которое отправлено от одного объекта **FXRuby** к другому, состоит из типа сообщения, идентификатора сообщения, и некоторых данных сообщения. Тип сообщения - константа, имя которой начинается с **SEL_**. Вы уже видели некоторые примеры этого, **SEL_COMMAND** тип сообщения.

Идентификатор сообщения - также константа, и он используется получателем сообщения, чтобы различить различные сообщения того же самого типа. Например, если цель ожидает получения сообщения **SEL_CHANGED** от двух различных отправителей, это могло бы отразиться на отправителях, используя два различных идентификатора сообщения определяется откуда эти сообщения. Далее, если цель хочет реализовать различное поведение для того же самого типа сообщения, это требует числового идентификатора сообщения. Например, когда виджет **FXTextField** получает сообщение **SEL_COMMAND** с идентификатором **ID_CURSOR_HOME**, это перемещает курсор в начало строки, но когда он получает **SEL_COMMAND** с идентификатором **ID_CURSOR_END**, он перемещает курсор в конец строки.

Наконец, данные сообщения - это объект, который предоставляет некоторый дополнительным контекст для сообщения. Например, когда объект **FXText** отправляет сообщение **SEL_INSERTED** к цели, он передает объект **FXTextChange** который указывает на то, какой текст был вставлен и где он был вставлен.

Вы можете использовать документацию **API**, чтобы идентифицировать какие типы сообщений определенный виджет может посылать к его цели и какие данные для этих сообщений. Например, рисунок 7.2, на следующей странице, показывает часть документации для класса **FXTable**. Заголовок "Events" выводит таблицу, перечисляющую типы сообщений и информация для каждого сообщения которую **FXTable** передает к цели. Есть подобная информация в документации API для каждого виджета **FXRuby**.

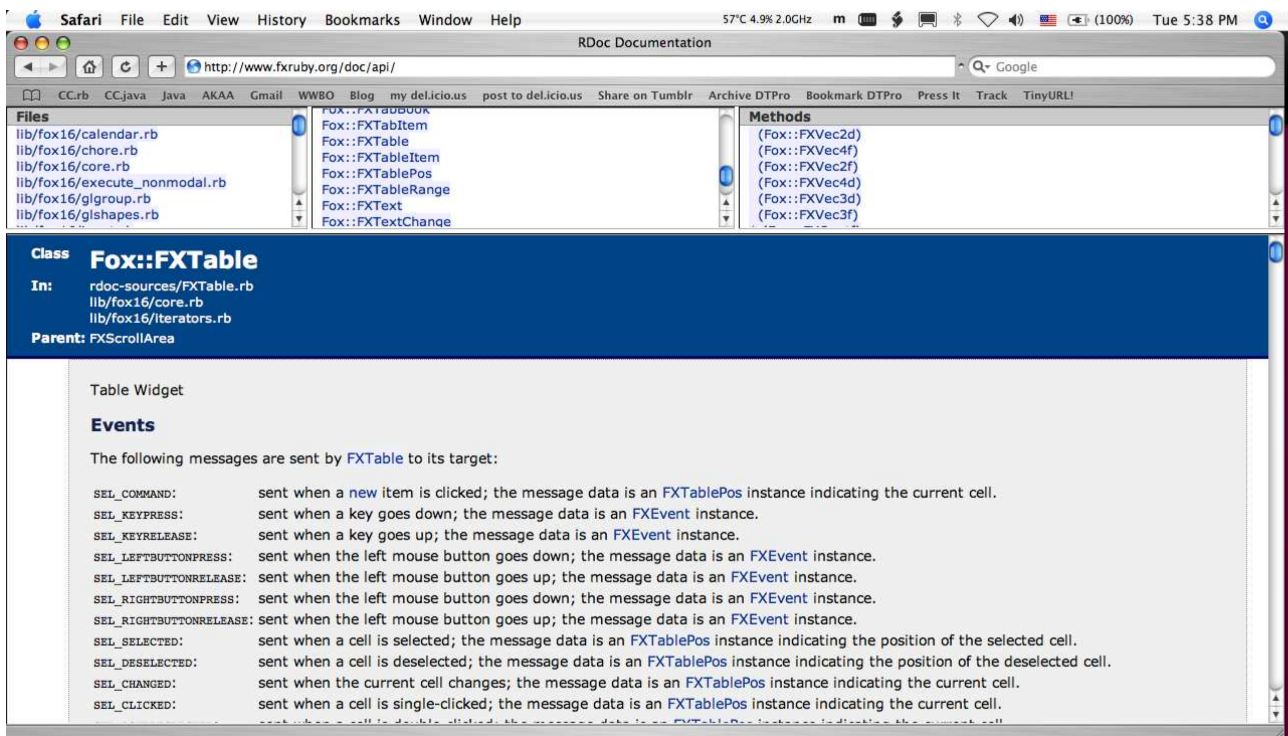



Figure 7.2: API documentation for FXTable

## Messages and Targets

Учитывая, как работает система обмена сообщениями **FOX**, что нам нужно сделать в нашем коде программы в ответ на эти сообщения? Предположите, что Ваше приложение имеет табличный виджет, и Вы хотите знать, когда пользователь щелкает по одной из ячеек в этой таблице, чтобы Вы могли вывести на экран некоторые дополнительные детали о данных в той ячейке. Вы бы посмотрели в документации API для класса **FXTable**, чтобы узнать, какие виды сообщений он отправляет когда пользователь щелкает по ячейке таблицы. Как оказывается, есть много кандидатов (включая **SEL_COMMAND**, **SEL_CLICKED**, и **SEL_SELECTED**) и кажется, что они отвечают всем требованиям.

Но это не так. Даже при том, что единственное действие (как щелчок по ячейке в таблице), может привести ко многим видам сообщений, отличающихся семантикой связанной с каждым типом сообщения. Например, сообщение **SEL_SELECTED** имеет отношение к новой выбираемой ячейке, происходит ли тот выбор в интерактивном режиме как результат щелчка мыши или программно в результате вызова метода *selectRange()* таблицы. Различие между **SEL_CLICKED** и **SEL_COMMAND** даже более тонкое. Таблица отправит сообщение **SEL_CLICKED**, если пользователь щелкнет где угодно на таблице, но она отправит его с сообщением **SEL_COMMAND** только если щелчок был фактически в табличной ячейке.

В этом примере сообщение **SEL_COMMAND** - самый подходящий кандидат. В большинстве случаев, самое полезное сообщение, которое виджет посылает к цели, является сообщением **SEL_COMMAND**. Определенное значение **SEL_COMMAND**, конечно, отличается для различного виды виджетов.

Теперь, когда мы остановились на **SEL_COMMAND** для табличного сообщения, мы должны сказать объекту **FXTable**, какой объект передать в цель и какой идентификатор сообщения оно должно использовать, когда он передает сообщение к той цели. Единственный способ сделать это - определить цель и идентификатор сообщения, когда Вы создаете таблицу:

```
table = FXTable.new(p,
:target => target_object, :selector => message_identifier, ...)
```

Вы можете также присвоить (или изменить) цель и идентификатор сообщения после создания таблицы, используя атрибуты цели и селектора:

```
table.target = target_object
table.selector = message_identifier
```

Если Вы умный читатель то, вероятно, поняли тот факт, что (по историческим причинам) атрибут, который Вы используете для чтения или установки идентификатора сообщения, называется селектором. Доверяйте мне, это действительно идентификатор сообщения.

Теперь, в этой точке разговора, я могу перейти к большому количеству страшных деталей о том, как установить карту сообщения, которая говорит целевому объекту какой из его методов надо вызвать, при получении сообщения анализируя детали переданного типа и идентификатора. Храбрые души, которые вынесли это в первые годы разработки **FXRUBY** должны были пройти через этот процесс, и люди, которые пользуются библиотекой **FOX** для их приложений **GUI** на **C++** все еще должны делать так. Хорошие новости: пользователи **FXRuby** не должны иметь дело с картами сообщения **FOX** больше.

## Connecting Messages to Code

Метод *connect()* обеспечивает прямой способ соединить сообщения отправленные из виджета до блока кода, который обрабатывает их. Под капотом: метод *connect()* создает целевой объект и идентификатор сообщения и затем присваивает это отправителю сообщения, таким образом, мы существенно не изменили работу модели событийно-управляемого программирования **FOX**. Но для Вас это означает, что та обработка сообщения **SEL_COMMAND** от таблицы нуждается только нескольких строках кода:

```
table.connect(SEL_COMMAND) do |sender, selector, data|
	# Handle a click inside a table cell
end
```

В этом примере мы только передаем единственный параметр в *connect()*, и это - тип сообщения, обработкой которого мы интересуемся. Есть различия в этом, что я поясню через мгновение, но это наиболее распространено форма. Мы "соединяем" сообщение **SEL_COMMAND** от таблицы до блока **Ruby**, который ожидает три параметра. Имена, которые Вы используете для этих параметров в блоке, конечно, ваше дело, поскольку Вы сохраняете их значения прямо. Первый параметр, имя отправителя - это ссылка на объект, который отправил сообщение (таблица). Второй параметр - селектор, который объединяет тип сообщения и идентификатор (больше на этом через мгновение). Третий параметр, данными - это ссылка на данные сообщения.

Согласно документации API для класса **FXTable**, данные сообщения, связанные с сообщением **SEL_COMMAND** - это **FXTablePos** экземпляр, указывающий на текущую ячейку. От него мы можем извлечь строку и номер столбца ячейки, по которой щелкнули:

```
table.connect(SEL_COMMAND) do |sender, selector, data|
	puts "User clicked cell at row: #{data.row}, column: #{data.col}"
end
```

Есть немного отличающийся способ вызвать *connect()*, и это полезно когда Вы хотите использовать тот же самый код обработчика для многократных виджетов или когда Вы только хотите выделить код обработчика из его собственного метода. Это различие включает передачу экземпляра Метода (или некоторый другой отчасти

вызываемого объекта, как экземпляр Proc) как второго параметра *connect()*:

```
def table_cell_clicked(sender, selector, data)
	puts "User clicked cell at row: #{data.row}, column: #{data.col}"
end

table.connect(SEL_COMMAND, method(:table_cell_clicked))
```

Эти две формы функционально эквивалентны, и выбор между ними, главным образом вопрос предпочтения. Я действительно рекомендую, чтобы Вы выделили код обработчика в отдельный метод, если это - больше чем несколько строк, прежде всего чтобы сохранить читаемость кода.

Наконец, я упоминал ранее, что второй параметр обработчика - это значение, которое комбинирует тип сообщения и идентификатор. Вы можете извлечь тип сообщения от этого значения, используя метод **FXSELTYPE()**, и Вы может извлечь идентификатор, используя метод **FXSELID()**:

```
table.connect(SEL_COMMAND) do |sender, selector, data|
	puts "The message type is: #{FXSELTYPE(selector)}"
	puts "and the message identifier is #{FXSELID(selector)}"
end
```

Практически, эта информация не очень полезна, потому что Вы обычно знаете, какой тип сообщения Вы обрабатываете. Это могло быть полезно, однако, если Вы используете тот же самый метод, чтобы обработать сообщения от больше чем одного объекта.

## 7.2 Mouse and Keyboard Events

В зависимости от сложности приложения, которое Вы создаёте, Вам вероятно не придется иметь дело с низкоуровневыми событиями мыши и клавиатуры непосредственно. Например, Вы уже видели, что **FOX** синтезирует щелчки кнопок мыши в **SEL_COMMAND** и другие типы сообщений от виджетов до их целей. Однако, есть некоторые случаи когда Вы действительно должны получить *basic event data* для мыши и клавиатуры и мы рассмотрим, как сделать это в этом разделе.

### Handling Mouse Events

Обычно **FOX** посылает связанные с мышью сообщения тому окну, на какое указывает курсор мыши. Когда нажата левая кнопка мыши, **FOX** отправит сообщение **SEL_LEFTBUTTONPRESS** этому окну, и когда левая кнопка мыши отпущена, **FOX** отправит ему соответствующее сообщение **SEL_LEFTBUTTONRELEASE**. Если Вы используете мышь с двумя или тремя кнопками, есть соответствующие **SEL_MIDDLEBUTTONPRESS**, **SEL_MIDDLEBUTTONRELEASE**, **SEL_RIGHTBUTTONPRESS**, так же как **SEL_RIGHTBUTTONRELEASE**, и если у Вашей мыши есть колесико прокрутки, **FOX** отправит сообщение **SEL_MOUSEWHEEL**, когда колесо будет прокручено вверх или вниз.

Message data связаны с событиями мыши и клавиатуры (и некоторые другие типы сообщений) объекта **FXEvent**. Обычно, единственная вещь, которую Вы хотите знать о нажатии кнопки и возникновении сообщения это то, что оно произошло, таким образом, связанные данные сообщения не важны. Однако, для некоторых приложений Вам может понадобиться знать точно на что курсор мыши указывал, когда кнопка мыши была нажата (или отпущена). В этих случаях Вы можете проверить значения атрибутов *win_x* и *win_y* в данных сообщения, которые скажут Вам координаты **x** и **y** в локальной системе координат для окна где событие имело место.

```
my_window.connect(SEL_LEFTBUTTONPRESS) do |sender, sel, event|
	p "Button pressed at coordinates (#{event.win_x}, #{event.win_y})"
end
```

Как альтернативу, Вы можете смотреть атрибуты *root_x* и *root_y*, чтобы получить координаты относительно корневого окна.

Когда мышь переместится, Вы получите сообщение **SEL_MOTION**. Как и в случае нажатия кнопки и возникновения события, данные сообщения укажут на текущую позицию курсора мыши (куда мышь переместилась). **FOX** также отслеживает предыдущую позицию мыши в атрибутах *last_x* и *last_y*. Помните, что значения *last_x* и *last_y* это координаты окна, не *root* координаты.

## Handling Keyboard Events

В предыдущем разделе мы говорили о том как события связанные с мышью передаются окну, на которое указывает курсор мыши. Подобно этому, связанные с клавиатурой сообщения всегда отправляются окну которое в данный момент содержит клавиатурный фокус.

Когда пользователь нажимает клавишу на клавиатуре, **FOX** отправит сообщение **SEL_KEYPRESS** объекту который в настоящий момент имеет фокус. Когда клавиша будет отпущена, FOX отправит сообщение **SEL_KEYRELEASE**. В обоих случаях данные передаются с сообщением экземпляра **FXEvent**, который включает информацию, о том какая клавиша была нажата, когда сообщение было сгенерировано. Атрибуты **FXEvent** в которых Вы будете больше всего заинтересованы - это *code* и *state*.

Код события говорит Вам, какая клавиша была нажата (или отпущена). Это значение будет соответствовать одной из символьных констант, перечисленных в документации API для модуля **FOX** (http://www.fxruby.org/doc/api/classes/Fox.html). Имена этих констант начинаются с префикса **KEY_**, и в большинстве случаев код клавиши будет иметь понятное имя. Например, когда Вы нажимаете клавишу «a», код события будет **KEY_a**. Для некоторых более неясных случаев Вы можете должны сделать немного "reverse engineering", чтобы выяснить какой код **FOX** отправляет Вам. Получить ее числовое значение и тогда поискать имя константы в документации.

Состояние события говорит Вам, какие модифицирующие клавиши были нажаты во время генерации события. Вы можете протестировать это логически Выполнением операции "И" состояние события с флагами модификатора, перечисленными на рисунке 7.3. Например, следующий кодовый набор *shift_check.checkState* будет *true*, если клавиша *Shift* была нажата, когда событие было сгенерировано:

http://media.pragprog.com/titles/fxruby/code/keyboard.rb

```
self.connect(SEL_KEYPRESS) do |sender, sel, event|
	shift_check.checkState = (event.state & SHIFTMASK) != 0
end
```

| **Modifier Flag** | **Meaning** |
|---|---|
| ALTMASK | Alt key is pressed |
| CAPSLOCKMASK | Caps Lock key is pressed |
| CONTROLMASK | Ctrl key is pressed |
| METAMASK | Meta key is pressed |
| NUMLOCKMASK | NumLock key is pressed |
| SCROLLLOCKMASK | ScrollLock key is pressed |
| SHIFTMASK | Shift key is pressed |

Figure 7.3: Keypress modifier flags

Мы повторно осветим проблемы связанные с событиями клавиатуры позже в Разделе 8.1, *«Getting Pushy with Buttons»*, на странице 104, когда мы рассмотрим на то, как

определять акселераторы и горячие клавиши для виджетов.

## 7.3 Timers, Chores, Signals, and Input Events

События мыши и клавиатуры всегда сгенерированы пользователем непосредственно при взаимодействии с приложением. Однако, много других видов события могут иметь место в приложении FXRuby, и следующих разделах будет описано, как Вы можете использовать их в Ваших приложениях.

### Scheduling Tasks with Timeout Events

Когда Вы регистрируете событие тайм-аута в приложении, Вы просите FOX отправить сообщение Вашему приложению в некоторый момент в будущем. Например, предположите, что Вы хотели бы добавить возможность запланированного резервного копирования к каждые пять минут в Вашем приложении и автоматически сохранять работу пользователя:

```
app.addTimeout(5*60*1000) do
	# invoke the "save" operation
end
```

Первый параметр *addTimeout()* это количество времени в миллисекундах, **FOX** должен ожидать прежде, чем инициировать событие тайм-аута. Код, показанный ранее, не делает полностью то, что мы хотим, потому что событие тайм-аута - однократно; как только это выстрелило, **FOX** забывает о исходном запросе.

Чтобы заставить тайм-аут происходить каждые пять минут, мы должны передать в параметр *:repeat*

```
app.addTimeout(5*60*1000, :repeat => true) do
	# invoke the "save" operation, then re-register the timeout
end
```

Если Вы предпочли бы переместить обрабатывающий тайм-аут код из блока в метод, Вы можете передать код в метод объекта как второй параметр *addTimeout()*:

```
def save_data(sender, sel, data)
	# ...
end

app.addTimeout(5*60*1000, method(:save_data), :repeat => true)
```

*Отметьте, что когда Вы используете метод экземпляра как обработчик тайм-аута, сигнатура метода должна включать три стандартных параметра, как показано в этом примере.*

Метод *addTimeout()* возвращает значение которое полезно, только если Вы нуждаетесь в определении, находится ли событие тайм-аута все еще на рассмотрении, узнать сколько времени осталось до выполнения или отменить событие тайм-аута прежде, чем оно выполнится:

```
timeout = app.addTimeout(5*60*1000, :repeat => true) do
	# invoke the "save" operation, then re-register the timeout
end

# Elsewhere in the application code
if app.hasTimeout?(timeout) && app.remainingTimeout(timeout) < 30000
	app.removeTimeout(timeout)
end
```

Этот код говорит что если ранее зарегистрированный тайм-аут является все еще активным и есть не меньше чем тридцать секунд перед выполнением тайм-аута, приложение может отменить этот тайм-аут.

## Doing Chores in Idle Time

Другой способ попросить FOX, перенести Ваше приложене в некоторый момент в будущее – это зарегистрировать *chore*. Когда Вы работаете с интерактивным приложением, которое Вы создаёте с **FXRuby**, компьютер имеет много свободного времени ожидая от Вас следующее действие. Вместо того, чтобы позволить этому времени пропадать зря, Вы можете сказать **FOX** использовать это время простоя, чтобы заботиться о различном специализированном обслуживании задачи, которые не чувствительны ко времени. В отличие от событий тайм-аута, которые срабатывают в определённое время, *chores* обрабатываются как только очередь событий **FOX** становится пустой:

```
app.addChore do
	# take out the trash as soon as we get a chance
end
```

Кроме этого важного различия, *chores* ведут себя подобно событиям тайм-аута. Chore срабатывает один раз и затем отключается, если Вы не передадите параметр *:repeat* к *addChore()*. Вы можете использовать метод *hasChore?()*, чтобы определить, ожидает ли *chore* что-либо для обработки и использовать *removeChore()*, чтобы отменить ранее зарегистрированный *chore* прежде, чем он сработает. Конечно, нет никакого эквивалента для метода *remainingTimeout()*, потому что даже **FOX** не знает заранее когда будет время простоя, чтобы обработать *chore*.

Будьте осторожны относительно использования повторяющихся *chores*. Ваше приложение обычно будет иметь намного больше времени простоя, чем Вы могли бы ожидать, и это – полезная вещь. Когда Ваше приложение отдыхает, это позволяет Вашему компьютеру посвятить часть его времени к другим приложениям, которые работают рядом с Вашим **GUI**. Если Вы планируете *chore*, которое повторяется много раз, Ваше приложение начнет съедать процессорное время, и его производительность и производительность других работающих приложений пострадает.

## Handling Operating System Signals

Операционная система отправляет сигнал приложению, когда она желает сообщить о некоторой исключительной ситуации. Например, если Вы имеете когда-либо писанную программу на C/C++, которая попыталась разыменовать указатель **NULL**, Вы получите вероятно, страшный сигнал “нарушение сегментации” (**SIGSEGV**).

Вы можете зарегистрировать обработчик сигналов в своем приложении, чтобы перехватывать эти сигналы и делать некоторую обработку в ответ на них. Например, большинство операционных систем отвечает на **Ctrl+C**, отправляя сигнал **SIGINT** в приложение. По умолчанию, процесс в котором Ваше приложение работает, будет завершен при получении этого сигнала. Для гарантии, что Ваше приложение сделает необходимую уборку прежде, чем будет завершено, Вы можете установить обработчик для того сигнала:

```
app.addSignal("SIGINT" ) do
	# save the user's work, then exit the application
end
```

Первым параметром *addSignal()* является строка, указывающая на имя сигнала, который Вы желаете перехватить. Имена сигналов - это стандарт **POSIX** и поддерживаются методом Ruby *trap()*.

Говоря о методе *trap()*, Вы могли бы задаваться вопросом, является ли уместным использовать *addSignal()* или *trap()*, когда Вы должны ответить на определенный сигнал. Оказывается, что Вы можете использовать что угодно, и это - действительно только вопрос предпочтения. Отметьте это, если Вы регистрируете более чем один обработчик сигнала для того же самого сигнала, последний зарегистрированный обработчик - тот, который будет использоваться:

```
app.addSignal("SIGINT" ) do
	# this handler for SIGINT is registered first...
end
Signal.trap("SIGINT" ) do
	# ... but this one replaces it.
end
```

## Reacting to I/O with Input Events

Мы посмотрим как Ваше приложение **FXRuby** может обработать входные события. Вы хотели бы использовать эту возможность, чтобы иметь дело с вводом с разных мест кроме непосредственно с **GUI**, а именно, каналов или сокетов. Если Ваше приложение должно реагировать на данные, записанные в канале, Вы могли бы установить таймер или *chore*, чтобы периодически проверять этот канал на наличие новых данных, но это - довольно неэффективный способ. Лучший подход должен использовать в своих интересах механизмы операционной системы для того, чтобы отреагировать на эти изменения, и метод *addInput()* обеспечивает удобный интерфейс для того, чтобы сделать это.

Вы можете добавить обработчик, используя метод *addInput()*:

```
@pipe = IO.popen("tail -f /var/log/system.log" )
app.addInput(@pipe, INPUT_WRITE) do
	# respond to new data in the I/O stream
	data = @pipe.read_nonblock(256)
end
```

Что бы отреагировать больше чем на один вид события для данного источника, необходимо передать некоторую комбинацию флагов: **INPUT_READ**, **INPUT_WRITE** и **INPUT_EXCEPT** как второй параметр *addInput()*. Это немного усложняет обработку, так как Вы теперь должны определить какой тип сообщения был отправлен (**SEL_IO_READ**, **SEL_IO_WRITE** или **SEL_IO_EXCEPT**):

```
app.addInput(@pipe, INPUT_WRITE|INPUT_EXCEPT) do |sender, sel, data|
	case FXSELTYPE(sel)
		when SEL_IO_WRITE:
			# something was written to the file
		when SEL_IO_EXCEPT:
			# an exception has occurred
	end
end
```

До сих пор мы сосредотачивались на сообщениях, которые **FOX** посылает Вашему приложению когда что-то происходит. В большинстве случаев это действия, которые пользователь предпринимает, такие как перемещение курсора мыши или нажатие клавиш. При некоторых особых обстоятельствах это - высокоуровневый вид событий, такой как истечение таймера или некоторые данные, записанные в файл, который Вы смотрите.

В следующих разделах мы рассмотрим другое и мощное применение этой *target* and *message-based* системы, и автоматический механизм обновления **GUI FOX**.

## 7.4 Syncing the User Interface with the Application Data

Автоматический механизм обновления **GUI FOX** - одна из самых мощных функций, но это - также одна из наиболее хитрых концепций, которой должен научиться разработчик плохо знакомый с **FOX**. Много других инструментариев **GUI**, таких как Java Swing, применяют то, что известно, как Observer pattern для синхронизации пользовательского интерфейса с данными. В соответствии с этим подходом, событие пользовательского интерфейса инициирует сообщение от виджета к модели так, чтобы модель могла обновить свое значение. Аналогично, изменение в данных модели инициируют сообщение назад к наблюдателям (представлениям) так, чтобы они могут синхронизироваться с моделью.

Мы уже обсудили, как Вы можете использовать метод *connect()*, чтобы обработать сообщения от виджетов и таким образом изменить данные модели:

http://media.pragprog.com/titles/fxruby/code/buttonexample.rb

```
activate_button = FXButton.new(p, "Activate Launch Sequence" ,
        :opts => BUTTON_NORMAL|LAYOUT_CENTER_X)
activate_button.connect(SEL_COMMAND) do |sender, sel, data|
        @controller.activate_launch_sequence
end
```

**FOX** использует другой подход, когда дело доходит до обновления **GUI** в ответ на изменения в модели. Вместо отправки сообщения к объекту модели для просмотра объектов, говорящее о том что изменение произошло, приложение периодически отправляет специальное сообщение типа **SEL_UPDATE** к каждому виджету, говоря ему обновить его состояние. Объект **GUI** может зарегистрировать свой интерес к получению запросов обновления из приложения вызывая его метод *connect()* и передавая тип сообщения **SEL_UPDATE**.

В следующем примере cancel_button включает или отключает себя согласно текущему состоянию модели:

http://media.pragprog.com/titles/fxruby/code/buttonexample.rb

```
cancel_button.connect(SEL_UPDATE) do |sender, sel, data|
        sender.enabled = @controller.launch_sequence_activated?
end
```

Другое типичное использование обновления GUI должно показать или скрыть виджет, зависящий на состоянии приложения:

```
encrypt_drives_button.connect(SEL_UPDATE) do |sender, sel, data|
        sender.visible = @edition.ultimate?
end
```

Если Вы интересуетесь другим подходом к задаче синхронизации пользовательского интерфейса и данных модели, Вы можете использовать библиотеку Joel VanderWerf's **FoxTails** library (http://redshift.sourceforge.net/foxtails/). С **FoxTails**, Вы можете идентифицировать атрибуты модели как "observable" и затем непосредственно связывают их с виджетами. Когда Вы взаимодействуете с виджетом, он автоматически обновляет значение атрибута, и наоборот. Это может быть чрезвычайно удобная альтернатива процессу установки обработчиков **SEL_COMMAND** и **SEL_UPDATE** для виджетов. Но пока мы находимся здесь, позвольте мне говорить Вам о целях данных.

## 7.5 Using Data Targets for GUI Update

В предыдущем разделе мы изучили, как синхронизировать **GUI** с данными модели, обрабатывая сообщение **SEL_UPDATE**. Типовое приложение для этого должно сохранить

установку для определенного виджета, такого как текстовое поле, в синхронизации с определенным атрибутом модели, таким как имя пользователя. Это, конечно, не трудно сделать. Мы знаем, как обработать сообщение **SEL_COMMAND** от **FXTextField**, чтобы обновить данные модели всякий раз, когда пользователь вводит новое значение в виджет:

```
user_name_textfield.connect(SEL_COMMAND) do |sender, sel, data|
	@user_name = sender.text # the FXTextField is the sender
end
```

Аналогично, мы знаем, как обработать сообщение **SEL_UPDATE**, чтобы обновить установка виджета всякий раз, когда данные модели изменяются:

```
user_name_textfield.connect(SEL_UPDATE) do |sender, sel, data|
	sender.text = @user_name
end
```

Когда мы рассматриваем этот пример в изоляции, он не походит на такое грандиозное предприятие. Но что, если имя пользователя выведено на экран в многократных расположениях на пользовательском интерфейсе? Чтобы сохранить вещи непротиворечивыми, Вы должны были бы записать обработчики **SEL_UPDATE** для каждого виджета, появление которого зависит от значения этой части данных модели. Вообще говоря, когда Вы имеете дело с большим количеством данных модели, которые могут использоваться в больше чем одном месте в **GUI**, поддержка всего этого кода, необходимого для сохранения модели и представления в согласии могут быстро выйти из-под контроля.

Класс **FXDataTarget** обеспечивает прямое решение этой проблемы. Цель данных - специальный вид объекта, который содержит несколько бит данных (как имя пользователя) и знает, как сделать правильную вещь в ответ на определенные типы сообщения, такие как **SEL_COMMAND** и **SEL_UPDATE**:

```
@user_name = FXDataTarget.new("Rollie" )
user_name_textfield = FXTextField.new(p, 20,
	:target => @user_name, :selector => FXDataTarget::ID_VALUE, ...)
```

Отметьте, что мы можем связать *@user_name* со многими виджетами в пользовательском интерфейсе, если это имеет смысл. Если Вы изменяете настройки для имени пользователя в одном из тех виджетов, все другие виджеты соединённые с data target будут обновлены. Аналогично, если Вы изменяете значение *@user_name.value*, все настройки виджетов будут обновлены, чтобы отразить новое значение.

Мы будем видеть более конкретные примеры того, как использовать **FXDataTarget** в Главе 8, *«Создание Простых Виджетов»*, на странице 100. Прежде, чем мы сделаем это, тем не менее, мы рассмотрим более внимательно оптимизацию, используемую **FOX**, которая делает его одним из самых быстрых и дружественным инструментарием **GUI**.

## 7.6 Responsive Applications with Delayed Layout and Repaint

FOX реализует различные виды оптимизации, чтобы сделать пользовательский интерфейс настолько быстрым, насколько возможно. В этом разделе мы более близко рассмотрим два способа оптимизации: задержанное расположение и задержанное перерисование.

### Delayed Layout

Расположение - процесс гарантирующий, что все виджеты в пользовательском интерфейсе помещены в надлежащие место с надлежащими размерами. **FOX** использует

технологию под названием задержанное расположение, чтобы эффективно повторно вычислить расположение пользовательского интерфейса в ответ на изменения. Когда Вы вызываете *recalc()* в окне, расположение окна отмечается как “грязное,” подразумевая, что оно должно быть обновлено, но ничего сразу не делается. Вместо этого этот вызов *recalc()* проходит через всю иерархия окон из оболочки этого окна, отмечая все промежуточные окна по пути как грязные. Оболочка тогда регистрирует *layout chore* в приложении, чтобы выполнить изменения в расположении позже, когда есть время простоя.

Большинство операций, в которых Вы хотели бы сменить расположение будет помечено как *dirty (грязный)* вызовом *recalc()* для Вас. Например, если Вы меняете текст в **FXLabel** *label* вызовет *recalc()* сама. Если Вы изменяете стиль фрейма для **FXButton** от **FRAME_RAISED** до **FRAME_SUNKEN**, кнопка вызовет *recalc()* для себя. Одно известное исключение к этому правилу имеет отношение к добавлению новых дочерних окон в родительское окно. В этой ситуации расположение родительского окна не отмечается как грязное, таким образом, это Ваше дело вызвать *recalc()* для родительского окна, чтобы гарантировать что расположение окна должным образом обновлено:

http://media.pragprog.com/titles/fxruby/code/dynamic.rb

```
FXLabel.new(contents, "Dynamically Added Field" )
FXTextField.new(contents, 20)
contents.create # create server-side resources
contents.recalc # mark parent layout as dirty
```

Так как **FOX** использует *chore*, чтобы выполнить это задержанное расположение, это означает что обновление произойдет только после того, как со всеми другими событиями ожидания покончено. Обычно это не проблема, но иногда Вы не можете ожидать так долго, и Вы должны обновить расположение сразу же. В этой ситуации, Вы можете вызвать *layout()*, чтобы немедленно обновить расположение, но убедитесь, что сделали вызов в самый верхний виджет в иерархии.

### Delayed Repaint

Тесно связанный предмет к задержанному расположению это задержанное перерисование. Так же, как постоянный перерасчет расположения пользовательского интерфейса может стать в вычислительном отношении дорог, повторенное перерисование маленьких разделов экрана может стать очень дорогим. Чтобы выполнять перекрашивание наиболее эффективно, **FOX** ставит в очереди события перекрашивания, пока нет простоя Вашего приложения. Вы можете вызвать *update()* в окне, чтобы отметить его как грязное и нуждающееся в repaint, так же как Вы вызвали бы *recalc()* в окне чье расположение изменилось. Если Вы вызываете *repaint()* в окне, все ожидающие события *repaint* для того окна сразу будут обработаны. Аналогично, если Вы вызываете *repaint()* на объекте приложения, все ожидающие *repaints* для всех окон сразу будут обработаны.

Задержанное расположение и перекрашивание - особенно хорошая оптимизация потому что по большей части Вы не должны действительно сделать многого в своих приложениях. Другая важная оптимизация в работе **FOX** - его явное различие между тем, что это называется *client-side* и *server-side* представления объектов пользовательского интерфейса. Вы будете действительно часто использовать это в своем коде. Чтобы должным образом использовать эту функцию, мы обсудим это подробно.

## 7.7 Client-Side vs. Server-Side Objects

Одна из важных вещей в **FXRuby** то, как виджеты и другие объекты пользовательского интерфейса сконструированы и созданы. Когда знающие люди говорят о клиентских

объектах и серверных ресурсах в **FOX** и **FXRuby**, они обращаются к разделению между "клиентским" пространством — то есть объекты, которые Вы инстанцируете в своих программах, которые выделены на "куче" — и пространстве "сервера" — ресурсы, которые выделены в X сервере (или в Windows GDI) соответствующие этем объектам. Терминология может немного сбить с толку, так как мы обычно используем термины клиент и сервер когда говорим о архитектуре сетевых приложений.

Рисунок 7.4, иллюстрирует жизненный цикл объектов *client-side* и их *server-side* ресурсов. Когда Вы вызываете *create()* на некоторый объект, создаётся серверный ресурс для того клиентского объекта. Вызов *destroy()* на объекте, уничтожает его серверную коллегу, но не имеет никакого эффекта на клиентский объект. Метод *detach()* является своего рода компромиссом между этими двумя; он обрывает соединение между клиентским объектом и серверным ресурсом, но фактически не уничтожают последний.

Как эта двухступенчатая конструкция и процесс создания влияют на Вас как прикладного программиста? Ну, Вы уже видели некоторое доказательство из этого, в самой первой программе мы записали:

http://media.pragprog.com/titles/fxruby/code/hello.rb

```
require 'fox16'

app = Fox::FXApp.new
main = Fox::FXMainWindow.new(app, "Hello, World!" ,
        :width => 200, :height => 100)
app.create
main.show(Fox::PLACEMENT_SCREEN)
app.run
```

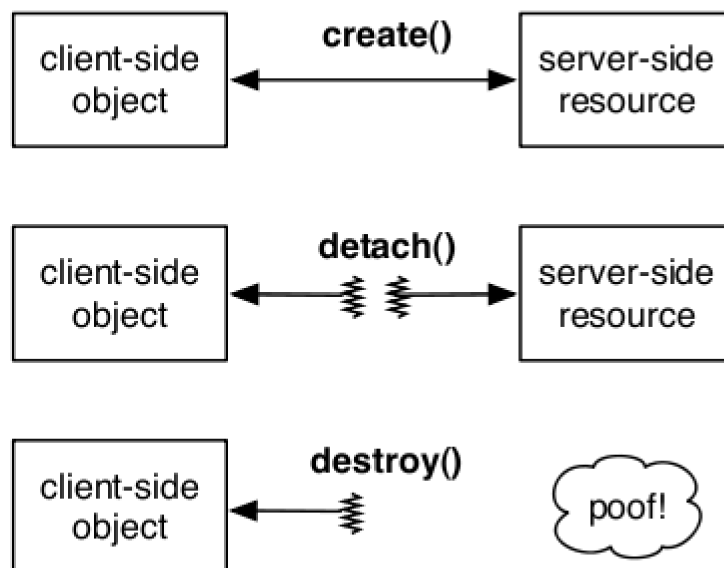



Figure 7.4: The create, detach, and destroy life cycle

Когда мы вызываем *create()* на объекте **FXApp**, приложение тогда идет через весь набор виджетов, вызывая **create()** на каждом из них и создавая server-side окно для него. Вплоть до этой точки, объект **FXMainWindow** который Вы создали на предыдущей строке, не имел серверную коллегу. Это стоит отметить, что это обсуждение применяется не только к окнам, но и ко всем видам объектов пользовательского интерфейса, включая значки, изображения, и шрифты — они все проходят через эту ту же самую конструкцию и процесс создания.

Если Ваша программа создала все свои объекты пользовательского интерфейса, это начальный вызов *create()* единственное, в чём Вы будете нуждаться. Практически,

Вы будете писать программы, которые создают новые виджеты на лету в ответ на различные события. Например, каждый раз Вы выводите на экран новое диалоговое окно в Вашей программе, Вы будете создавать набор новых виджетов. Если Вы пишете программу, которая загружает и выводит на экран данные изображений любого вида, Вы будете создавать объекты **FXImage**. **FXRuby** облегчает Вам работу по созданию всего этого, таким образом, Вы не должны волноваться о предварительном построении Вашего приложения. То, что Вы действительно должны сделать, однако, не забудьте вызывать *create()* на этих объектах, которые Вы создаете динамически, как только программа запущена и работает.

Я не могу не подчеркнуть последнюю мысль. Источник номер один ошибки в программах **FXRuby**, которые я видел (включая мое собственное) перестали работать ошибка вызова *create()* на динамически созданных объектах пользовательского интерфейса. Лучший вариант развития событий - это что пользовательский интерфейс не будет похож на то что Вы ожидали. Рассмотрите следующий пример:

```
# Add a row to the contact information form for providing
# an alternate e-mail address.
add_button.connect(SEL_COMMAND) do
	FXLabel.new(@contact_info, "Alternate e-mail address: " )
	FXTextField.new(@contact_info, 20)
end
```

В этом фрагменте кода мы пытаемся добавить новое поле к форме когда пользователь щелкает add_button. Этот код будет работать, но мы никогда не увидим новое поле на экране, потому что нет никакого серверного окна для этого:

```
# Add a row to the contact information form for providing
# an alternate e-mail address.
add_button.connect(SEL_COMMAND) do
	FXLabel.new(@contact_info, "Alternate e-mail address: " )
	FXTextField.new(@contact_info, 20)
	@contact_info.create # create dynamically constructed widgets
end
```

Добавление вызова *create()* на родительском окне решает эту проблему. Родитель будет идти через все его дочерние окна, и вызывать *create()* на них. Отметьте, что не повреждает вызов create() на объекте, который уже был создан. **FXRuby** распознает, что объект уже имеет серверный ресурс, связанный с ним и будет идти дальше к следующему объекту.

Худший вариант - то, что Ваша программа отказывает потому что например, некоторый бит кода глубоко в недрах библиотеки **FOX** попробовал потянуть шрифт, у которого не было серверного ресурса шрифта связанным с ним:

```
# Change the font for this label
new_font = FXFont.new(app, "helvetica" , 14)
label.font = new_font
```

Без вызова *create()* для *new_font*, эта программа откажет в некоторый момент после того, как Вы присвоите шрифт полю. Я говорю “некоторая точка после”, потому что эта точка немного непредсказуема — она не станет проблемой, пока **FXRuby** фактически не должен перерисовать это поле и получить доступ к его шрифту. Как в предыдущем примере, добавляя вызов *create()* для метки или для шрифта непосредственно, Вы поимеете проблемы.

## 7.8 How Windows Work

Прежде, чем мы рассмотрим специфические особенности различных виджетов в следующей главе, мы поговорим о том, что имеют все виджеты **FXRuby**, а именно, что каждый класс виджета **FXRuby** — потомок базового класса **FXWindow**. Если Вы

посмотрите документацию **API** для класса **FXWindow** Вы быстро обнаружите, что это очень сложный класс, с огромным числом атрибутов методов. Я не собираюсь охватить их все в этом разделе, но мы осветим часть из них.

**FXRuby** делает различие между родительскими и дочерними окнами. В этом случае мы не говорим об иерархии классов, где мы могли бы заметить, что класс **FXLabel** - родитель **FXButton** класс. Отношение, которое мы описываем здесь, имеет отношение включение окон в других окнах: все дочерние окна для данного родительского окна будут выведены на экран в пределах границ родителя. Вы можете видеть эту терминологию, отраженную на имена многих методов **FXWindow** и атрибутов. Например, родительский атрибут для окна возвращает ссылку на своего родителя, в то время как метод *children()* для окна возвращает массив ссылок на его дочерние окна.

FXRuby также делает различие между окном рабочего стола (корень), окном верхнего уровня (top-level or shell), и другими видами окон. Окна Shell прямые дочерние элементы корневого окна, и они всегда — экземпляры **FXShell** или один из его подклассов. Так, например, основное окно Вашего приложения (экземпляр **FXMainWindow**) является окном оболочки (shell), как любое диалоговое окно, которые выводит на экран Ваше приложение.

Иногда, Вы должны будете иметь дело с системами координат окна. В Разделе 7.2, *«Обработка событий мыши»*, на странице 85, мы говорили о том, как данные события для связанных с мышью сообщений включают информацию о том в каком месте на экране событие имело место в локальных координатах и корневых координатах окна. В отличие от декартовой системы координат, здесь система координат начинается с (0, 0) в верхнем левом углу окна. X координата увеличивается при перемещении направо, и Y координата увеличивается при движении вниз.

Методы *translateCoordinatesFrom()* и *translateCoordinatesTo()* могут использоваться для преобразования между различными системами координат оконной системы. Предположите, например, что Вы имеете дело с событием **SEL_MOTION**, которое произошло в окне A, но мы хотели бы знать расположение курсора мыши с точки зрения системы координат окна B:

```
window_a.connect(SEL_MOTION) do |sender, selector, event|
	b_x, b_y =
		sender.translateCoordinatesTo(window_b, event.win_x, event.win_y)
end
```

Поскольку событие имело место в окне A, значения *event.win_x* и *event.win_y* рассматриваются с точки зрения системы координат A. Вызов *translateCoordinatesTo()* возвращает массив, содержащий координаты в система координат окна Б.

Мы рассмотрели много сложного материала в этой главе, и к этому материалу Вы будете повторно возвращаться, когда начнёте писать свои собственные приложения. Понимая как реализована модель событийно-управляемого программирования **FOX**, Вы может написать намного больше тесно интегрированных приложений, которые требуют меньше строк кода, чтобы скрепить виджеты к данными, которые они представляют. Знание работы алгоритмов задержанного размещения и repaint поможет Вам избежать некоторых из ловушек с которыми сталкиваются менее опытные разработчики.

С этими знаниями , мы готовы идти дальше к изучению стандартных виджетов, таких как *labels*, кнопки, и текстовые поля, которые использует почти каждое приложение **FXRuby**.

## Глава 8

# Создание Простых Виджетов

Виджеты - стандартные блоки приложений **GUI**. Это объекты специального назначения, которые выводятся на экран и осуществляют коммуникацию между пользователями и программным обеспечением. Если Вы подобно большинству людей, работаете с программным обеспечением которое включает некоторый графический интерфейс пользователя, Вы постоянно используете виджеты, даже если делаете это бессознательно. Когда Вы щелкаете по выпадающему меню в Вашем текстовом процессоре и выбирает команду из того меню, Вы взаимодействуете с приложением через виджеты. Когда Вы захватываете прокрутку панели на правой стороне документа и прокручиваете его назад или вперед, Вы снова используете виджет, чтобы взаимодействовать с программным обеспечением.

В его книге «About Face» [Coo95] Алан Купер говорит об основных четырех типах виджетов, с точки зрения пользователей:

- Обязательные виджеты, которые используются, чтобы инициировать функцию
- Виджеты выбора, используемые, чтобы выбрать опции или данные
- Виджеты записи, используемые, чтобы ввести данные
- Виджеты дисплея, используемые, чтобы непосредственно управлять программой визуально

Некоторые виджеты могут, комбинировать эти типы. Рисунок 8.1, содержит список виджетов, которые мы рассмотрим в этой главе, наряду с советом о том, когда использовать их. У нас нет достаточного места в этой книге, чтобы описать все виджеты **FXRuby**, но если Вы изучите некоторые из обычно используемых виджетов, то Вы поймёте терминологию и соглашения о присвоении имен, которые используются повсюду. После того, как Вы закончите читать эту главу и следующие несколько глав и у Вас появятся навыки, Вы сможете интегрировать другие более специализированные виджеты в Ваше приложение без проблем.

| **Widget Class** | **What's It For?** |
|---|---|
| FXLabel | Текстовая метка на экране с дополнительным значком, для декоративных или информативных целей. |
| FXButton | Кнопка в качестве "нажимаемого" интерфейса к команде. |
| FXRadioButton | Группа переключателей для выбора одного из многих возможных вариантов. |
| FXCheckButton | Кнопка проверки, чтобы позволить пользователю выбирать или отменять выбор опция. |
| FXTextField | Текстовое поле, чтобы позволить пользователю редактировать одну строку текста. |
| FXToolTip | Подсказка, чтобы вывести на экран временное, информативное сообщение о цели некоторого виджета. |
| FXStatusBar | Строка состояния, чтобы вывести на экран детализированную, контекстно-зависимую справку о некотором другом виджете или состояние приложения. |

Figure 8.1: Simple Widgets

## 8.1 Creating Labels and Buttons

Я не думаю, что когда-либо разрабатывал приложение **GUI**, которое не имело бы некоторое количество меток и кнопок. Они легко понятны и просты в использовании, таким образом, это хорошая практика для начала.

## Displaying Text with Labels

Мы можем использовать виджет **FXLabel**, чтобы вывести сообщение на пользовательском интерфейсе. Оно может быть простым, таким как строка заголовка для функции в пользовательском интерфейсе, или более сложным, таким как ряд инструкций для некоторой задачи. Текст метки может состоять из одной или более строк, разделенных символами новой строки, и метка может дополнительно вывести на экран значок. С технической точки зрения, текст для метки опционален, и возможно вывести на экран виджет «label», у которого есть только значок, но эта практика, более характерна для виджетов кнопок (который мы рассмотрим в следующем разделе).

По умолчанию текст метки центрируется (и горизонтально и вертикально) в ограничивающем прямоугольнике метки. Однако, метка различные опции выравнивания, которые мы можем использовать, чтобы определить как текст метки выравнивается. Например, чтобы выровнять текст метки по левой стороне, передайте опцию **JUSTIFY_LEFT** конструктору **FXLabel**.

http://media.pragprog.com/titles/fxruby/code/labelexample1.rb

```
label = FXLabel.new(self, "Left-justified text" , :opts => JUSTIFY_LEFT)
```

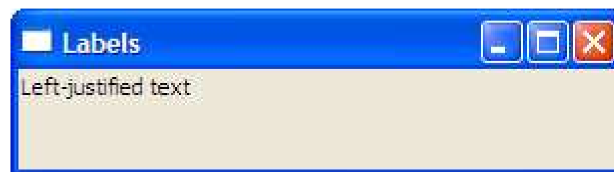



Figure 8.2: Label displaying left-justified text

Рисунок 8.2 показывает, как текст этой метки выведен на экран выровненный по левой стороне. Вы можете также измените текстовое выравнивание для метки, устанавливая непосредственно атрибут.

http://media.pragprog.com/titles/fxruby/code/labelexample2.rb

```
label.justify = JUSTIFY_RIGHT|JUSTIFY_BOTTOM
```

Рисунок 8.3 показывает текст метки, выровненный по правой и нижней части ограничивающего прямоугольника метки. Для перечисления опций текстового выравнивания см. документацию API для класса **FXLabel**.

Figure 8.3: Label displaying bottom- and right-justified text

По умолчанию метка рисуется без визуального отражения фрейма вокруг неё. Стиль фрейма может быть изменен комбинацией флагов стиля фрейма в конструкторе **FXLabel**. Например, чтобы создать метку с сплошной линией вокруг ее границ, используйте стиль фрейма **FRAME_LINE**.

http://media.pragprog.com/titles/fxruby/code/labelexample3.rb

```
line_frame = FXLabel.new(p, "Line Frame" , :opts => FRAME_LINE)
```

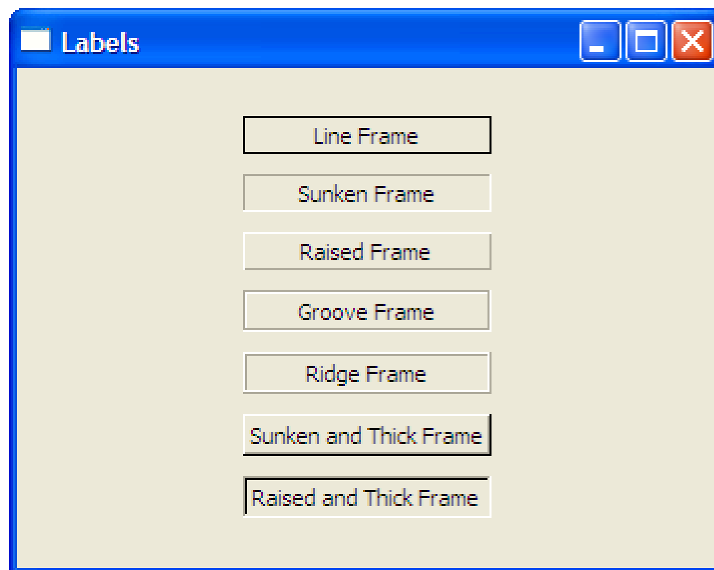



Figure 8.4: Labels displaying the various frame styles

Вы можете также изменить стиль фрейма, устанавливая атрибут **frameStyle** метки после того, как она была создана. В этом примере мы показываем *sunken* фрейм.

http://media.pragprog.com/titles/fxruby/code/labelexample3.rb

```
sunken_framed_label = FXLabel.new(p, "Sunken Frame" )
sunken_framed_label.frameStyle = FRAME_SUNKEN
```

Рисунок 8.4 показывает примеры меток со всеми поддерживаемыми стилями фрейма. Для полного списка доступных стилей фрейма см. документацию API для класса **FXWindow**.

*Константы стиля фрейма связаны с базовым классом FXWindow потому что много различных видов виджетов наследуют эти стили, не только labels.*

Как упоминалось ранее Вы можете также включать значок в метку. Например, Вы можете взять значок из файла формата GIF и затем передайте его как параметр конструктору **FXLabel**.

http://media.pragprog.com/titles/fxruby/code/labelexample4.rb

```
question_icon =
        FXGIFIcon.new(app, File.open("question.gif" , "rb" ).read)
question_label =
        FXLabel.new(self, "Is it safe?" , :icon => question_icon)
```

Figure 8.5: A label with an icon

Мы рассмотрим более подробно как создавать значки в Главе 11, *«Создание Визуально Богатых Пользовательских интерфейсов»*, на странице 142. Пока, это достаточно, чтобы знать, что Вы можете создать объект значка изображения из

файла (или некоторого другого источника) и затем использовать его в качестве художественного оформления для меток, кнопок, и других видов виджетов.

По умолчанию значок будет размещён по центру метки. Это может сделать текст метки трудно читаемым, если значок не прозрачный или есть высокий контраст между цветами значка и цветом текста. Метка поддерживает много опций, которые определяют, как текст и значок расположены относительно друг друга. Например, чтобы поместить значок перед текстом (другими словами, с левой стороны от него), используйте опцию **ICON_BEFORE_TEXT**.

http://media.pragprog.com/titles/fxruby/code/labelexample4.rb

```
question_label.iconPosition = ICON_BEFORE_TEXT
```

Рисунок 8.5 показывает, как этот пример выглядит под Windows. Снова, сошлитесь на документацию API для класса **FXLabel** для полного перечисление доступных настроек позиции значка.

## Getting Pushy with Buttons

**FXButton** - это действие для метки в том смысле, что он может быть "нажат" и это выполняет некоторую команду в Вашей программе. Как метка, кнопка может вывести на экран строку сообщения и значок. В отличие метки, у кнопки обычно есть 3-D "выпуклое" отображение, которое позволяет ей выделиться. Рисунок 8.6 показывает две различных кнопки, один включена а другая отключена.

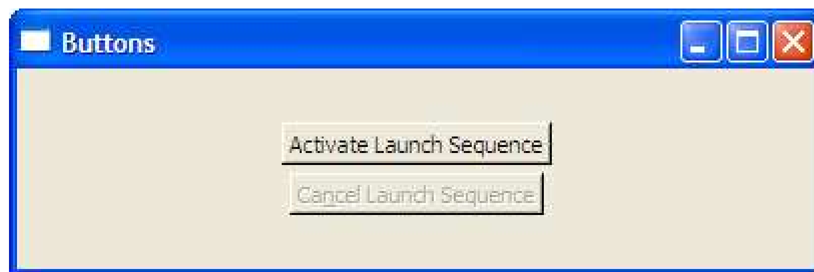



Figure 8.6: Unlike labels, buttons look “pressable.”

Когда кнопка нажата и отпущена, она отправляет сообщение **SEL_COMMAND** к цели:

http://media.pragprog.com/titles/fxruby/code/buttonexample.rb

```
activate_button = FXButton.new(p, "Activate Launch Sequence" ,
        :opts => BUTTON_NORMAL|LAYOUT_CENTER_X)
activate_button.connect(SEL_COMMAND) do |sender, sel, data|
        @controller.activate_launch_sequence
end
```

Кнопка также отправит несколько других сообщений к своей цели, но эти сообщения редко полезны практически. Если Вы интересуетесь ими, помните, что Вы можете всегда ссылаться на онлайновую документацию API для класса, чтобы узнать обо всех сообщениях.

Вы можете связать акселераторы и горячие клавиши с кнопками и другими видами виджетов. Акселератор - комбинация нажатия клавиш, которое вызывает действие в Вашем приложении. Например, нажатии **Ctrl+C** вызовет действие *Copy* во многих приложениях. Горячая клавиша - специальный вид акселератора, который является комбинацией клавиши **Alt** и буквы, такой как **Alt+F**, чтобы открыть меню *File*.

Горячие клавиши обычно связаны с кнопками, или подобными кнопке виджетами, которые имеют связанную текстовую строку. Вы кодируете горячую клавишу для кнопки, делая это:

http://media.pragprog.com/titles/fxruby/code/buttonexample.rb

```
cancel_button = FXButton.new(p, "Ca&ncel Launch Sequence" ,
        :opts => BUTTON_NORMAL|LAYOUT_CENTER_X)
```

В этом примере, символ амперсанда, который предшествует букве n в метке кнопки указывает, что нажатие клавиш **Alt+N** должено инициировать команду этой кнопки, так же, как если бы пользователь нажал эту кнопку с мышью.

Мы говорили в предыдущей главе о том, как использовать автоматическое обновление **GUI**, чтобы обновить состояние виджетов пользовательского интерфейса в зависимости от состояния приложения. Есть довольно общая ситуация в котором Вы могли бы хотеть использовать это, чтобы изменить состояние кнопки, и это - когда Вы хотите отключить кнопку, потому что команда связанная с той кнопкой в настоящий момент не доступна. Например, мы должны отключите кнопку *Cancel* Launch Sequence если последовательность запуска не была активирована. Вы можете записать обработчик **SEL_UPDATE**, чтобы сделать это:

http://media.pragprog.com/titles/fxruby/code/buttonexample.rb

```
cancel_button.connect(SEL_UPDATE) do |sender, sel, data|
        sender.enabled = @controller.launch_sequence_activated?
end
```

Отметьте, что в этом блоке, отправитель - ссылка на *cancel_button* виджет, так как это - отправитель сообщения **SEL_UPDATE**.

## Making Choices with Radio Buttons

Когда Вы нуждаетесь в выборе значения из группы взаимно исключающих опций, Вы должны рассмотреть использование группы виджетов **FXRadioButton**. Radio buttons - разумный выбор когда число опций фиксировано и небольшое. Если число вариантов неизвестно до времени выполнения, Вы - вероятно, используете различные объекты интерфейса пользователя, такие как список или combo box, которое разработаны, чтобы разместить произвольное число элементов. Вы будете также использовать подобные списку виджеты, если Вы даёте пользователю больше чем несколько вариантов, потому что длинный столбец переключателей является визуально нелепым.

Давайте напишем короткий пример программы, чтобы продемонстрировать всё, что Вы должны знать при работе с переключателями. Во-первых, это - хорошая практика, чтобы использовать виджет **FXGroupBox** для отображения групповых переключателей. Вы можете использовать опцию **FRAME_GROOVE** или **FRAME_RIDGE** с group box, чтобы влиять на стиль схемы, и Вы можете дополнительно присвоить заголовок групповому блоку:

http://media.pragprog.com/titles/fxruby/code/radiobuttons.rb

```
groupbox = FXGroupBox.new(self, "Options" ,
:opts => GROUPBOX_NORMAL|FRAME_GROOVE|LAYOUT_FILL_X|LAYOUT_FILL_Y)
```

Figure 8.7: Use radio buttons for mutually exclusive choices.

Теперь Вы можете добавить несколько переключателей, чтобы представить различные опции:

http://media.pragprog.com/titles/fxruby/code/radiobuttons1.rb

```
@radio1 = FXRadioButton.new(groupbox, "Good Enough" )
@radio2 = FXRadioButton.new(groupbox, "Perfect" )
```

Давайте добавим переменную экземпляра, чтобы запомнить индекс в настоящий момент установленного переключателя и удостовериться что первая радио кнопка нажата по умолчанию:

http://media.pragprog.com/titles/fxruby/code/radiobuttons1.rb

```
@choice = 0
@radio1.checkState = true
```

Теперь, так как все еще нет никакой ссылки между значением @choice и переключателями, давайте соединим каждый из переключателей к блоку который обновит @choice всякий раз, когда один из переключателей установлен:

http://media.pragprog.com/titles/fxruby/code/radiobuttons1.rb

```
@radio1.connect(SEL_COMMAND) { @choice = 0 }
@radio2.connect(SEL_COMMAND) { @choice = 1 }
```

Рисунок 8.7 показывает, на что этот пример похож в Windows. Если Вы выполняете программу в этой точке и начинаете выбирать две радио кнопки, Вы увидите довольно серьёзную проблему. Оказывается групповой блок не делает ничего, чтобы осуществить взаимную исключительность переключателей — это только оформление витрины. Гарантировать что только один из переключателей в группе выбран за один раз должны Вы сами в Вашей программе.

Прямой способ сделать это - соединить каждую радио кнопку в группе к блоку, который обновляет состояние радио кнопок на основании на значений @choice:

http://media.pragprog.com/titles/fxruby/code/radiobuttons2.rb

```
@radio1.connect(SEL_UPDATE) { @radio1.checkState = (@choice == 0) }
@radio2.connect(SEL_UPDATE) { @radio2.checkState = (@choice == 1) }
```

Если Вы выполняете программу снова, Вы должны найти, что она ведет себя лучше в этой точке. Когда Вы выбираете одну радио кнопку, другие кнопки не выбраны, и наоборот. Несмотря на этот успех, мы можем предположить, что этот подход не будет масштабироваться хорошо для большего числа вариантов. Код, требуемый для управления состояния кнопок привёл бы к большому количеству кода.

Более изящный способ решить эту проблему состоит в том, чтобы создать экземпляр FXDataTarget, для хранения номера выбранного варианта:

http://media.pragprog.com/titles/fxruby/code/radiobuttons3.rb

```
@choice = FXDataTarget.new(0)
```

Теперь мы можем связать эти данные для каждой радио кнопки в группе:

http://media.pragprog.com/titles/fxruby/code/radiobuttons3.rb

```
radio1 = FXRadioButton.new(groupbox, "Good Enough" ,
        :target => @choice, :selector => FXDataTarget::ID_OPTION)
radio2 = FXRadioButton.new(groupbox, "Perfect" ,
        :target => @choice, :selector => FXDataTarget::ID_OPTION+1)
```

С этим изменением, мы можем избавиться от всех вызовов *connect()* для переключателей. Хотя это намного проще чем наша предыдущая попытка, есть небольшое препятствие. Проблема в том, что мы не имеем способа знать, когда значение @choice фактически изменяется. Это могло бы быть существенной проблемой, если мы хотим изменить некоторую другую часть GUI всякий раз, когда сделан различный выбор. К счастью, **FXDataTarget** подобно любому другому объекту **FOX**, может соединяться с его обработчик команд:

```
@choice.connect(SEL_COMMAND) do
	puts "The newly selected value is #{@choice.value}"
end
```

С этим заключительным изменением мы извлекаем всю пользу из использования **FXDataTarget** как шлюза для любых изменений к @choice, когда эти изменения имеют место. Теперь мы идем в коммутаторы и смотрим на различные подходы для пользовательских выборов, и это с виджетом **FXCheckButton**.

## Check Buttons: Yes? No? Maybe?

Когда Вы имеете дело с настройками приложения, которые могут иметь одно из двух возможных состояний, виджет **FXCheckButton** - вероятно, Ваш лучший выбор представить эту возможность.

Позвольте мне быть более определенным: это - вероятно, Ваш лучший выбор когда те два возможных состояния - противоположны друг другу, как "on" или "off". Даже при том, что наш пример переключателей из предыдущего раздела предлагает пользователю только два варианта, проверка кнопок не была бы лишней.

При проверке кнопки могут фактически быть в одном из трех возможных состояний. В дополнению к "checked" и "unchecke" кнопка проверки может быть в неопределенном, или "возможном", состоянии. Я встречал эту возможность в различных случаях. Например, предположите что у Вас есть кнопка проверки, которую Вы используете чтобы указать, будут ли выбранные статьи в новостном канале прочтены. Если они будут считаны, кнопка проверки должна быть в состоянии checked и если ни одна из них не будет считана, кнопка проверки, должна быть в состоянии unchecked. Но что, если некоторые из выбранных статей могут быть прочитаны а другие нет? В этой ситуации Вы могли бы хотеть установить кнопку в неопределенное состояние. Когда кнопка проверки находится в этом состояние, это будет показано как checked, но будет иметь недоступный фон (см. рисунок 8.8 на следующей странице). Когда пользователь щелкает по кнопке в неопределенном состоянии, её состояние изменится на unchecked, и в этой точке Вы вернулись к основному проверенному/непроверенному переключению. Нет никакого способа для пользователя нажать кнопку проверки назад в неопределенное состояние.

Класс **FXCheckButton** использует значения *true*, *false* и *MAYBE*, соответственно, для представления состояний *checked*, *unchecked* и *indeterminate*. Вы можете установить состояние кнопок через атрибут **checkState** и проверить их состояние, используя запросы *checked?()*, *unchecked?()* и *maybe?()*:

```
checkbutton = FXCheckButton.new(...)
checkbutton.checkState = true
checkbutton.checked? # returns true
checkbutton.unchecked? # returns false
checkbutton.maybe? # returns false
```

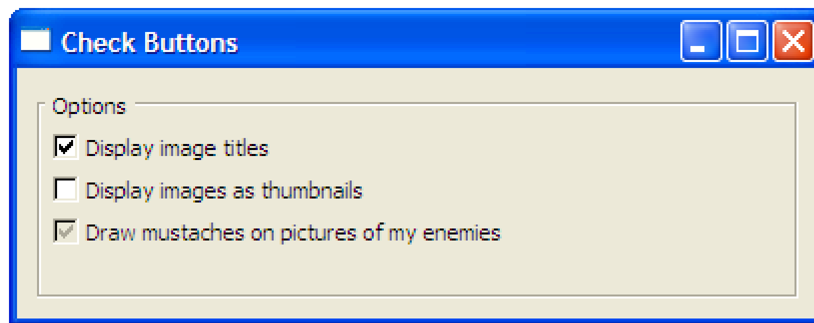



Figure 8.8: Check Buttons in the checked, unchecked, and maybe states

Как это имело место с **FXRadioButton**, часто удобно to connect a check button к цели данных:

http://media.pragprog.com/titles/fxruby/code/checkbutton.rb

```
@titles = FXDataTarget.new(true)
@thumbnails = FXDataTarget.new(false)
@mustaches = FXDataTarget.new(MAYBE)
groupbox = FXGroupBox.new(self, "Options" ,
        :opts => GROUPBOX_NORMAL|FRAME_GROOVE|LAYOUT_FILL)
titles_check = FXCheckButton.new(groupbox,
        "Display image titles" ,
        :target => @titles, :selector => FXDataTarget::ID_VALUE)
thumbnails_check = FXCheckButton.new(groupbox,
        "Display images as thumbnails" ,
        :target => @thumbnails, :selector => FXDataTarget::ID_VALUE)
mustaches_check = FXCheckButton.new(groupbox,
        "Draw mustaches on pictures of my enemies" ,
        :target => @mustaches, :selector => FXDataTarget::ID_VALUE)
@titles.connect(SEL_COMMAND) do
        puts "The new value for 'titles' is #{@titles.value}"
end
```

В этой программе мы создаем отдельный объект **FXDataTarget** для каждой настройки и затем связываем эте цели данных с соответствующей кнопкой. Так как @mustaches инициализирован к значению *MAYBE*, третья кнопка проверки (mustaches_check) запустится в неопределенном состоянии. Мы можем также соединить каждую из целей данных к блоку кода для уведомления, когда их значения изменяются.

Виджеты **FXRadioButton** и **FXCheckButton** - позволяют пользователю делать выбор из фиксированного набора вариантов, и они самые простые инструменты, которые **FXRuby** дает Вам для того, чтобы обеспечить такую функциональность. Естественно, **FXRuby** обеспечивает другие виды виджетов которые не ограничивают ввод пользователя фиксированным набором вариантов, и в следующем разделе мы рассмотрим один такой виджет, **FXTextField**.

## 8.2 Editing String Data with Text Fields

Виджет **FXTextField** является соответствующим, когда Вы должны предусмотреть ввод и последующее редактирование однострочных текстовых строк. Рисунок 8.9 показывает несколько текстовых полей в примере программы, которую мы будем использовать в этом разделе. Для того, чтобы работать с многострочным текстом, Вы должны смотреть Главу 10, *«Редактирование текста с текстовым Виджетом»*, на странице 133.

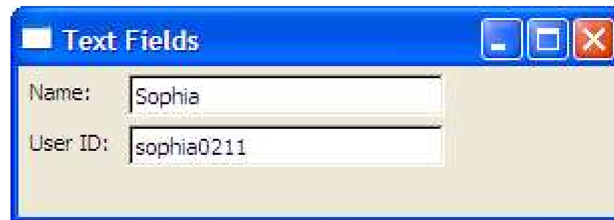



Figure 8.9: Use text fields to edit single lines of text.

В большинстве случаев, Вы хотите обработать сообщение **SEL_COMMAND** от текстового поля. **FXTextField** отправляет сообщение **SEL_COMMAND** в его цель, когда пользователь нажимает клавишу *Return* (или *Enter*) после ввода некоторого текста в текстовом поле. Это также отправит сообщение **SEL_COMMAND** когда текстовое поле теряет клавиатурный фокус (потому что пользователь щелкнул где-то в другом месте или нажал клавишу *Tab*, чтобы сместить фокус некоторому другому виджету).

*Вы можете переопределить это поведение, передавая флаг **TEXTFIELD_ENTER_ONLY** когда Вы создаёте **FXTextField**.*

Для некоторых приложений Вы хотели бы ограничить виды текста который может быть введён в текстовое поле. **FXTextField** поддерживает различные режимы ввода для некоторых общих случаев. Например, когда Вы используете текстовое поле, чтобы записать пароль, это - установившаяся практика, чтобы замаскировать текст пароля, используя звездочки. Вы можете сделать это, передавая флаг **TEXTFIELD_PASSWD** когда Вы создайте текстовое поле. Аналогично, Вы можете ограничить ввод в поле целым числом или значением с плавающей точкой, используя режимы **TEXTFIELD_INTEGER** и **TEXTFIELD_REAL**.

Если Ваше приложение нуждается в еще более сложных ограничениях на вводимый текст, Вы можете обработать сообщение **SEL_VERIFY**, которое текстовое поле передаёт к его цели. Это сообщение подобно **SEL_CHANGED**, но с важным различием: сообщение **SEL_VERIFY** отправляется предварительно перед тем как изменения "*committed*". Например, если мы хотим проверить, что текст начинается с буквы и состоит только из букв и числа, мы могли бы сделать что-то вроде этого:

http://media.pragprog.com/titles/fxruby/code/textfield.rb

```
userid_text.connect(SEL_VERIFY) do |sender, sel, tentative|
	if tentative =~ /^[a-zA-Z][a-zA-Z0-9]*$/
		false
	else
		true
	end
end
```

Отметьте что, если текст не соответствует ожидаемому образцу, блок возвращает *true*. Это кажется немного парадоксальным, но это - наш способ сообщения **FOX**, что сообщение **SEL_VERIFY** было "обработано" и что не надо ничего делать дальше. Если текст соответствует и блок возвращает *false*, **FOX** продолжит и обновит содержание текстового поля.

Наконец, как с другими виджетами, о которых мы говорили в этой главе, Вы можете связать текстовое поле с целью данных:

http://media.pragprog.com/titles/fxruby/code/textfield.rb

```
@name_target = FXDataTarget.new("Sophia" )
name_text = FXTextField.new(p, 25,
	:target => @name_target, :selector => FXDataTarget::ID_VALUE)
@name_target.connect(SEL_COMMAND) do
	puts "The name is #{@name_target.value}"
end
```

Если Вы не требуете обработки **SEL_VERIFY**, это является самым удобным способ работать с текстовым полем. Как показано Вы можете также соединить цель данных с некоторым нисходящим целевым объектом, если Вы хотите быть уведомленными относительно изменения к целевому значению данных.

Мы собираемся завершить эту главу, рассмотрев как Вы можете создавать сообщения подсказки и строку состояния. Хотя они оба технически только другой вид виджета, мало чем отличаясь метки, они необычны в том смысле, что они всегда работают в соединении с другими виджетами, чтобы предоставить своего рода высокоуровневую услугу для приложения.

## 8.3 Providing Hints with Tooltips and the Status Bar

Подсказка - специальный вид всплывающего окна, которое знает, что показать непосредственно всякий раз, когда курсор мыши останавливается в определенном месте на несколько секунд. *Tooltip* спрашивает виджет, на который указывает курсор мыши, текст подсказки и выводит на экран тот текст. *Tooltip* выведет на экран этот текст в течение короткого времени, и затем исчезнет; он также скроется, как только Вы переместите курсор мыши в новое расположение.

FXRuby облегчает добавление подсказки к Вашему приложению. Во-первых, Вы должны создать объект **FXToolTip**:

http://media.pragprog.com/titles/fxruby/code/tooltipexample.rb

```
FXToolTip.new(app)
```

Отметьте, что есть только один объект подсказки для всего приложения, что может казаться немного парадоксальным, так как Вы будете видеть, что подсказка раскрывается во всех местах!

Затем, Вы должны определить текст подсказки, который должен быть выведен на экран для каждого виджета, так как текст подсказки по умолчанию для виджета пуст. Для виджетов **FXButton** и других виджетов, которые произошли из **FXButton**, Вы можете встроить текст подсказки непосредственно в название кнопки, когда Вы создаете кнопку:

http://media.pragprog.com/titles/fxruby/code/tooltipexample.rb

```
upload_button = FXButton.new(self, "Upload\tUpload Files" )
```

Отметьте, что текст подсказки разделен от метки кнопки символом табуляции.

Много других виджетов, также позволяют устанавливать их текст подсказки, используя атрибут *tipText*:

http://media.pragprog.com/titles/fxruby/code/tooltipexample.rb

```
dial = FXDial.new(self, :opts => DIAL_HORIZONTAL)
dial.range = 0..11
dial.tipText = "Volume"
```

Подобно *tooltip*, виджет **FXStatusBar** способен к отображению контекстно-зависимого сообщения о виджете, когда курсор мыши перемещается по этому виджету. В отличие от подсказки, строка состояния - постоянное приспособление на главном окне Вашего приложения – она не просто раскрывается на короткое и затем исчезает снова как подсказка. Традиционно, строка состояния помещена вдоль нижнего края главного окна и имеет ширину главного окна, но это не обязательно если Ваше приложение имеет некоторые другие потребности расположения.

Для виджетов **FXButton** и других, которые произошли из **FXButton**, Вы можете определите сообщение справки для виджета непосредственно в метке кнопки:

http://media.pragprog.com/titles/fxruby/code/tooltipexample.rb

```
download_button = FXButton.new(self,
"Download\tDownload Files\tStart Downloading Files in the Background" )
```

Отметьте, что текст справки строки состояния разделен от текста подсказки вторым символом табуляции. Вы можете использовать атрибут helpText, чтобы определить текст справки строки состояния для других виджетов.

```
dial.helpText = "This one goes to eleven"
```

# Глава 9

# Sorting Data with List and Table Widgets

Простые виджеты, о которых мы узнали в предыдущей главе предназначены для единственного значения (если у них есть какое-либо реальное "значение", связанное с ними вообще). FXRuby также обеспечивает много более сложных виджетов для того, чтобы иметь дело с наборами значений. Рисунок 9.1, перечисляет виджеты, которые мы будем рассматривать в этой главе, наряду с кратким описанием того, когда Вы хотели бы рассмотреть использование их в Вашем приложения. Мы начнем с FXList.

| Widget Class | What's It For? |
|---|---|
| FXList | FXList выводит на экран всегда видимый, плоский список элементов и позволяет пользователю выбирать один или более элементов из него. |
| FXListBox | FXListBox выводит на экран выпадающий, плоский список элементов и позволяет пользователю выбирать единственный элемент из него. |
| FXComboBox | FXComboBox выводит на экран выпадающий, плоский список элементов и позволяют пользователю выбирать единственный элемент из него. В отличие от FXListBox, FXComboBox доступен для редактирования. |
| FXTreeList | FXTreeList выводит на экран список иерархически структурированных элементов и позволяет пользователю выбирать один или больше элементов из него. |
| FXTable | FXTable выводит на экран набор элементов в табличном виде и позволяет пользователю выбирать один или более элементов из него. |

Figure 9.1: List Widgets

## 9.1 Displaying Simple Lists with FXList

Виджет **FXList** выводит на экран список элементов, где у каждого элемента есть связанная текстовая строка и дополнительный значок. Если список содержит больше элементов чем он может вывести на экран, появится вертикальная полоса прокрутки, чтобы позволить Вам прокручивать список.

По умолчанию **FXList** пуст. Вы можете добавить элементы до конца списка используя метод *appendItem()*:

http://media.pragprog.com/titles/fxruby/code/listexample.rb

```
groceries = FXList.new(self,
        :opts => LIST_EXTENDEDSELECT|LAYOUT_FILL)
groceries.appendItem("Milk" )
groceries.appendItem("Eggs" )
groceries.appendItem("Bacon (Chunky)")
```

Вы можете также добавить элемент к началу списка, вставить элемент в определенную позицию в списке, или удалить элемент из списка (используя *prependItem()*, *insertItem()* или *removeItem()* методы, соответственно):

http://media.pragprog.com/titles/fxruby/code/listexample.rb

```
groceries.prependItem("Bread" )
groceries.insertItem(2, "Peanut Butter" )
groceries.removeItem(3)
```

## Making Selections in Lists

**FXRuby** поддерживает несколько атрибутов, имеющих отношение к выбору в списке. Текущий элемент - просто последний элемент списка на который Вы щелкнули, это элемент, у которого в настоящий момент есть клавиатурный фокус. Если нет выбранного элемента, **currentItem** для списка равен -1; иначе, это - целое число, индекс текущего элемента. Когда текущий элемент изменяется, **FXList** отправляет и сообщение **SEL_CHANGED** и сообщение **SEL_COMMAND** к виджету списка:

```
groceries.connect(SEL_COMMAND) do |sender, sel, index|
	puts "The new current item is #{sender.currentItem}"
end
```

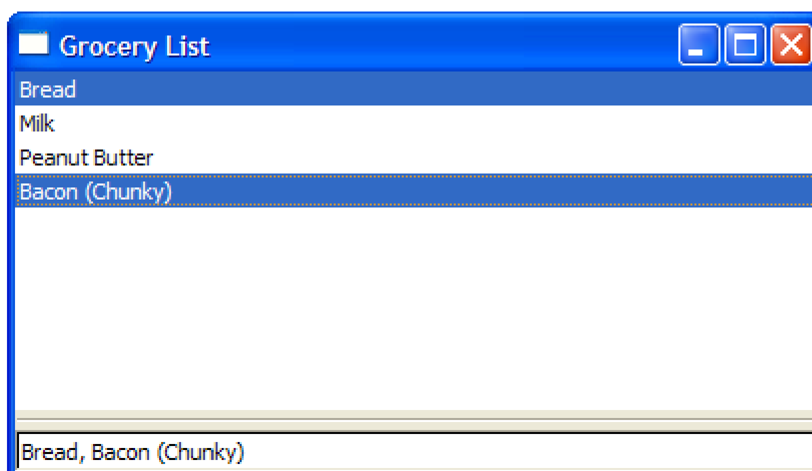



Figure 9.2: FXList in action

Список отправляет много других интересных сообщений своей цели когда, например, пользователь дважды щелкает по элементу списка. Для полного перечисление всех сообщений, что **FXList** передает к его цели, смотрите **API** документацию.

Режим выбора для **FXList** устанавливает политику для того, сколько элементов могут быть выбраны одновременно и как Вы изменяете выбор. Один режимов выбора, который Вы будете использовать часто, является **LIST_BROWSESELECT**. В этом режиме есть всегда только один выбранный элемент списка, последний на который Вы щелкнули. Другой обычно используемый режим выбора списка **LIST_EXTENDEDSELECT**. В этом режиме любое число элементов может быть выбрано. *Ctrl +clicking* элемент переключает свое выбранное состояние, и содержание вниз клавиша *Shift* при щелчке по элементам расширяет текущий выбор включая все промежуточные элементы.

***FXList** также обеспечивает менее часто используемые **LIST_SINGLESELECT**, **LIST_AUTOSELECT** и **LIST_MULTIPLESELECT** режимы выбора.*

## Какие Элементы Выбраны?

Когда список сконфигурирован в **LIST_SINGLESELECT**, **LIST_BROWSESELECT** или режим **LIST_AUTOSELECT**, Вы можете безопасно предположить что **currentItem** это в настоящий момент выбранный элемент. Когда список сконфигурирован в **LIST_EXTENDEDSELECT** или режим **LIST_MULTIPLESELECT**, Вы нуждаетесь в проверке каждого элемента списка индивидуально, чтобы узнать, выбран ли он.

Один способ сделать это - выполнить итерации по всем индексам элементов:

```
selected_indices = []
0.upto(list.numItems-1) do |index|
	selected_indices << index if list.itemSelected?(index)
end
```

Другой подход - выполнить итерации по экземплярам **FXListItem** непосредственно, тестирование их состояния **selected?()**:

http://media.pragprog.com/titles/fxruby/code/listexample.rb

```
selected_items = []
groceries.each { |item| selected_items << item if item.selected? }
```

Как Вы могли бы ожидать, **FXList** и **FXListItem** обеспечивают несколько дополнительных методов, имеющих отношение к поведению и появлению списка. Для всех деталей см. документацию **API** для этих классов.

В зависимости от числа элементов в списке и доступном “real estate” в Вашем пользовательском интерфейсе, **FXList** может не быть лучшим выбором для отображения набора данных. Если Вы должны вывести на экран длинный список элементов но имейте только небольшое количество пространства, комбинированный список мог бы работать лучше. Мы рассмотреть этот виджет дальше.

## 9.2 Good Things Come in Small Packages: FXComboBox and FXListBox

Виджеты **FXComboBox** и **FXListBox** - оба вариации виджета **FXList**. Оба из них похожи на комбинацию **FXTextField** и **FXArrowButton**. Когда Вы нажимаете кнопку стрелки, текстовое поле расширяется, чтобы вывести на экран весь список элементов. После того, как Вы выбираете элемент из список, список исчезает чтобы принять исходное положение. Как **FXList**, они могут оба использоваться, чтобы вывести на экран плоский список элементов в котором пользователь может выбрать элемент. В отличие от **FXList**, они позволяют Вам выбирать только один элемент за один раз.

Нет никаких твердых правил о том, когда предпочтительно использовать регулярный **FXList** в противоположность **FXComboBox** или **FXListBox**. Очевидно, если Вы нуждаетесь в выборе более чем одного элемента, Вы хотели бы использовать **FXList**. С другой стороны, если единственный выбор является соответствующим и если у Вас нет достаточного количества свободного места в пользовательском интерфейсе, чтобы вывести на экран список, поле комбинированного списка или поле списка - хороший, компактный способ скрыть содержание списка, когда оно не необходимо.

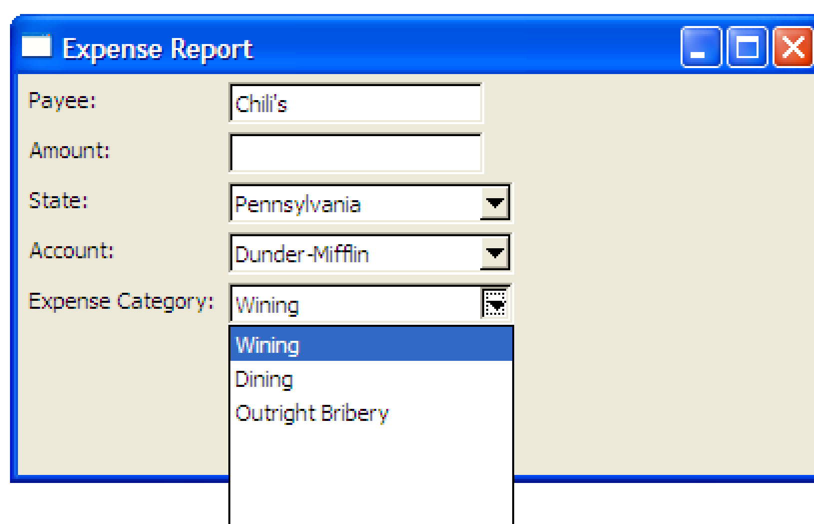



Figure 9.3: Separated at birth? FXComboBox and FXListBox

Различия между полями комбинированного списка и полями списков являются тонкими, однако, в зависимости от того, как Вы используете их, они являются довольно

взаимозаменяемыми. Я имею в виду, можете Вы говорить о различии между ними в рисунке 9.3? Я тоже не могу. В основном, если все, в чем Вы нуждаетесь - это выбрать элемент из списка, Вы должны использовать **FXListBox**. Если Вы хотите быть в состоянии ввести текстовую строку как альтернативу существующим элементам списка и видеть, что элемент добавлен к списку элементов, Вы должны использовать **FXComboBox**.

Как **FXList**, оба эти виджета обеспечивают *prependItem()*, **appendItem()**, **insertItem()** и **removeItem()** методы для того, чтобы изменить содержание списка:

http://media.pragprog.com/titles/fxruby/code/comboboxexample.rb

```
states = FXListBox.new(matrix,
        :opts => LISTBOX_NORMAL|FRAME_SUNKEN|FRAME_THICK|LAYOUT_FILL_X)
$state_names.each { |name| states.appendItem(name) }
```

В любое время может быть выбран один элемент, и атрибут **currentItem** указывает индекс этого элемента (или -1, если нет никакого текущего элемента).

Так как **FXComboBox** может редактироваться, есть несколько дополнительных проблем для этого виджета. Одна проблема это то, что текст, который пользователь ввёл в текстовое поле поля комбинированного списка должен быть добавлен к списку элементов.

По умолчанию поле комбинированного списка использует опцию **COMBOBOX_NO_REPLACE**, что означает, что содержание списка остается тем же самым независимо от ввода в текстовое поле:

http://media.pragprog.com/titles/fxruby/code/comboboxexample.rb

```
accounts = FXComboBox.new(matrix, 20,
        :opts => COMBOBOX_NO_REPLACE|FRAME_SUNKEN|FRAME_THICK|LAYOUT_FILL_X)
```

Когда Вы используете доступный для редактирования **FXComboBox**, Вы не обязательно должны проверять **currentItem**, чтобы узнать выбор пользователя, так как он возможно, ввел некоторый новый текст в текстовое поле. По этой причине, Вы должны вместо этого проверить значение текстового атрибута поля комбинированного списка к определите его текущее значение:

http://media.pragprog.com/titles/fxruby/code/comboboxexample.rb

```
accounts.connect(SEL_COMMAND) do |sender, sel, data|
        assign_expense_account(sender.text)
end
```

Отметьте, что значение поля комбинированного списка переданное в сообщении **SEL_COMMAND** фактически равно значению его текстового атрибута, таким образом, для этого примера Вы, передали в данные непосредственно в метод *assign_expense_account()*.

Если Вы хотели бы строки, которые пользователь вводит в текстовое поле, добавить к списку полей комбинированного списка, у Вас есть несколько вариантов как поместить их туда:

- Используйте опцию **COMBOBOX_INSERT_FIRST**, чтобы вставить новый элемент в начало списка.
- Используйте опцию **COMBOBOX_INSERT_LAST**, чтобы вставить новый элемент в конец списка.
- Используйте опцию **COMBOBOX_INSERT_BEFORE**, чтобы вставить новый элемент перед текущим элементом.
- Используйте опцию **COMBOBOX_INSERT_AFTER**, чтобы вставить новый элемент после текущего элемента.

В моем опыте, опции **COMBOBOX_INSERT_BEFORE** и **COMBOBOX_INSERT_AFTER** немного сбивают с толку пользователя, и все обычно используют только опцию **COMBOBOX_INSERT_FIRST**. Отметьте, что у **FXComboBox** нет встроенной опции, чтобы автоматически поддержать порядок сортировки элементов, но Вы можете это сделать, вызывая *sortItems()* для комбинированного списка во время обработчика **SEL_COMMAND**:

http://media.pragprog.com/titles/fxruby/code/comboboxexample.rb

```
categories.connect(SEL_COMMAND) do |sender, sel, data|
	assign_expense_category(sender.text)
	sender.sortItems
end
```

Вызов *sortItems()* не будет нарушать текст, вводимый в текстовое поле, но если Вы нажмете кнопку стрелки, чтобы вытолкать область списка вниз, то Вы увидите, что недавно добавленный элемент появляется в корректной позиции в сортированном списке.

Виджеты, рассмотренные в этой главе являются плоскими списками из элементов. **FXRuby** также поддерживает иерархически структурированные данные посредством виджета **FXTreeList**, и мы обсудим это далее.

## 9.3 Branching Out with Tree Lists

В отличие от **FXList**, **FXComboBox** и **FXListBox**, которые оперируют плоскими списками, **FXTreeList** разработан для использования с иерархически структурированными данными.

Хотя мы используем слово дерево, чтобы описать данные этого списка и представление, Вы должны отметить, что это не похоже на классические древовидные структуры данных, которые Вы, возможно, изучали в своих классах информатики. Одна особенно запутывающая точка то, что стандартная документация для класс **FXTreeList** использует термин корневой элемент, чтобы обратиться к любому из самых верхних видимых элементов в дереве. Фактический корень дерева никогда не показывается на экране, и мы можем обратитесь к этому только косвенно при использовании **API FXTreeList**.

Как только Вы привыкаете к терминологии используемой FXRuby, чтобы говорить о классе **FXTreeList**, Вы найдете, что это удобно практически. Вы можете изменить контент древовидного списка, используя знакомые методы *prependItem()*, *insertItem()*, *appendItem()* и *removeItem()*, хотя соглашения о вызовах немного отличаются из-за иерархической природы списка. Первый параметр методов *prependItem()* и *appendItem()* - ссылка на родительский элемент для элемента, который Вы добавляете.

Если это - верхний элемент, передаётся NIL как первый параметр:

http://media.pragprog.com/titles/fxruby/code/treelistexample.rb

```
treelist = FXTreeList.new(treelist_frame,
	:opts => TREELIST_NORMAL|TREELIST_SHOWS_LINES| \
		TREELIST_SHOWS_BOXES|TREELIST_ROOT_BOXES|LAYOUT_FILL)
artist_1      = treelist.appendItem(nil, "Alison Kraus" )
album_1_2   = treelist.appendItem(artist_1, "Forget About It" )
track_1_2_3 = treelist.appendItem(album_1_2, "Ghost in this House" )
track_1_2_2 = treelist.prependItem(album_1_2, "Maybe" )
track_1_2_1 = treelist.insertItem(track_1_2_2, album_1_2, "Stay" )
album_1_1   = treelist.prependItem(artist_1, "Every Time You Say Goodbye" )
```

Есть три опции, которые Вы можете использовать, чтобы управлять как соединения между родительскими и дочерними элементами в древовидном списке выведены на экран. Если опция **TREELIST_SHOWS_LINES** выбрана, древовидный список рисует точками линии от родительского элемента до каждого из его дочерних элементов. Если выбран **TREELIST_SHOWS_BOXES**, древовидный список выведет на экран маленькое поле слева от любого элемента дерева, у которого есть один или более дочерних элементов; если тот древовидный элемент расширен, поле будет содержать тире, и если древовидный элемент будет свернут, то это будет содержать знак плюс. Теперь, по некоторым причинам, опция **TREELIST_SHOWS_BOXES** применяется только к элементам, вложенным где-нибудь ниже высокоуровневых элементов. Если Вы также хотите видеть поля рядом с высокоуровневыми элементами (**FOX** называет их элементами корневого уровня), Вы должны также передать в опция **TREELIST_ROOT_BOXES**. Отметьте что опция **TREELIST_ROOT_BOXES** не имеет никакого эффекта, если **TREELIST_SHOWS_BOXES** также не включено.

Сказав все о этих трех опциях, как показано в примере кода, я никогда не находил серьезное основание опустить любого из них. Рисунок 9.4, даёт Вам общее представление на что дерево списка похоже в этом случае.

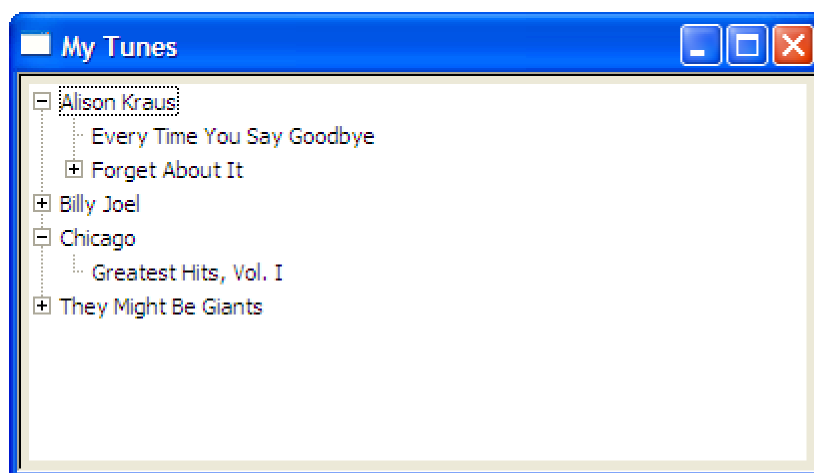



Figure 9.4: A sample FXTreeList

## Keeping Track of the Selection

**FXTreeList** поддерживает те же самые виды режимов выбора как **FXList**, и они работают теми же самыми способами, таким образом, вещи которым Вы уже учились о них применяются здесь также. Атрибут **currentItem** все еще указывает на последний элемент, по которому щелкнули, хотя в этом случае это - ссылка на **FXTreeItem** вместо целочисленного индекса.

Определение, какие элементы выбраны в древовидном списке, может быть хитрым, когда режим выбора учитывает многократно выбранные пункты. Прямой способ сделать это, отследить выбранные пункты в Массиве (или некотором другом контейнере) и затем использовать сообщения **SEL_SELECTED** и **SEL_DESELECTED** от **FXTreeList**, чтобы обновить массив:

http://media.pragprog.com/titles/fxruby/code/treelistexample.rb

```
selected_items = []
treelist.connect(SEL_SELECTED) do |sender, sel, item|
        selected_items << item unless selected_items.include? item
end
treelist.connect(SEL_DESELECTED) do |sender, sel, item|
        selected_items.delete(item)
end
```

Этот метод работает хорошо на списке дерева любого размера, потому что это недорого, в вычислительном отношении. Если Вы знаете, что древовидный список не идет в содержите все, что много элементов, однако, Вы можете счесть это просто пересечением дерево каждый раз текущие изменения элемента, и запись, который элементы выбраны, достаточно быстро в Ваших целях. Только перехватите сообщение **SEL_COMMAND** от **FXTreeList**:

http://media.pragprog.com/titles/fxruby/code/treelistexample.rb

```
treelist.connect(SEL_COMMAND) do |sender, sel, current|
	selected_items = []
	treelist.each { |child| add_selected_items(child, selected_items) }
end
```

Это add_selected_items(), который проходит дерево рекурсивно и смотрит, какие элементы выбраны:

http://media.pragprog.com/titles/fxruby/code/treelistexample.rb

```
def add_selected_items(item, selected_items)
	selected_items << item if item.selected?
	item.each { |child| add_selected_items(child, selected_items) }
end
```

Теперь прежде, чем мы закончим эту главу, мы рассмотрим суперсекретный прием о соединении щелчка правой кнопки pop-up меню с **FXTreeList**.

## Creating Context Menus for Tree Items

Пользователи привыкли к возможности щелкнуть правой кнопкой по объекту в пользовательском интерфейсе, чтобы вывести на экран контекстно-зависимое раскрывающееся меню для того объекта. Вы можете сделать это с почти любым видом объекта в **FXRuby**, но не уверены, что это подходит, когда решаете добавить **FXTreeList** к приложению. По этой причине я собираюсь представить небольшой рецепт для того, как сделать, чтобы щелчок правой кнопкой по древовидному списку выводил pop-up меню, принимая во внимание, что подобный метод может применяться к другим виджетам. Я собираюсь вскользь пройтись по деталям различных частей меню непосредственно, но мы раскроем это подробно позже, в Главе 13, *«Усовершенствованное управление Меню»*, на странице 187.

Первый шаг должен поймать сообщение **SEL_RIGHTBUTTONRELEASE**, которое **FXTreeList** передаёт к его цели. Если мышь перемещалась в то время как кнопка была нажата (событие **SEL_RIGHTBUTTONPRESS**) и отпущена, вызов *moved?()* возвратит *true*, и в этом случае мы игнорируем событие. Иначе, мы можем использовать координаты окна в event data, чтобы определить, какой элемент в дереве был выделен:

http://media.pragprog.com/titles/fxruby/code/treelistexample.rb

```
treelist.connect(SEL_RIGHTBUTTONRELEASE) do |sender, sel, event|
	unless event.moved?
		item = sender.getItemAt(event.win_x, event.win_y)
		unless item.nil?
			# ...
		end
	end
end
```

Метод *getItemAt()* возвратит **nil**, если не будет никакого элемента в указанных координатах. Иначе, он возвратит ссылку на этот **FXTreeItem**.

Следующий шаг - создание **FXMenuPane** и добавление одной или более команды меню к нему:

http://media.pragprog.com/titles/fxruby/code/treelistexample.rb

```
treelist.connect(SEL_RIGHTBUTTONRELEASE) do |sender, sel, event|
	unless event.moved?
		item = sender.getItemAt(event.win_x, event.win_y)
		unless item.nil?
			FXMenuPane.new(self) do |menu_pane|
				play = FXMenuCommand.new(menu_pane, "Play Song" )
				play.connect(SEL_COMMAND) { play_song_for(item) }
				info = FXMenuCommand.new(menu_pane, "Get Info" )
				info.connect(SEL_COMMAND) { display_info_for(item) }
				# ...
			end
		end
	end
end
```

Наконец, создайте menu pane, вызовите *popup()* на ней, чтобы вывести ее на экран, и затем запустите, вложенный цикл выполнения сосредоточенный на этой области меню:

http://media.pragprog.com/titles/fxruby/code/treelistexample.rb

```
treelist.connect(SEL_RIGHTBUTTONRELEASE) do |sender, sel, event|
	unless event.moved?
		item = sender.getItemAt(event.win_x, event.win_y)
		unless item.nil?
			FXMenuPane.new(self) do |menu_pane|
				play = FXMenuCommand.new(menu_pane, "Play Song" )
				play.connect(SEL_COMMAND) { play_song_for(item) }
				info = FXMenuCommand.new(menu_pane, "Get Info" )
				info.connect(SEL_COMMAND) { display_info_for(item) }
				menu_pane.create
				menu_pane.popup(nil, event.root_x, event.root_y)
				app.runModalWhileShown(menu_pane)
			end
		end
	end
end
```

Как только пользователь щелкает по одной из команд меню или щёлкает вне pop-up меню, область меню будет скрыта и приложение вернётся из цикла события, запущенного вызовом *runModalWhileShown()*. Рисунок 9.5 показывает на что раскрывающееся меню похоже, когда я щелкаю правой кнопкой по одной из песен в списке. Это простой код, чтобы добавить к приложению, и когда используется должным образом, это может действительно улучшить удобство пользования программы.

Figure 9.5: Adding a context menu for the tree list

Так, теперь у нас есть опции для того, чтобы иметь дело с обоими плоскими списками данных также как вложенные списки данных. Затем мы собираемся рассмотреть еще один из виджетов, который **FXRuby** предоставляет для наборов данных, и это - виджет **FXTable**.

## 9.4 Displaying Tabular Data with FXTable

Виджет **FXTable** - один из более сложных виджетов в Инструментарии **FOX**. Вновь прибывшие иногда путают виджет **FXTable** с менеджером по расположению **FXMatrix**, который Вы можете использовать, чтобы разметить набор из виджетов в строках и столбцах. **FXTable** действительно размечает его содержание в строках и столбцах, но это не менеджер по расположению; в некоторых другие инструментариях, Вы, возможно, встречали этот вид виджета, называемого как виджет сетки или виджет электронной таблицы.

### Storing Data in a Table

Наше исследование **FXTable** начинается со взгляда на то, как составить таблицу и добавить некоторые данные в неё. В этом разделе мы собираемся учиться немного о том, как таблица фактически управляет своими данными. Мы будем видеть что таблица делает это очень эффективно, и мы будем также учиться как определить элементы, которые могут вмещать многократные табличные ячейки (*span*).

Как и списочные виджеты, которые мы рассмотрели в предыдущей главе, таблицы пусты по умолчанию. Самый эффективный способ заполнить таблицу состоит в том, чтобы использовать метод *setTableSize()*:

http://media.pragprog.com/titles/fxruby/code/treelistexample.rb

```
table = FXTable.new(self, :opts => LAYOUT_FILL)
table.setTableSize(10, 10)
```

Важная вещь, которую необходимо знать о *setTableSize()* и всех методах которые изменяют размер таблицы - это, что **FXTable** делает различие между пустыми ячейками и теми, у которых есть некоторый контент (или данные) связанный с ними. Оба вида ячеек занимают место на экране когда таблица отрисована, но внутренне, **FXTable** выделяет память (в форме объекта **FXTableItem**) только для тех ячеек, у которых фактически есть контент. Это делает таблицу очень эффективной с точки зрения использования памяти, и это означает Вы можете сохранить довольно большие таблицы с очень небольшими накладными расходами.

Вы должны также понять, что метод *setTableSize()* является разрушительным методом. Инициализируете ли Вы табличный размер или просто изменяете размеры таблицы, чтобы сделать её больше или меньше, *setTableSize()* уничтожает все существующие табличные элементы. Так, если Ваша таблица уже содержит некоторые данные и Вы только хотите увеличить её на несколько строк или столбцов, вызов *setTableSize()* не является способом сделать это. Вместо этого используйте некоторую комбинацию из методов *appendRows()*, *appendColumns()*, *insertRows()* и *insertColumns()*.

Элемент *spanning* вмещает в себя больше что одну позицию в таблице. Вы можете создать элемент *spanning* просто передавая в элемент к *setItem()* несколько смежных строк и столбцов:

http://media.pragprog.com/titles/fxruby/code/tabltexample1.rb

```
table.setItemText(2, 1, "This is a spanning item" )
table.setItemJustify(2, 1, FXTableItem::CENTER_X)
spanning_item = table.getItem(2, 1)
table.setItem(2, 2, spanning_item)
table.setItem(2, 3, spanning_item)
table.setItem(3, 1, spanning_item)
table.setItem(3, 2, spanning_item)
table.setItem(3, 3, spanning_item)
```

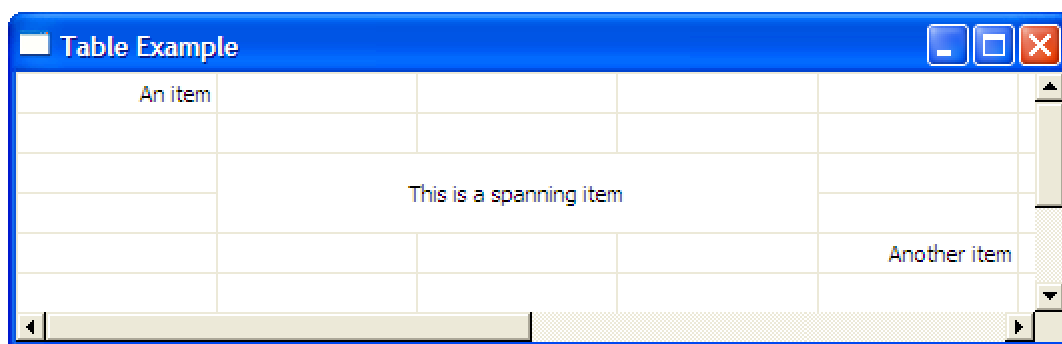



Figure 9.6: Table with a spanning item

В этом примере элемент в позиции (2, 1) охватывает блок 3-2 ячеек в таблице. Когда эта часть таблицы отрисована, ни одна из внутренних частей линий сетки не будет показана. Рисунок 9.6 показывает то, на что похож в таблице этот элемент.

## Modifying the Table Display Options

До сих пор мы говорили о том, как Вы можете добавить или изменить табличные данные. Одинаково важная тема - отображение этих данных и более определенно действия пользователя по управлению таблицей.

По умолчанию сетка по горизонтали и вертикали выведена на экран так, чтобы границы отдельных табличных ячеек были ясно очерчены. Если Вы хотели бы выключить отображение линий сетки, установите одно или оба атрибута **horizontalGridShown** и **verticalGridShown** в значение *false*:

```
table.horizontalGridShown = false
```

Все ячейки в строке имеют ту же самую высоту, и все ячейки в столбце имеют ту же самую ширину. Однако, разные строки могут иметь разную высоту, и разные столбцы

могут быть разной ширины. По умолчанию, пользователь не может изменить ни один из этих размеров. Вы можете всегда изменить высоту строки и ширину столбцов программно, используя такие методы как *setRowHeight()* и *setColumnWidth()*, но чтобы позволить пользователю в интерактивном режиме изменять их размеры, Вы должны включить флаг **TABLE_ROW_SIZABLE**, флаг **TABLE_COL_SIZABLE**, или оба:

```
table.tableStyle |= TABLE_COL_SIZABLE
```

Когда одна или обе эти опции включены, пользователь может щелкнуть разделитель между двумя элементами в строке (или столбце) в заголовке и перетаскиванием изменить размеры соседних строк (или столбцов).

Что касается оглавлений столбцов и строк, Вы можете также управлять их содержанием, чтобы обеспечить (например) заголовки для столбцов таблицы:

http://media.pragprog.com/titles/fxruby/code/tableexample2.rb

```
table.setColumnText(0, "Ruby 1.8.6")
table.setColumnText(1, "Ruby 1.9")
table.setColumnText(2, "JRuby")
table.setColumnText(3, "Rubinius")
```

Если Вы хотите выключить отображение заголовка строки (симпатичный запрос в понедельник), сначала измените его режим на **LAYOUT_FIX_WIDTH**, и затем установите его ширину, чтобы обнулить пиксели. Вы можете сделать то же самое для заголовка столбца устанавливая **columnHeaderMode** в **LAYOUT_FIX_HEIGHT** и **columnHeaderHeight** в ноль:

```
table.rowHeaderMode = LAYOUT_FIX_WIDTH
table.rowHeaderWidth = 0
table.columnHeaderMode = LAYOUT_FIX_HEIGHT
table.columnHeaderHeight = 0
```

В таблице на рис. 9.6, на предыдущей странице, заголовки строк и столбцов скрыты.

У Вас также есть определенная свобода управления отображением элементов отдельной таблицы. Каждый табличный элемент может иметь ассоциацию с текстовой строкой и значком. Вы можете измените эти значения, используя методы *setItemText()* и *setItemIcon()*.

```
table.setItemText(5, 3, "Timeout" )
table.setItemIcon(5, 3, stopwatch_icon)
table.setItemJustify(5, 3, FXTableItem::CENTER_X)
table.setItemIconPosition(5, 3, FXTableItem::BEFORE)
```

В таблице на рисунке 9.7 часть ячеек содержит текст, выровненный по правому краю, а остальные содержат текст и иконку, выровненные по центру.

Наконец, пользователь может отредактировать содержание табличной ячейки, дважды щелкнув по этой ячейке и введя некоторый новый текст нажав клавишу *Enter*. Вы можете отключите эту опцию, установив атрибут **editable** в *false*:

```
table.editable = false
```

Table Example

| | Ruby 1.8.6 | Ruby 1.9 | JRuby | Rubinius |
|---|---|---|---|---|
| app_answer | 1.00 | 5.46 | 1.79 | 2.85 |
| app_factorial | Error | 1.28 | 0.48 | Error |
| app_easier_fact | 1.00 | 0.87 | 0.28 | 0.48 |
| app_fib | 1.00 | 4.70 | 2.55 | 1.55 |
| app_mandelbrot | 1.00 | 2.63 | 0.55 | Error |
| app_pentomino | 1.00 | 2.21 | 0.81 | Timeout |

Figure 9.7: Table items with icons

До сих пор мы сосредотачивались на аспектах отображения в **FXTable**: как поместить данные и как изменить их отображение. Как и другие виджеты, которые мы рассмотрели в этой главе, таблицы также полезны как механизм ввода. Чтобы закончить этот раздел, мы собираемся рассмотреть, как пользователи могут делать выбор в таблицах.

## Управление Табличным Выбором

Таблица несколько менее гибка чем виджеты списка с точки зрения модели выбора. Она поддерживает только один режим выбора, и в том режиме Вы можете выбрать или единственную ячейку или непрерывный блок ячеек. Вы не может, например, выбрать одну ячейку в верхнем левом углу и другую ячейку в нижнем правом углу не выбирая все ячейки в между ними.

Когда Вы щелкаете в ячейке, чтобы начать создавать выбор, эта ячейка становится anchor ячейкой. Атрибуты **anchorRow** и **anchorColumn** таблицы содержат индексы строки и столбца данной ячейки. Если Вы удерживаете клавишу *Shift* и щелкните где-то в другом месте в таблице, выбор будет расширен от базовой ячейки к ячейке, по которой Вы щелкнули. Как это имело место с виджетом списка, текущий элемент (идентифицированный **currentRow** и **currentColumn**) это последняя ячейка, по которой Вы щелкнули.

Важно распознать, что выбор не "растет", чтобы включать и ранее выбранные ячейки и недавно выбранные ячейки, если они все, не лежат на той же самой стороне привязки. Другими словами, выбор в таблице всегда основан на базовой ячейке. Это может быть немного парадоксально сначала. Атрибуты **SelStartRow**, **selEndRow**, **selStartColumn** и **selEndColumn** будут всегда содержать индексы начальной и конечной строки и столбца для всего выбора, когда есть тот. Помните, что исходя из работы модели выбора в таблице, одна из тех конечных точек — любая (**selStartRow**, **selStartColumn**), или (**selEndRow**, **selEndColumn**) — будет anchor cell.

Когда пользователь щелкнет по заголовку строки, все ячейки в той строке станут выбранными. Аналогично, когда пользователь щелкает по заголовку столбца, все ячейки в том столбце становятся выбранными. Вы можете отключить это поведение установкой опций **TABLE_NO_ROWSELECT** и **TABLE_NO_COLSELECT**:

```
# Disable row and column selections
table.tableStyle |= TABLE_NO_ROWSELECT|TABLE_NO_COLSELECT
```

Таблица посылает сообщение **SEL_COMMAND**, когда Вы щелкаете по табличному элементу, и данные сообщения в экземпляре **FXTablePos**. **FXTablePos** это простой объект данных с методами доступа к *row* и *col* для чтения строки и столбца выбранного табличного элемента.

Вы можете, конечно, программно изменить выбор, хотя Вы обычно полагаетесь на пользователя, чтобы выполнить эти действия в интерактивном режиме. В любом

случае **FXTable** не позволить Вам сделать это если это нарушает его модель выбора. Например, в следующем коде результатом будет только одна ячейка – ячейка (5, 5) будет выбрана:

```
table.selectItem(0, 0)
table.selectItem(5, 5)
```

Для выбора нескольких ячеек используйте метод selectRange():

```
# Select all of the cells between (0, 0) and (5, 5), inclusively
table.selectRange(0, 0, 5, 5)
```

Как это было для **FXTreeList**, самый легкий способ хранить выбранные табличные элементы это сохранить их в Массиве, содержание которого обновляется в ответ на сообщения **SEL_SELECTED** и **SEL_DESELECTED**:

```
selected_items = []
table.connect(SEL_SELECTED) do |sender, sel, pos|
      item = sender.getItem(pos.row, pos.col)
      selected_items << item unless selected_items.include? item
end
table.connect(SEL_DESELECTED) do |sender, sel, pos|
      selected_items.delete(sender.getItem(pos.row, pos.col))
end
```

На этом мы завершим рассмотрение виджетов **FXRuby** для работы с наборами данных, но есть много других, подобных виджетов в библиотеке, которые Вы можете рассмотреть сами. Например, виджет **FXTable** использует пару внутренних виджетов **FXHeader**, чтобы вывести на экран заголовки строк и заголовки столбцов, но Вы можете вытащить тот виджет и использовать его отдельно. **FXFoldingList** - своего рода пересечение **FXTreeList** и **FXHeader**, который позволяет Вам связывать multiple столбцы данных с каждым элементом в древовидном списке. **FXIconList** используется диалоговым окном файла для обеспечения различных видов представлений списка файлов, но Вы можете также применить его чтобы вывести на экран другие виды списков. Вы найдете документацию для каждого из этих виджетов в документации API **FXRuby**, и стандартная поставка FXRuby включает примеры для каждого из них.

Далее мы собираемся рассмотреть более сложные виджеты. Виджет **FXText**, который Вы можете использовать для редактирования больших текстовых документов.

# Глава 10

# Editing Text with the Text Widget

В Главе 8, Создавая Простые Виджеты, на странице 100, мы научились как использовать виджет **FXTextField**, для ввода текста пользователем. **FXTextField** - хороший выбор, когда Вы имеете дело с коротким однострочным текстом, таким как данные формы или имя файла. Ясно что это не решение на все случаи жизни. В этой главе мы рассмотрим как использовать виджет **FXText**, который Вы можете использовать для просмотра и редактирования многострочных текстовых документов. **FXText** - один из наиболее сложных виджетов в библиотеке FOX, и хотя у него есть много общих черт с **FXTextField**, на первый взгляд его API является немного сложным. Вы получите базовые знания о виджете **FXText** в этой главе, но Вы должны смотреть документацию по API, чтобы узнать о его расширенных функциях.

Во-первых, хорошие новости. Если Вы просто нуждаетесь в полнофункциональном компоненте редактирования текста где-нибудь в Вашем приложении, Вам не надо ничего более чем создать виджет **FXText**, инициализировать его некоторым текстом по умолчанию, и затем попросите у него его содержание в некоторое время позже используя его текстовый атрибут:

```
text = FXText.new(text_frame, :opts => TEXT_WORDWRAP|LAYOUT_FILL)
text.text = "By default, the text buffer is empty."
```

**FXText** предлагает почти всю функциональность, из которой Вы ожидали бы компонент редактирования текста. На рисунке 10.1 представлены некоторые привязки клавиш, которые распознает **FXText**. Для всех нажатий клавиш, которые перемещают курсор, Вы можете удержать клавишу *Shift* для выбора текста между тем, где курсор расположен и куда он перемещается.

| **Keystroke** | **Action** |
|---|---|
| Left , Right , Up , or Down | Moves left, right, up or down |
| Ctrl + Left or Ctrl + Right | Moves to end of previous word or beginning of next word |
| Ctrl + Up or Ctrl + Down | Scrolls up or down one line, leaving cursor in place |
| Home | Moves to beginning of line |
| Ctrl + Home | Moves to beginning of document |
| End | Moves to end of line |
| Ctrl – End | Moves to end of document |
| PgUp | Pages up |
| PgDn | Pages down |
| Insert | Toggles overstrike mode |
| Ctrl + Insert or Ctrl + C | Copies selection to clipboard |
| Shift + Insert or Ctrl + V | Pastes clipboard contents |
| Ctrl + A | Selects all text |
| Shift + Delete or Ctrl + X | Cuts selected text |

Figure 10.1: Some keystrokes that FXText understands

Кроме возможности отформатировать большие тексты, **FXText** также полезен для простого отображения текста во многих обстоятельствах. Если необходимо только чтение, установите **editable** в **false**:

```
text.editable = false
```

Если Вы хотите использовать **FXText** более сложным способом, то Вы должны изучить его API. Давайте сделаем это теперь, начнём с API для того, чтобы добавить и удалить текста из текстового буфера.

## Добавление и удаление текста

Когда Вы используете виджет **FXTextField**, есть только один путь программно измените его значение, это присвоить новое значение его текстовому атрибуту. Это разумно до тех пор, пока Вы работаете с относительно короткой строкой и это будут очень небольшие издержки - заменить все за один раз. Но совсем другая история когда Вы работа с виджетом **FXText**. **FXText** оптимизирован для того, чтобы работать с очень большими телами текста, и как результат имеет много методов для того, чтобы изменить тот текст.

По умолчанию текстовый буфер для виджета **FXText** пуст. Вы можете установить его значение, присваивая строку его текстовому атрибуту, но это справедливо в начале работы. Проблема - когда Вы присваиваете текст, используя текстовый атрибут, виджет **FXText** должен повторно вычислить множество метрик, которые используются внутри его, например позиции всех строк. Это в вычислительном отношении дорогая работа. Когда у Вас есть выбор, Вы должны вместо этого использовать один из других методов, которые обеспечивает **FXText**. Например, чтобы добавить некоторый текст в конец буфера, используйте метод *appendText()*:

```
text.appendText(additional_text)
```

Метод **appendText()** принимает дополнительный второй параметр, у которого значение по умолчанию *false*:

```
text.appendText(additional_text, true)   # notify target of change
```

Если Вы установили значение *true* как дополнительный второй параметр в *appendText()*, **FXRuby** отправит сообщения **SEL_INSERTED** и **SEL_CHANGED** в текстовый виджет после того как текст будет модифицирован:

```
text.connect(SEL_INSERTED) do |sender, sel, change|
puts "The string #{change.ins} was inserted at position #{change.pos}"
end
```

*Все методы, которые мы обсудим в этом разделе, принимают этот дополнительный параметр **true** как заключительный параметр. См. документацию API для **FXText** для большего количества деталей о том как сообщения отправляются текстовому виджету в результате.*

Данные для сообщения **SEL_INSERTED** — это экземпляр **FXTextChange**, который включает информацию о тексте, который вставлен (заменен или удалён) от текстового буфера виджета **FXText**. Используйте *insertText()*, чтобы вставить строку в определенной позиции в текстовый буфер:

```
text.insertText(pos, inserted_text)
```

Вы можете заменить один блок текста другим используя *replaceText()*:

```
text.replaceText(pos, amount, replacement_text)
```

Отметьте, что блок текста, который Вы заменяете, может быть короче или длиннее чем заменяющая строка; именно поэтому Вы должны определить как много символов Вы хотите заменить.

Наконец, Вы можете удалить блок текста, используя *removeText()*:

```
text.removeText(pos, amount)
```

До сих пор мы замяли тот факт, что для редактирования пользователем текста мы, возможно, не в состоянии определить позицию строк, требуемых для параметров методов *insertText()*, *replaceText()* и *removeText()*. В следующем разделе мы рассмотрим некоторые из методов где **FXText** определяет где мы находимся в текстовом буфере.

## Навигация в тексте

Методы, которые мы обсуждали в предыдущем разделе, предполагают что Вы уже знайте позицию, в которой Вы хотите добавить, заменить, или удалить некоторый текст. Вы можете всегда прочитать значение атрибута **cursorPos** и определить текущую позицию курсора, но иногда Вы должны найти некоторую относительную позицию (или некоторое абсолютно отличающееся место).

Давайте опустимся на самый низкий уровень. Если Вы хотите найти позицию предыдущего символа (относительно известной позиции в текстовом буфере), Вы может использовать метод *dec()*. Аналогично, чтобы найти позицию следующего символа, используйте метод *inc()*:

```
previous_character_pos = text.dec(text.cursorPos)
```

Да ведь Вы можете спросить, не можем ли мы просто вычесть или добавьте 1 к текущей позиции, чтобы найти символ в смежной позиции? Если Вы попробуете, то возможно это будет работать просто великолепно. Причина того, что Вы должны быть осторожными относительно этого в том, что **FXText** (и любой виджет **FOX**, который занят выводом на экран или иначе имеет дело со строковыми значениями) работает с UTF-8. Каждый символ в Unicode может быть представлен больше чем одним байтом в текстовом буфере. Поэтому для того, чтобы точно найти Ваш путь от одной позиции до следующей, Вы должны всегда использовать *inc()* и *dec()* методы.

Вы можете использовать методы *wordStart()* и *wordEnd()*, чтобы определить позицию начала или окончания слова, которое занимает текущую позицию в тексте. Например, чтобы определить стартовую позицию текущего слова, используйте что-то вроде этого:

```
current_word_start_pos = text.wordStart(text.cursorPos)
```

Вы можете использовать методы *leftWord()* и *rightWord()*, чтобы найти позицию для конца предыдущего слова или начало следующего слова:

```
next_word_start_pos = text.rightWord(text.cursorPos)
```

В том же направлении Вы можете найти позицию в начала или конца строки, используя методы *lineStart()* и *lineEnd()*:

```
current_line_start_pos = text.lineStart(text.cursorPos)
```

Есть даже методы, чтобы определить стартовые позиции предыдущей или следующей строки в тексте. Отметьте, что есть тонкое различие между "*lines*" текста и "*rows*" текста. *Line* текста (иногда называемый логической строкой), не заканчивается, пока Вы не достигните символа новой строки в текстовом буфере, даже если текст содержит более чем несколько строк на экране. Если Вы выключили опцию перехода на новую строку для **FXText**, начало *line* и начало *row* относительно данной позиции, будет равно, как будет конц *line* и конец *row* относительно этой позиции.

Важно быть знакомым с методами, которые мы обсудили в этом разделе, но они адресуют только один аспект навигации через текст. Может быть, что Вас не так интересует текущая позиция и как переместиться от этой начальной точки, а скорее то, что Вы обеспокоены обнаружением других расположений в тексте. Мы рассмотрим тот аспект навигации.

## Поиск в тексте

До сих пор мы говорили о том, как найти Ваш путь через текст, относительно известной позиции в текстовом буфере. Вы можете использовать метод *findText()*, чтобы искать определенную строку в текстовом буфере. По умолчанию, *findText()*

возвратит пару массивов, которая говорит Вам, где он нашел первое совпадение строки поиска:

```
text.text =
      "Now is the time for all good men " +
      "to come to the aid of their country."
first, last = text.findText("the" ) # returns [7], [10]
```

Первый массив содержит стартовую позицию всех соответствий которые *findText()* нашёл. Второй массив содержит конечную позицию, плюс 1. Именно поэтому значение *last* в предыдущем примере [10] а не [9]. Если *findText()* не найдет соответствий, то оба возвращаемых значения будут *null*:

```
first, last = text.findText("women")      # returns nil, nil
```

По умолчанию, *findText()* запускает поиск с начала текстового буфера и ищет точное совпадение Вашей строки поиска. Вы можете измените стартовую позицию для поиска, определяя значение для параметра *start*:

```
first, last = text.findText("the" , :start => 20)    # returns [44], [47]
```

Вы можете также изменить параметры поиска, например, чтобы выполнить поиск нечувствительный к регистру или искать назад от стартовой позиции:

```
first, last = text.findText("ThE" , :start => 20,
      :flags => SEARCH_BACKWARD|SEARCH_IGNORECASE) # returns [7], [10]
```

Для более сложных поисков Вы можете использовать встроенный в **FOX** механизм регулярных выражений, **FXRex**. Например, чтобы найти первое четырехбуквенное слово в тексте, мы могли использовать поиск подобный этому:

```
first, last = text.findText("\\w{4}" ,
      :flags => SEARCH_REGEX)       # returns [11], [15]
```

Если регулярное выражение содержит группы фиксации, массивы, возвращенные *findText()*, будут содержать начальную и конечные позиции для каждой из этих групп:

```
first, last = text.findText("the (\\w+ (\\w+)) (\\w+)" ,
      :flags => SEARCH_REGEX)       # returns [7, 11, 16, 20], [23, 19, 19, 23]
```

Последний пример более сложен. Первая группа - все соответствия выражению, и это первые элементы первых и последних массивов. Соответствующее выражение, “the time for all” начинается в позиции 7 и заканчивается в 23. Теперь, перемещаясь слева направо, следующая группа соответствующая подвыражению “time for”, которая начинается в 11 и заканчивается в 19. Следующая группа соответствует "for", которая начинается в 16 и заканчивается в 19. Наконец, последняя группа соответствует "all", начинается в 20 и заканчивается в 23.

Механизм регулярного выражения **FOX** очень мощен, но в некоторых случаях синтаксис, возможно, не идентичен стандартным регулярным выражениям Ruby. Вы можете счесть что легче только извлечь текст из виджета **FXText** и затем искать с помощью более знакомых регулярных выражений Ruby:

```
if text.text =~ /the (\w+ (\w+)) (\w+)/
      group0, group1, group2, group3 = $~[0], $~[1], $~[2], $~[3]
end
```

**FXRuby** создает новый объект **String**, который содержит содержимое текстового буфера всякий раз, когда Вы читаете значение текстового атрибута. В зависимости от размера из текстового буфера и как часто Вы должны выполнить поиск, это возможно не будет большим препятствием по использованию ресурсов.

## Применение стилей к тексту

Серьезный недостаток **FOX** и **FXRuby** - отсутствие виджета который может вывести на экран HTML или сложный текст. Виджет **FXText** - определенно не этот вид виджета, хотя он оказывает некоторую минимальную поддержку для "стилизованного" текста.

Первый шаг по включению поддержки стилизованного текста, это установить атрибут стиля в *true*:

```
text.styled = true
```

Следующим шагом определяют один или более стилей, которые будут применены к тексту. Мы сделаем это при первом построении экземпляра **FXHiliteStyle** и затем установкой его атрибута. По умолчанию метод **new()** для класса **FXHiliteStyle** не является чем-то полезным, так как не может инициализировать различные цвета стиля для произвольного виджета **FXText**. Лучше использовать метод *from_text()*, который инициализирует экземпляр **FXHiliteStyle** с текущей цветовой схемой виджета **FXText**:

```
style1 = FXHiliteStyle.from_text(text)
```

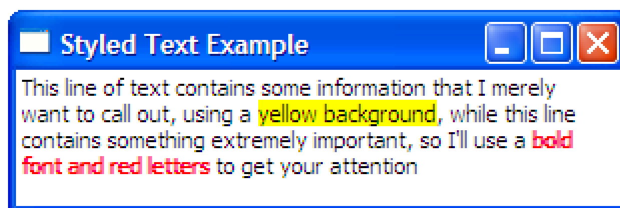



Figure 10.2: Displaying styled text in a text widget

Теперь измените один или более атрибутов для этой установки стиля. Предположим Вы желаете вывести на экран некоторый текст, выделенный с желтым фоном, с некоторым другим полужирным текстом и красным:

```
style1 = FXHiliteStyle.from_text(text)
style1.normalBackColor = "yellow"
style2 = FXHiliteStyle.from_text(text)
style2.normalForeColor = "red"
style2.style = FXText::STYLE_BOLD
```

Затем, мы должны сказать нашему виджету **FXText** использовать этот массив стиля:

```
text.hiliteStyles = [style1, style2]
```

Наконец, используйте метод *changeStyle()*, чтобы установить стиль для указанного блока текста к определенному стилю в массиве стиля. Первые два параметра *changeStyle()* - это начальная позиция и длина текста, стиль которого надо поменять. Мы будем использовать *findText()*, чтобы определить местоположение интересующей нас строки:

```
first, last = text.findText("yellow background")
start_pos = first[0]
length = last[0] - first[0]
text.changeStyle(start_pos, length, 1)
```

Последним параметром *changeStyle()* является индекс стиля. Отметьте что "нулевой" индекс стиля зарезервирован для текстового стиля значения по умолчанию виджета. Это когда Вы передаете ноль как последний параметр *changeStyle()*, текст будет

выведен на экран, используя нормальную, не стилизованную цветовую схему. С тех пор мы передали 1 как индекс стиля в этом примере, **FXText** будет использовать желто-фоновый стиль, чтобы вывести на экран фразу “yellow background” Рисунок 10.2 показывает чем этот пример под Windows.

Мы потратили несколько предыдущих глав, чтобы рассказать о некоторых виджетах, которые Вы можете использовать, чтобы заботиться об основах пользовательского интерфейса. В следующей главе мы сместим наш взгляд на некоторые из инструментов, которые расширяют возможности GUI в FXRuby при создании красивых интерфейсов.

# Глава 11

# Создание Визуально Богатых Пользовательских Интерфейсов

До сих пор мы сосредотачивались на основах пользовательского интерфейса и взаимодействии между приложением и его пользователем. Если это было всё, что обязано иметь приложение, с которым люди любили бы работать, то можно было бы дальше не продолжать. Цена вопроса — это то, что пользователи привыкли работать с приложениями, которые не только функциональны, но и имеют богатый визуальный интерфейс (то есть красивы).

К счастью, для нас, **FXRuby** обеспечивает много возможностей по созданию визуально богатыех пользовательских интерфейсов. В процессе создания приложения Picture Book, мы уже узнали, что **FXRuby** обеспечивает обширную поддержку отображения изображений. В Главе 8, Создавая Простые Виджеты, на странице 100 мы получили краткое введение по различным видам графических объектов, когда мы изучили, как создать значки и использовать их в полях, кнопках, и других виджетах. Хотя большинство кода мы до сих пор писали с использованием шрифта по умолчанию, FXRuby обеспечивает доступ ко всем установленным системным шрифтам так, чтобы Вы могли использовать различные гарнитуры всюду по Вашему желанию. Вы можете также изменить форму курсора мыши при определенных обстоятельствах, чтобы предоставить пользователю визуальный индикатор о состояние приложения или вида работы, которую они выполняют. В этой главе мы будем исследовать все эти функции и увидим, как Вы сможете использовать это эффективно в Ваших собственных приложениях.

Объекты пользовательского интерфейса, которые мы рассмотрим в этой главе, особенные, потому что, в отличие от виджетов, шрифты, курсоры, изображения, и иконки - это совместно используемые ресурсы. Когда Вы создаёте виджет подобный **FXButton**, этот виджет занимает точное положение в иерархии виджета. Другой способ смотреть на это состоит в том, что у кнопки есть только одно родительское окно. В противоположность этому, совместно используемый ресурс может быть связан со многими другими объектами пользовательского интерфейса. Например, когда Вы создаете объект приложения, он создает объект **FXFont** по умолчанию, который используется всеми виджетами в Вашем приложении.

Важность этого состоит в том, что не надо создавать многократные экземпляры этих объектов, если они все имеют те же самые характеристики. Например, предположим что вместо шрифта по умолчанию, который Вы видите всюду в своём приложении, Вы хотите создать пользовательский шрифт, используя Женевскую гарнитуру, для использования в определенном виджете **FXLabel**. Вы сначала создали бы новый объект **FXFont**, и если необходимо, вызвали *create()* для него:

```
geneva_font = FXFont.new(app, "Geneva" , 10)
geneva_font.create
my_special_label.font = geneva_font
```

Затем Вы присвоили бы это метке.

```
my_special_label.font = geneva_font
```

Теперь предположите, что Вы хотите использовать тот же самый Женевский шрифт в того же самого размера в виджете **FXText** в другом месте Вашего приложения. Вы могли бы создать новый объёкт **FXFont** с точно теми же самыми свойствами шрифта, но это было бы тратой ресурсов. Лучший выбор это простое присваивание geneva_font как шрифта для виджета **FXText**.

```
my_special_text.font = geneva_font
```

С тем введением состоялось на наше первое знакомство с совместно используемыми ресурсами **FXFont**.

## Использование пользовательских шрифтов

Вы можете думать что ваш шрифт принимает всё семейство размеров и стилей для выбранного типа шрифта или что применяется только определенный размер и стиль от этого семейства. В **FXRUBY**, объект **FXFont** соответствует второму определению. Если Вы должны вывести на экран ту же самую гарнитуру в двух различных размерах символов, Вы должны создать два различных объекта **FXFont**.

Есть три метода для того, чтобы создать объекты **FXFont**. Все они дают Вам некоторый способ определить Ваши требуемые характеристики шрифта, такие как имя гарнитуры, размер точки, и ширина. **FOX** будет использовать эти параметры, чтобы идентифицировать доступный системный шрифт это лучше всего соответствует тому что Вы хотите. Конструктор шрифта, который Вы будете, вероятно, использовать чаще всего, включает много параметров для определения требуемого шрифта. Например, Вы могли использовать следующий код, чтобы запросить а шрифт 14-point, используя курсивную разновидность гарнитуры Arial:

```
label1 = FXLabel.new(self, "This label uses a 14 point Arial italic font.")
label1.font = FXFont.new(app, "Arial", 14, :slant => FXFont::Italic)
```

Другой метод для того, чтобы создать шрифт включает передачу строки, содержащей описание шрифта. Вы можете проконсультироваться с документацией по API для класса **FXFont** по точному формату этой строки, но вот пример того, как запросить Times 12-point полужирный шрифт:

```
label2 = FXLabel.new(self, "This label uses a 12 point Times bold font." )
label2.font = FXFont.new(app, "Times,120,bold" )
```

Третий метод для того, чтобы создать шрифт включает передачу в объект **FXFontDesc**. **FXFontDesc** - это объект данных с атрибутами которые соответствуют параметрам, что Вы передали бы в первую форму метода **FXFont.new()**, на который мы смотрели (например, имя шрифта, размер и так далее). Практически, Вы обычно не будете создавать объект **FXFontDesc** непосредственно. Во многих случаях Вы получите это из **FXFontDialog** (который мы опишем в Главе 14, «*Providing Support with Dialog Boxes*», на странице 196). Более прямой способ получить объект **FXFontDesc** состоит в том, чтобы использовать метод *listFonts()*.

Метод *listFonts()* возвращает список всех доступных шрифтов. Это - мощное и независимое от платформы метод для того, чтобы выбрать шрифт, так как это не требует от Вас явной идентификации всех характеристик шрифта.

Figure 11.1: Examples of custom fonts

В этом примере мы запросим список всех прямых (некурсив) шрифтов фиксированной ширины и затем только выберем первый:

```
label3 = FXLabel.new(self, "This label should use a fixed-width font.")
fonts = FXFont.listFonts("", :slant => FXFont::Straight,
      :hints => FXFont::Fixed)
label3.font = FXFont.new(app, fonts.first)
```

Если мы хотим быть немного более придирчивыми, мы можем искать наименьшую фиксированную ширину шрифта это - размер 2 точки (20 decipoints) или больше:

```
label4 = FXLabel.new(self,
      "This label should use a very small fixed-width font!" )
fonts = FXFont.listFonts("" , :slant => FXFont::Straight,
      :hints => FXFont::Fixed)
sorted_by_size = fonts.sort { |a, b| a.size <=> b.size }
label4.font = FXFont.new(app, sorted_by_size.find { |f| f.size > 20 })
```

Рисунок 11.1 показывает некоторые из шрифтов, сгенерированных этим примером под Windows. Отметьте что в этом случае, шрифт фиксированной ширины с самым маленьким размером точки (показанный на последней строке) фактически более широк чем предыдущее шрифт фиксированной ширины, который мы выбрали. Это не ошибка; размер точки шрифта - относительная мера, это обычно основано на расстоянии от вершины самого высокого надстрочного элемента к нижней части самого низкого подстрочного элемента.

Как также имеет место с веб-дизайном, Вы должны быть консервативными в числе и виде шрифтов в своих приложениях, чтобы убедиться, что они будут работать хорошо во множестве платформ и в различных разрешениях дисплея.

Другая возможность для sprucing up своего приложения, использование пользовательских курсоров мыши, чтобы обеспечить визуальные индикаторы об изменении состояния приложения или работе, которую выполняет пользователь. Давайте рассмотрим то, как FXRuby использует курсоры.

## Использование различных курсоров мыши

Форма курсора - только одна из тех вещей, о которых Вы обычно не думайте. Для большого количества приложений GUI, которые Вы создадите, у Вас будете курсор по умолчанию, обеспеченный FOX. Для некоторых приложений, однако, полезно изменить форму курсора во время определенных операций, чтобы предоставить визуальный индикатор пользователю о том, что происходит.

Вы можете изменить форму курсора для определенного окна присвоением экземпляра **FXCursor** к его атрибуту **defaultCursor**. Всякий раз, когда указатель мыши входит в окно, форма курсора изменится к форме, которую Вы определили. Класс **FXApp** обеспечивает некоторые встроенные курсоры "по умолчанию", которые Вы можете использовать когда Вы должны изменить форму курсора. В следующем примере, форма курсора изменится на встроенную форму **DEF_HAND_CURSOR** всякий раз, когда пользователь поместит мышь на метку:

```
box = FXLabel.new(frame, "Gimme Five!" ,
      :width => 40, :height => 40,
      :opts => LAYOUT_CENTER_X, :padding => 40)
box.defaultCursor = app.getDefaultCursor(DEF_HAND_CURSOR)
```

Рисунок 11.2, показывает то, на что похож **DEF_HAND_CURSOR**. Если ни один из встроенных курсоров не удовлетворяет Ваши потребности, Вы можете всегда создать свои собственные курсоры. Самый легкий способ сделать это - использовать подкласс **FXGIFCursor**, который может создать курсор из изображения GIF.

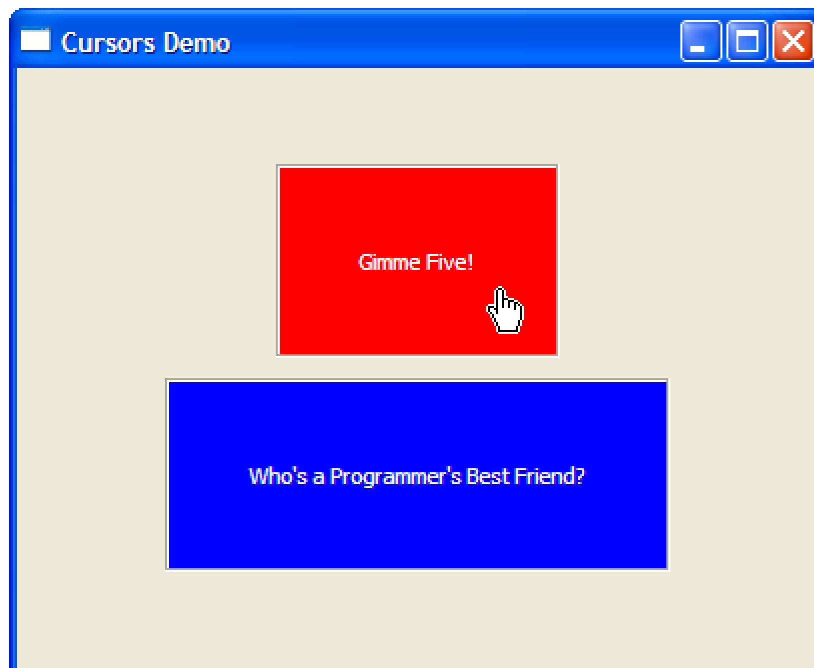



Figure 11.2: Talk to the hand (cursor)

```
box = FXLabel.new(frame,
      "Who's a Programmer's Best Friend?" ,
      :width => 40, :height => 40,
      :opts => LAYOUT_CENTER_X, :padding => 40)
custom_cursor = FXGIFCursor.new(app,
      File.open("rubycursor.gif" , "rb" ).read)
custom_cursor.create
box.defaultCursor = custom_cursor
```

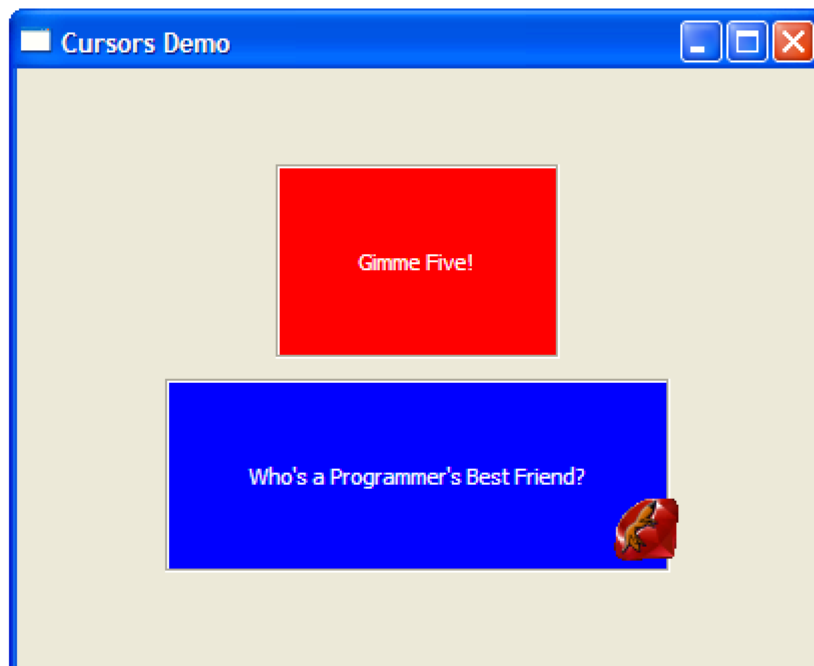



Figure 11.3: Using a GIF image for a custom cursor shape

Рисунок 11.3, показывает пользовательский курсор. Отметьте что размер изображения GIF должен быть 32x32 пикселя или меньше; это из-за ограничения на

размер курсоров в Windows. Также отметьте что **FOX** различает курсор по умолчанию для окна и курсор перетаскивания для окна. Курсор изменяется на перетащить всякий раз, когда окно захвачено (обычно происходит в результате пользовательское удержание кнопки мыши и перетаскивания мыши). Если Вы изменяете курсор по умолчанию для окна, Вы будете, вероятно, изменять перетаскивающий курсор для того окна также, используя атрибут **dragCursor**.

Есть особый случай, который возникает достаточно часто. Когда Ваше приложение делает что-то отнимающее много времени, такое как загрузка большого файла или выполнение сложного вычисления, Вы обычно хотите изменить форму курсора к "занятому" курсору (который похож на наручные часы). Вы можете использовать методы **beginWaitCursor()** и **endWaitCursor()**, чтобы временно переключить курсор от его текущей формы до занятого курсора и назад:

```
app.beginWaitCursor # save current shape and switch to busy cursor
      ... perform time-consuming operation ...
app.endWaitCursor   # revert to previous cursor shape
```

Вы можете также использовать транзакционную форму **beginWaitCursor()**, которая будет автоматически вызывать **endWaitCursor()**, когда блок заканчивается:

```
open_button = FXButton.new(self, "Open File" )
open_button.connect(SEL_COMMAND) do
      app.beginWaitCursor do
            open_file # this may take awhile...
      end
end
```

Важно отметить, что в этом примере пользователь будет не в состоянии сделать что-либо с программой, в то время как она занята открытием файла. Вы должны быть осторожным с этим. Если задача, которую Вы выполняете собирается занять много времени, но может быть сделана в фоновом режиме, Вы должен рассмотреть выполнение этой задачи в отдельном потоке, чтобы Ваша программа смогла реагировать на другие запросы пользователя.

Когда Вы создавали приложение Picture Book, то приобрели большой опыт, по работе с изображениями. В следующих нескольких разделах, мы рассмотрим эту информацию и изучим некоторые новые приемы.

## Создание и отображение изображений

Самый легкий способ начать работать с объектами **FXImage** состоит в том, чтобы использовать один из специфичных для подклассов, таких как **FXJPGImage**, чтобы отобразить непосредственно некоторые изображения. Например, Вы можете создать изображение формата JPEG в одной строке кода:

```
birdsnest_image =
      FXJPGImage.new(app, File.open("birdsnest.jpg", "rb").read)
```

FOX не очень заботится о том,где Вы получаете данные изображения. Если данные где-нибудь в сети, Вы можете так же, как легко использовать стандартную Ruby библиотеку open-url, чтобы получить их:

```
require 'open-uri'

oscar_image = FXJPGImage.new(app, open("http://tinyurl.com/35o8yy").read)
```

Есть подклассы **FXImage** для многих популярных форматов изображений. См. рисунок 11.4 для перечисления обычно используемых типов файлов и соответствующих имён классов.

| Image File Format | Class Name |
|---|---|
| Windows Bitmap (BMP) | FXBMPImage |
| Graphics Interchange Format (GIF) | FXGIFImage |
| Joint Photographic Experts Group (JPEG) | FXJPGImage |
| Portable Network Graphics (PNG) | FXPNGImage |
| Tagged Image File Format (TIFF) | FXTIFImage |

Figure 11.4: FXImage subclasses

Класс **FXImageFrame** - подкласс **FXFrame** чья единственная цель состоит в том, чтобы вывести на экран изображение:

```
FXImageFrame.new(tab_book, birdsnest_image,
      :opts => FRAME_RAISED|FRAME_THICK|LAYOUT_FILL)
```

Виджет **FXImageFrame** не сложен. Если Вы запускаете простое приложение и изменяете размеры окна, чтобы сделать меньше, Вы увидите что изображение не уменьшается, чтобы соответствовать новому размеру окна; оно только осекается по краям. Когда Вы выводите на экран изображение неизвестного размера, Вы могли бы использовать виджет **FXImageView**, который выводит на экран изображение в окне прокрутки:

```
FXImageView.new(tabbook_page, oscar_image, :opts => LAYOUT_FILL)
```

Третий путь использования изображений это рисование в контексте устройства для больших возможностей для приложения общего назначения. Я не собираюсь затрагивать эту тему здесь, но Вы можете посмотреть пример программы в **dctest.rb** в стандартной поставке FXRuby, чтобы видеть, как это работает.

В дополнение к основным возможностям загрузки и отображения изображений, FOX обеспечивает много API для управления и преобразования изображения. Давайте рассмотрим эти функции.

## Управление Данными Изображения

При некоторых обстоятельствах Вы, возможно, должны сделать незначительные модификации изображения. Мы занялись этим в Главе 5, Пункт 2: *Display and Entire Album,* на странице 43, когда мы использовали метод **scale()**, чтобы уменьшить исходные изображения так, чтобы мы могли видеть больше из них одновременно. Класс **FXImage** обеспечивает несколько дополнительных API, которые поддерживают манипулирование и преобразование данных изображения, но они требуют дополнительных усилий с точки зрения установки.

Для начала, по умолчанию **FXImage** уничтожает свою копию оригинала изображения, как только создано серверное представление. Если Вы нуждаетесь в управлении исходным изображением только перед вызовом **create()** для изображения, это поведение по умолчанию не представляет проблему. В другом случае, Вы хотите быть в состоянии управлять данными изображения после начального вызова **create()**, Вы должны сказать FXImage держать копию данных исходного изображения (*знающие люди называют это client-side pixel buffer*), посредством установки опции **IMAGE_KEEP**, когда Вы создайте изображение:

```
@image = FXJPGImage.new(app,
      File.open("birdsnest.jpg", "rb").read,
      :opts => IMAGE_KEEP)
```

Теперь Вы можете вызвать один или больше API для манипулирования изображением, чтобы изменить клиентскую копию изображения пиксельного буфера. Например, Вы можете использомать метод **crop()**, чтобы обрезать части исходного изображения. Рисунок 11.5, на следующей странице показывает демонстрационное изображение со всем его оригинальным содержимом. Давайте посмотрим на то, на что это похоже

после обрезки всего кроме верхнего квадрата изображения.

Figure 11.5: Original image before cropping

```
@image.crop(0, 0, 0.5*@image.width, 0.5*@image.height)
```

Первыми двумя параметрами **crop()** являются координаты для верхнего левого угла области, которую Вы хотите сохранить, и третье и четвертое параметры - ширина и высота той области. Рисунок 11.6, на страница 153, показывает, что остаётся от изображения после того, как оно было обрезано. Как ожидалось, изображение - одна четверть своего первоначального размера и содержит только верхний левый квадрат исходного изображения.

**crop()** изменяет размеры изображения к новой ширине и высоте, копирует часть исходного изображения, и затем заполняет любые разрывы с цветом заливки (который по умолчанию является черным). В большинстве случаев Вы будете использовать **crop()**, чтобы сохранить некоторую область, которая является подмножеством оригинала изображение, но **crop()** достаточно гибок, чтобы позволить Вам делать некоторые необычные вещи. Например, рассмотрите этот вызов **crop()** для того же самого исходного изображения:

```
@image.crop(0.5*@image.width, 0.5*@image.height,
      @image.width, @image.height)
```

В этом примере, мы используем центр изображения как верхний левый угол области обрезки, которая кажется разумной, но мы, также говорим, что размер нового изображения должен иметь ту же самую ширину и высоту как исходное изображение. Рисунок 11.7, на странице 154, показывает результат из этого вызова **crop()**. Отметьте, что новые части, которые не присутствовали в исходное изображение, заполнены в черным.

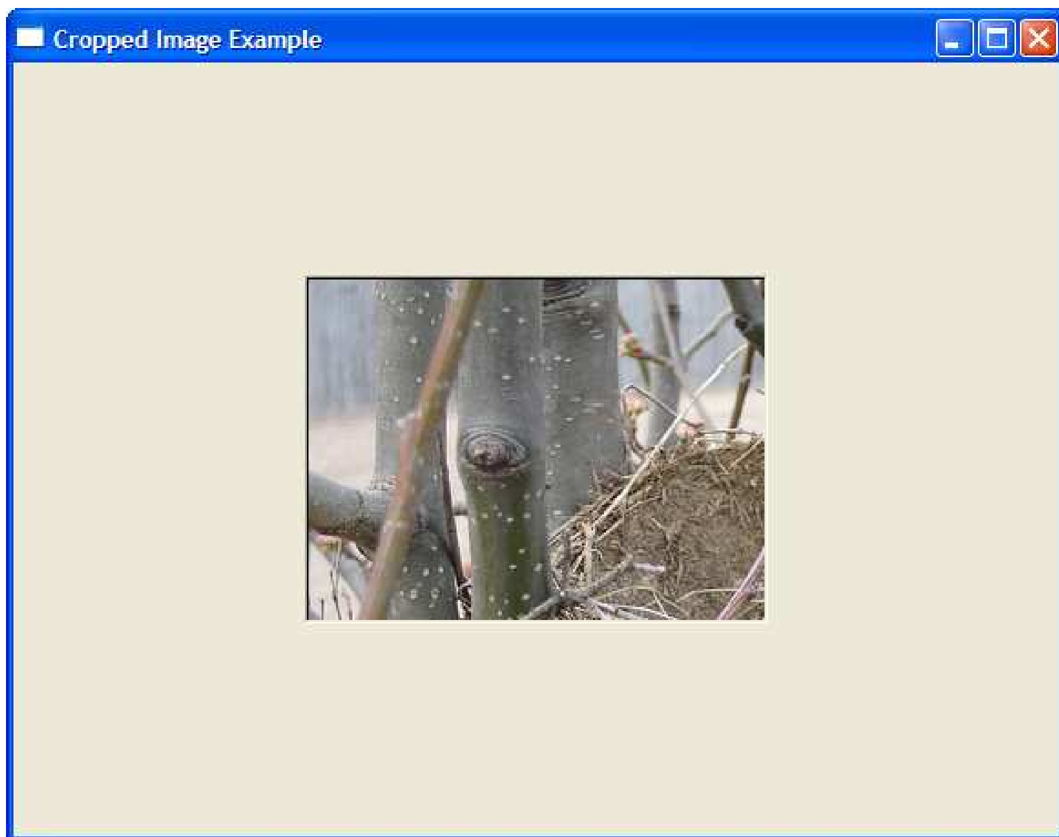



Figure 11.6: Resulting image after cropping

Давайте рассмотрим другой метод манипулирования изображением. Вы можете создать зеркальное отображение текущего изображения, используя метод **mirror()**:

```
@image.mirror(false, true)
```

Первый параметр **mirror()** указывает, должно ли изображение быть зеркально отражено в горизонтальном направлении (то есть, зеркально отражено слева направо). Второй параметр указывает, должно ли изображение быть вертикально зеркально отражено. В этом примере мы зеркально отражаем изображение только в вертикальном направление. Рисунок 11.8, на странице 155, показывает что фотография гнезда птицы было зеркально отражено вертикально.

Мы ранее отметили что, если Вы хотите быть в состоянии управлять client-side pixel buffer после того, как изображение создано, Вы должны убедиться что установлена опция **IMAGE_KEEP**, когда Вы создаете изображение. Также важно, чтобы отметить это, когда Вы впоследствии управляете изображением, Вы должны сказать FOX обновить серверное представление изображения, вызывая метод **render()**. Многие API для манипулирования изображением, включая **scale()**, **crop()** и **mirror()** автоматически вызывают **render()** для Вас после того, как они закончили перестраивать пиксели. Другие API, такие как *blend()* и *gradient()*, не делают этого, и Вам решать вызывать ли *render()* для изображения после использования этих API.

В дополнение к масштабированию, обрезке и зеркальному отражению изображений, FOX обеспечивает **API** для вращения, сдвига, исчезания и смешивания изображения; см. документацию по **API FXImage**. Это - довольно полезный набор инструментов, но для чего-либо более сложного Вы, вероятно, будете использовать вспомогательную библиотеку RMagick Тима Хантера.

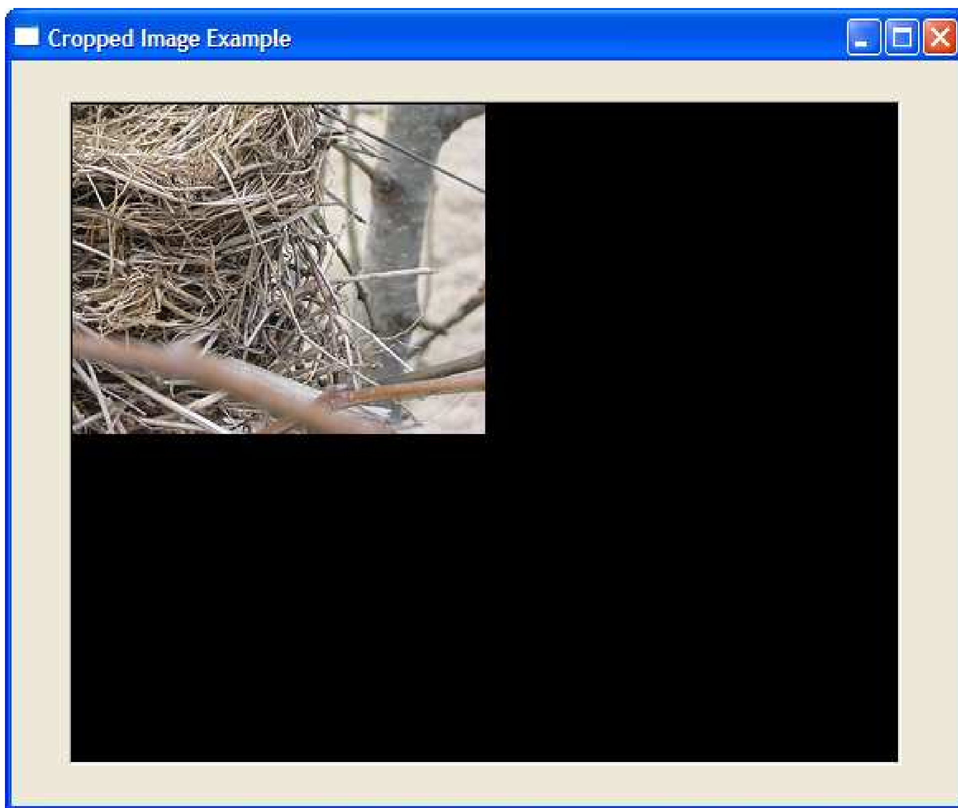



Figure 11.7: Cropping gone bad

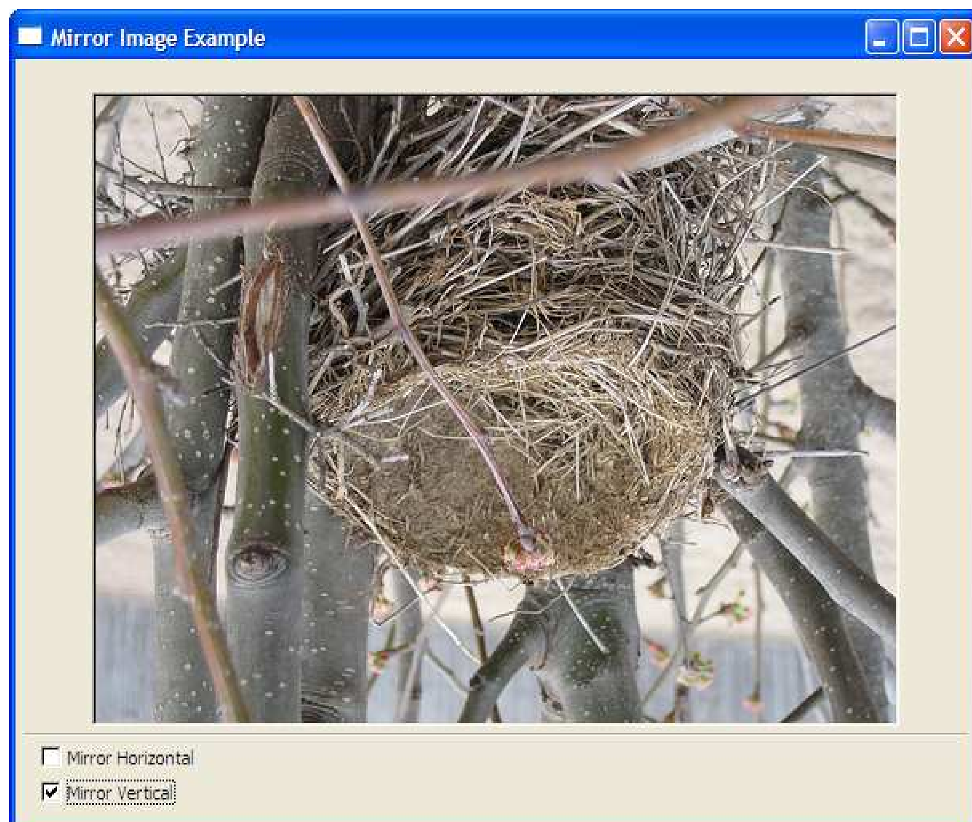



Figure 11.8: A mirror image

До сих пор мы рассматривали только один аспект того, как использовать растровые изображения, чтобы улучшить наши приложения. В следующем разделе мы рассмотрим,

как использовать изображения в качестве иконок, чтобы обеспечить маленькие элементы декора для меток, кнопок и других виджетов.

## Создание и отображение иконок

Основное различие между изображениями и иконками то, что некоторые части иконок обрабатываются как прозрачные. Вы определяете цвет прозрачности для иконки, и затем когда FOX рисует эту иконку, под этими областями показываются цвета под иконкой. Иконки также обычно меньше в размер чем Ваши изображения, но во всех кроме нескольких ситуаций нет никаких ограничений на размер значка. Другое важное различие состоит в том, что иконки могут использоваться в намного большем количестве мест чем просто изображение. Вы уже видели, как Вы можете использовать значки в качестве художественных оформлений для меток, кнопок и различных элементов списка. В Главе 13, «Расширенное Управление Меню», на странице 187, Вы увидите, что их также можно связать с пунктами меню.

Figure 11.9: Guessing the transparency color for BMP icons

Как это имело место с изображениями, самый легкий способ начать использовать иконки - это создать одно или несколько изображений при помощи одного из подклассов **FXIcon**. Например, Вы можете создать значок из файла GIF в одной строке:

```
gif_icon = FXGIFIcon.new(app, File.open("fxruby.gif" , "rb").read)
```

Единственная хитрая часть построения значка это то, что большинство графических форматов (кроме GIF) не поддерживают понятие прозрачности. Например, формат файла BMP не поддерживает прозрачность вообще. Когда Вы захотите создать значок из BMP изображения, Вы должны, по крайней мере, дать FOX подсказку, что цвет — один. Самый легкий способ сделать это использовать опцию **IMAGE_ALPHAGUESS**:

```
r_icon = FXBMPIcon.new(app,
      File.open("Rasl.bmp" , "rb").read,
      :opts => IMAGE_ALPHAGUESS)
```

На рисунке 11.9 приведён пример опции **IMAGE_ALPHAGUESS** в действии. Когда Вы определите опцию **IMAGE_ALPHAGUESS**, FOX предположит цвет прозрачности, основанный на цветах четырех углов изображения. И это обычно - довольно хорошее предположение; для этого примера, все четыре угла исходных изображений BMP были белыми, таким образом, это правильное предположение, что белые области должны быть сделаны прозрачными.

Если алгоритм **IMAGE_ALPHAGUESS** не смог определить правильное предположение, Вы можете явно определить, какое значение цвета должно быть прозрачным, передав значение в опцию **IMAGE_ALPHACOLOR**:

```
red_transp = FXBMPIcon.new(app,
      File.open("shapes.bmp" , "rb").read,
      :opts => IMAGE_ALPHACOLOR)
red_transp.transparentColor = FXRGB(255, 0, 0)
```

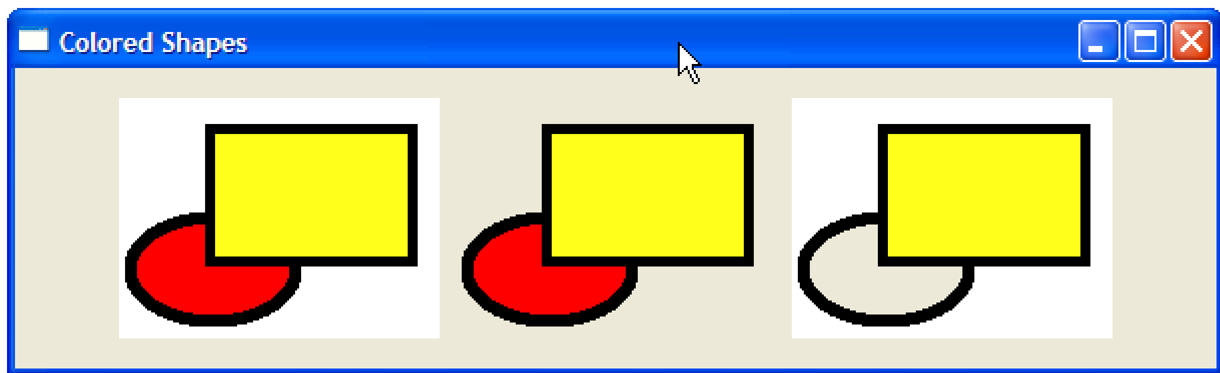



Figure 11.10: Fixing the transparency color for BMP icons

Рисунок 11.10 показывает три различных результата для цвета прозрачности из того же самого значка BMP. Первый результат, слева, показывает то, что мы получаем если мы ничего не говорим о цвете прозрачности: изображение просто непрозрачно. Средний результат показывает, что мы получаем, если мы используем опцию **IMAGE_ALPHAGUESS**: четыре угла изображения являются белыми, это, как предполагается, является цветом прозрачности. Третий результат, справа, показывает то, что мы получаем, когда мы определяем опцию **IMAGE_ALPHACOLOR** и цвет прозрачности красный (как в предыдущем коде).

На другом конце спектра формат PNG, который включает альфа-канал для того, чтобы определить степень прозрачности для каждого отдельного пикселя. Пиксели могут быть абсолютно непрозрачными, полностью прозрачными, или где-нибудь посередине. FOX рисует этот вид значков предполагая, что любой пиксель не абсолютно прозрачен (и таким образом имеет ненулевое альфа-значение) и должен быть видимым. Другая сторона этого — это то, что Вы не можете вывести на экран частично прозрачные области в значках, но однако Вы не ограничены единственным цветом прозрачности как в большинстве других форматов.

## Еще Одна Вещь

Мы обсуждали этот момент в Разделе 7.7, *«Client-Side us. Server-Side Objects»* на странице 95. Если Вы создаете изображение или иконку динамически, после того, как программа запущена, убедитесь что вызвали **create()**, чтобы соединить это с серверным ресурсом. Иначе, Ваша программа откажет. Эта проблема возникает в различных ситуациях, но особенно это распространено при добавлении новых элементов к списку во время выполнения.

Предположите, что у Вас есть виджет **FXListBox**, к которому элементы добавляются во время выполнения. ListBox содержит список имен пользователей, и каждую запись в списке связана с иконкой, которая указывает на состояние пользователя:

```
def add_user(user_name, status)
      status_icon = make_status_icon(status)
      status_icon.create
      users_listbox.appendItem(user_name, status_icon)
end
```

В этом примере из нашей гипотетической программы, приложение может вызвать метод **add_user()**, чтобы добавить нового пользователя с данным состоянием на лету. Прежде, чем добавить элемент списка, мы вызываем метод **make_status_icon()**, чтобы создать новый экземпляр **FXIcon**, который графически показывает состояние пользователя. Ключ здесь в том, что надо вызвать **create()** на объекте значка прежде, чем мы свяжем это с элементом списка. Отметьте что наша реализация **make_status_icon()** может кэшировать объекты иконок, если это имеет смысл. Нет никакого вреда в вызове **create()** на уже созданном ресурсе, но Вы определенно должны гарантировать, что ресурс был создан прежде чем начнёт использоватся.

Построение приложения GUI примерно не помещает все виджеты в правильных местах и проводном соединении их вместе. Повсеместное использование красивых иконок и необычных шрифтов не является средством от плохо разработанного алгоритма взаимодействия с пользователем, но с соответствующее (и я надеюсь сделанное со вкусом), использование пользовательских шрифтов, курсоров, картинок и иконок, может улучшить пользовательский интерфейс, чтобы сделать его более визуально приятным конечному пользователю.

Поговорим о помещении всех виджетов в правильных местах. О том, как Вы можете использовать менеджеры расположения для размещения виджетов в пользовательском интерфейсе. Давайте сделаем это в следующей главе.

## Глава 12

# Менеджеры размещения

Менеджеры размещения - объекты, которые ответственны за расположение и размеры других виджетов. На его поверхности это может походить довольно просто, и истина в том, что в хорошо разработанном пользовательском интерфейсе, пользователи приложения не будут задумываться о их роли.

Менеджеры размещения дают Вам способы определить не только как размеры отдельных виджетов и определить их позиции, но также и как группы виджетов согласованы и расположены друг относительно друга. Они дают Вам инструменты, чтобы заставить всё соответствовать. Для тех случаев, где объекты не будут (или не должны) появляться на экране одновременно, менеджеры по расположению предоставляют Вам опции для того, чтобы вывести на экран различные части пользовательского интерфейса в разное время. Например, некоторые менеджеры размещения специального назначения позволяют Вам загружать различные части приложения, в то время как другие позволяют Вам прокручивать различные части окна для просмотра.

Менеджеры расположения также помогают Вам решить некоторые менее очевидные проблемы. Например, один из наиболее хитрых аспектов создания межплатформенного приложения - то, что некоторые ресурсы, такие как шрифты, могут быть изменены на лету. Изменения данного типа могут оказать значительное влияние на расположении частей пользовательского интерфейса которые отображают текст с использованием этих шрифтов. Менеджеры по расположению могут помочь Вам минимизировать воздействие этих изменений.

В этой главе мы начнём рассматривать модель размещения, фундаментальный алгоритм компоновки, который используют менеджеры размещения в FOX. **FXPacker** - симпатичный комплексный менеджер компоновки, и хотя мы будем использовать его в нашем начальном обсуждении модели компоновки, практически Вы не будете использовать **FXPacker** непосредственно. Вместо этого Вы будете используйте один из его подклассов, таких как **FXHorizontalFrame** или **FXVerticalFrame**.

Мы рассмотрим это и затем продолжим обсуждение основ компановки, исследуя, как создать более сложные разметки вложений, как создать фиксированные разметки и, (почему бы и нет) как изменить *padding* и *spacing*, чтобы получить правильный результат.

Как только мы поймём основные принципы, мы обратим внимание на некоторые из менеджеров компоновки более специального назначения. Мы уже применяли некоторые из этих менеджеров компоновки, такие как **FXMatrix**, **FXScrollWindow**, и **FXSplitter**, когда создавали приложение Picture Book. Мы более внимательно рассмотрим их и затронем некоторые тонкости в их использовании.
Мы также исследуем менеджер компоновки **FXTabBook**, с помощью которого Вы сможете

создать снабженные вкладками представления в стиле ноутбука. Мы начнём главу с общего взгляда на то, как решить некоторые общие проблемы компоновки в FOX.

## 12.1 Понимание Модели Компоновки

Когда Вы впервые создаете окно **FXPacker**, оно походит на пустое поле. Первый дочерний виджет, который Вы добавляете, будет привязан к одной из его четырех сторон.

Вы определяете, к какой стороне Вы хотите привязать его, передавая подсказку в одном атрибутов **LAYOUT_SIDE_TOP**, **LAYOUT_SIDE_RIGHT**, **LAYOUT_SIDE_BOTTOM**, или **LAYOUT_SIDE_LEFT**:

```
packer = FXPacker.new(self, :opts => LAYOUT_FILL)
child1 = FXButton.new(packer, "First" ,
      :opts => BUTTON_NORMAL|LAYOUT_SIDE_BOTTOM)
```

Здесь, мы определили, что первый дочерний виджет - кнопка, должен быть расположен вдоль нижней стороны компоновщика. По умолчанию компоновщик пытается выровнять эту кнопку по левой стороне. Мы можем переопределить это так, чтобы кнопка вместо этого центрировалась между левой и правой сторонами:

```
child1 = FXButton.new(packer, "First" ,
      :opts => BUTTON_NORMAL|LAYOUT_SIDE_BOTTOM|LAYOUT_CENTER_X)
```

Если мы хотим, чтобы кнопка была выровнена по правой стороне, мы можем заменить **LAYOUT_CENTER_X** на **LAYOUT_RIGHT**:

```
child1 = FXButton.new(packer, "First" ,
      :opts => BUTTON_NORMAL|LAYOUT_SIDE_BOTTOM|LAYOUT_RIGHT)
```

До сих пор упаковщик только изменяет местоположение кнопки с размерами по умолчанию. Мы можем запросить, чтобы кнопка была расширена горизонтально настолько, насколько возможно:

```
child1 = FXButton.new(packer, "First" ,
      :opts => BUTTON_NORMAL|LAYOUT_SIDE_BOTTOM|LAYOUT_FILL_X)
```

Теперь, что, если мы хотим расположить две кнопки по нижней стороне. Кажется, что мы должны сделать примерно так:

```
packer = FXPacker.new(self, :opts => LAYOUT_FILL)
child1 = FXButton.new(packer, "Bottom-Right" ,
      :opts => BUTTON_NORMAL|LAYOUT_SIDE_BOTTOM|LAYOUT_RIGHT)
child2 = FXButton.new(packer, "Bottom-Left" ,
      :opts => BUTTON_NORMAL|LAYOUT_SIDE_BOTTOM|LAYOUT_LEFT)
```

Атрибут **LAYOUT_SIDE_BOTTOM** говорит компоновщику, что мы хотим обе кнопки расположить напротив нижней стороны. Мы определяем **LAYOUT_RIGHT** для первой кнопки и **LAYOUT_LEFT** для второй. Мы знаем что размер по умолчанию кнопок является достаточно маленьким, чтобы они поместились. И что же происходит, когда Вы выполняете этот пример?

Предполагали ли Вы, что ничего не получиться? На рисунке 12.1 показано, как это выглядит под Windows, но расположение не совсем то, что мы ожидали. Первая кнопка правильно расположена против базового края и выровнена по правой стороне. Вторая кнопка правильно выровнена, но находится сверху, выше первой кнопки. Компоновщик расположив виджет считает всё остающееся пространство (иначе "полость") так же занятым. Это - истина, даже если Вы не используете **LAYOUT_FILL_X** или **LAYOUT_FILL_Y**.

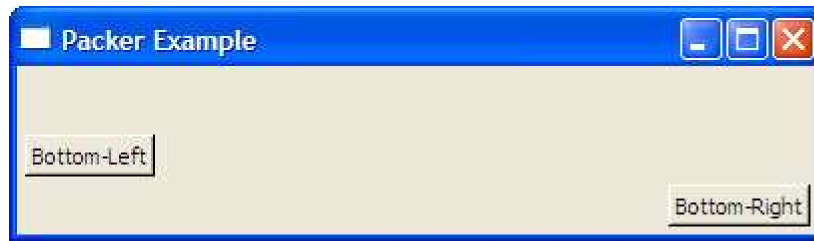



Figure 12.1: Not quite what we expected?

Давайте добавим еще несколько виджетов, расположенных против других сторон полости, чтобы видеть, как они размещаются:

```
packer = FXPacker.new(self, :opts => LAYOUT_FILL)
child1 = FXButton.new(packer, "Bottom-Right" ,
      :opts => BUTTON_NORMAL|LAYOUT_SIDE_BOTTOM|LAYOUT_RIGHT)
child2 = FXButton.new(packer, "Bottom-Left" ,
      :opts => BUTTON_NORMAL|LAYOUT_SIDE_BOTTOM|LAYOUT_LEFT)
child3 = FXLabel.new(packer, "Top" ,
      :opts => FRAME_GROOVE|LAYOUT_SIDE_TOP|LAYOUT_FILL_X)
child4 = FXLabel.new(packer, "Left-Center" ,
      :opts => FRAME_GROOVE|LAYOUT_SIDE_LEFT|LAYOUT_CENTER_Y)
```

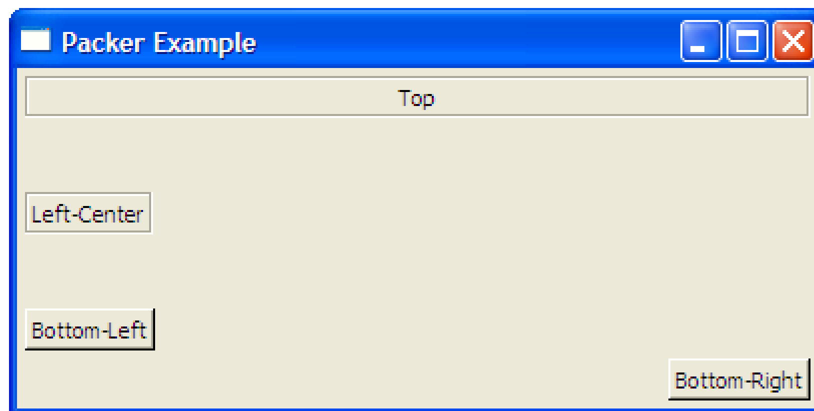



Figure 12.2: After adding more widgets

На рисунке 12.2 показано, как это выглядит под Windows. Это поможет понять, что произошло, см. рисунок 12.3, на следующей странице. Пунктирные линии указывают на пробелы, занятые каждым из дочерних виджетов. Отметьте что мы добавили четвертый дочерний виджет ("Лево-центральная" метка).

Но давайте не забывать нашу более раннюю цель относительно размещения кнопок в нижнем левом и нижнем правом углу компоновщика. Как сделать так, как нам хачется? Для любого нетривиального расположения вкладывать компоновщики один в другой. И это приводит нас в обсуждение следующей группы менеджеров по расположению, горизонтальные и вертикальные фреймы.

## Создание Простых Разметок с Горизонтальными и Вертикальными Фреймами

Компоновщики **FXHorizontalFrame** и **FXVerticalFrame** — это подклассы **FXPacker**, которые накладывают некоторые дополнительные ограничения на модель компоновщика, о которой мы говорили. Как Вы можете предположить по их именам, горизонтальный фрейм располагает свои дочерние элементы горизонтально, и вертикальный фрейм располагает свои дочерние элементы вертикально.

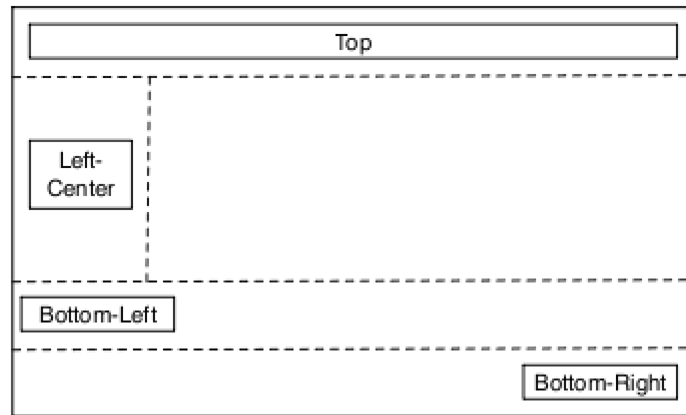



Figure 12.3: What’s going on behind the scenes

По умолчанию горизонтальный фрейм располагает свои дочерние элементы с левого края к правому, используя их размеры по умолчанию:

```
hframe = FXHorizontalFrame.new(self)
child1 = FXButton.new(hframe, "First")
child2 = FXButton.new(hframe, "Second")
child3 = FXButton.new(hframe, "Third")
```

Как это имело место с **FXPacker**, Вы можете определить более конкретно как дочерний виджет должен быть выровнен:

```
hframe = FXHorizontalFrame.new(self)
child1 =
      FXButton.new(hframe, "Right" , :opts => BUTTON_NORMAL|LAYOUT_RIGHT)
child2 =
      FXButton.new(hframe, "Left" , :opts => BUTTON_NORMAL|LAYOUT_LEFT)
```

Отметьте, что мы используем **LAYOUT_LEFT** и **LAYOUT_RIGHT**, не **LAYOUT_SIDE_LEFT** и **LAYOUT_SIDE_RIGHT**. Помните, что атрибуты компоновки (такие как **LAYOUT_SIDE_LEFT** и **LAYOUT_SIDE_RIGHT**), применяет только менеджер по расположению **FXPacker**.

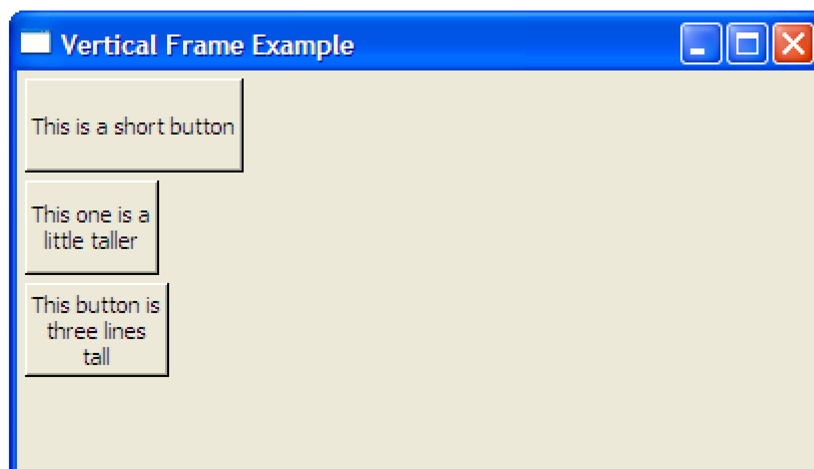



Figure 12.4: Vertical frame with uniform heights

Вы можете определить атрибут **PACK_UNIFORM_HEIGHT** для выравнивания всех виджетов во фрейме по одной высоте:

```
vframe = FXVerticalFrame.new(self, :opts => PACK_UNIFORM_HEIGHT)
child1 = FXButton.new(vframe, "This is a short button")
child2 = FXButton.new(vframe, "This one is a\nlittle taller")
child3 = FXButton.new(vframe, "This button is\nthree lines\ntall")
```

Отметьте, что параметр **PACK_UNIFORM_HEIGHT** определён для фрейма, а не для отдельных кнопок. Когда Вы выполните этот пример, Вы увидите что все три кнопки были приведены к высоте самого высокого виджета (третья кнопка). Рисунок 12.4 показывает, как это выглядит под Windows. Вы можете также установить для всех виджетов универсальную ширину.

```
vframe = FXVerticalFrame.new(self,
      :opts => PACK_UNIFORM_WIDTH|PACK_UNIFORM_HEIGHT)
child1 = FXButton.new(vframe, "This is a short button")
child2 = FXButton.new(vframe, "This one is a\nlittle taller")
child3 = FXButton.new(vframe, "This button is\nthree lines\ntall")
```

Рисунок 12.5, на следующей странице, показывает то, на что это похоже в Windows. Отметьте, что каждая из кнопок теперь столь же широка как самый широкий виджет (первая кнопка).

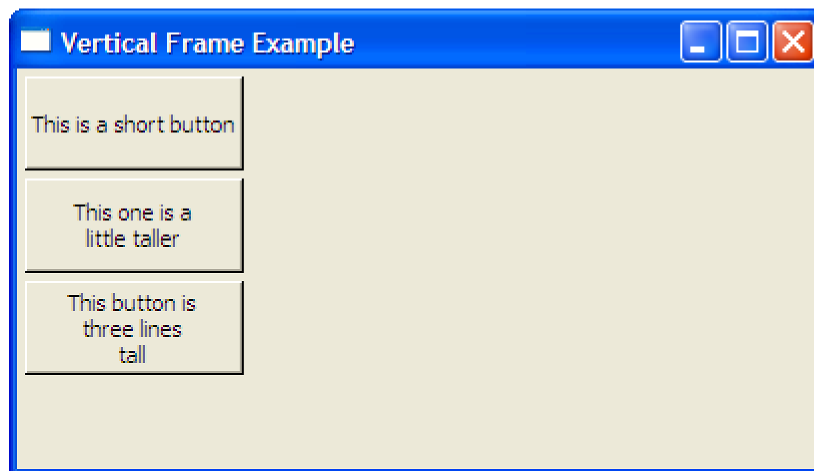



Figure 12.5: Vertical frame with uniform widths and heights

Понимание того, как модель компоновки работает и как использовать **FXHorizontalFrame** и **FXVerticalFrame**, чтобы создать простые разметки является необходимым для любого разработчика на **FXRuby**, но Вы начнете понимать действительная мощность менеджеров по расположению, когда Вы изучите, как вкладывать компоновщики друг в друга, чтобы создать более сложное форматирование.

## Вложение компоновщиков друг в друга

Давайте повторно посмотрим пример, который мы делали в предыдущем разделе, когда попробовали упаковать две кнопки вдоль нижней стороны компоновщика. Чтобы достигнуть этого расположения, мы вставим эти две кнопки в горизонтальный фрейм и затем поместим горизонтальный фрейм в компоновщик:

```
packer = FXPacker.new(self, :opts => LAYOUT_FILL)
hframe = FXHorizontalFrame.new(packer, :opts => LAYOUT_SIDE_BOTTOM)
child1 = FXButton.new(hframe, "Bottom-Right" ,
      :opts => BUTTON_NORMAL|LAYOUT_RIGHT)
child2 = FXButton.new(hframe, "Bottom-Left" ,
      :opts => BUTTON_NORMAL|LAYOUT_LEFT)
```

Прежде, чем Вы выполните этот пример, посмотрите на код и удостоверьтесь, что Вы понимаете как горизонтальный фрейм вкладывается в компоновщик.

На рисунке 12.6 показано что получилось.

Это уже достаточно близко к тому, что мы задумали. Кнопки находятся теперь на одном уровне, но вместо того, чтобы быть в противоположных углах они рядом друг рядом с другом. Это произошло из-за того, что мы не определили по какой стороне выровнять их, и было выбрано выравнивание по умолчанию **LAYOUT_LEFT**. Давайте исправим это, определив параметр **LAYOUT_FILL_X** для горизонтального фрейма:

```
packer = FXPacker.new(self, :opts => LAYOUT_FILL)
hframe = FXHorizontalFrame.new(packer,
      :opts => LAYOUT_SIDE_BOTTOM|LAYOUT_FILL_X)
child1 = FXButton.new(hframe, "Bottom-Right" ,
      :opts => BUTTON_NORMAL|LAYOUT_RIGHT)
child2 = FXButton.new(hframe, "Bottom-Left" ,
      :opts => BUTTON_NORMAL|LAYOUT_LEFT)
```

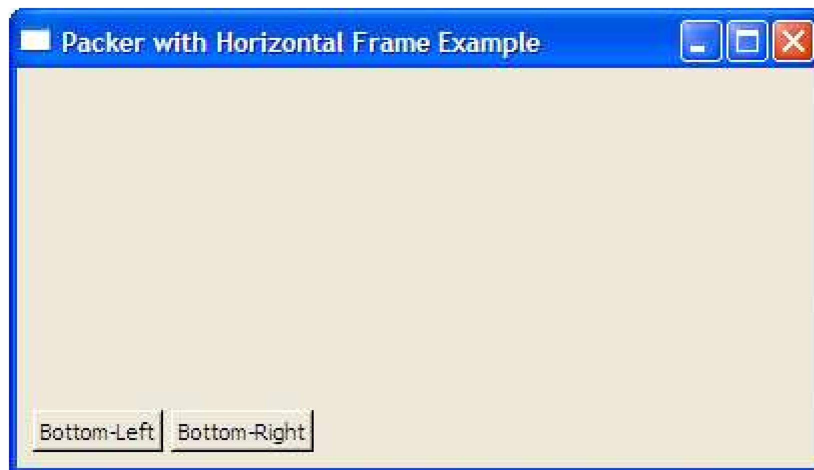



Figure 12.6: Packer with nested horizontal frame

Рисунок 12.7, показывает окончательный результат. Очевидно, это довольно простой пример по расположению вложения друг в друге, но это дает Вам пример того, как решить еще более сложное расположение. Мы повторно посетим это более подробно в Разделе 12.6, «*Стратегия Использования Различных Компоновщиков Вместе*», на странице 181.

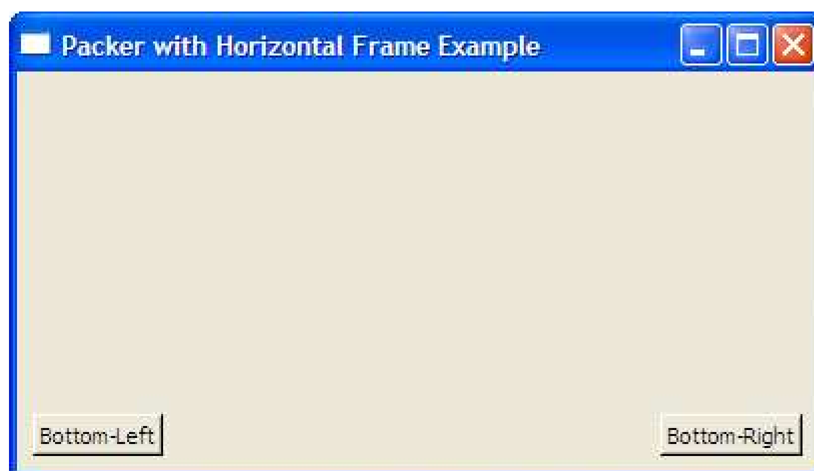



Figure 12.7: Packer with nested horizontal frame

До сих пор мы обсуждали, как создавать различные разметки без создания прямых ссылок на размеры или позиции виджетов. FOX также поддерживает фиксированные разметки, где Вы явно определите позиции и/или размеры виджетов, и мы рассмотрим как это работает ниже.

## Создание фиксированной разметки

Вы можете использовать фиксированные разметки. С этим подходом Вы можете определить позицию виджета явно вместо того, чтобы позволить компоновщику сделать это за Вас.

```
@fixed_pos_button = FXButton.new(self, "Fixed Position",
      :opts => BUTTON_NORMAL|LAYOUT_FIX_X|LAYOUT_FIX_Y,
      :x => 20, :y => 20)
```

Когда Вы будете выполняье этот пример, попытайтесь изменить размеры главного окна. Благодаря параметрам **LAYOUT_FIX_X** и **LAYOUT_FIX_Y**, Вы увидите что кнопка остается в той же самой позиции, относительно верхнего левого угла основного окна, независимо от того, что Вы делаете. Если Вы изменяете размеры окна так, что оно становится слишком маленьким, чтобы отобразить кнопку, кнопка будет отсечена.

Вы можете также фиксировать размер виджета, используя параметры **LAYOUT_FIX_WIDTH** и **LAYOUT_FIX_HEIGHT**:

```
@fixed_size_label = FXLabel.new(self, "Fixed Size Label",
      :opts => (LAYOUT_CENTER_X|LAYOUT_CENTER_Y|
                LAYOUT_FIX_WIDTH|LAYOUT_FIX_HEIGHT),
      :width => 120, :height => 40)
```

Это не бескомпромиссное суждение, между прочим. Если Вы только хотите фиксировать позицию **X** виджета, но отпустить другие параметры, передайте флаг **LAYOUT_FIX_X** и значение для атрибута **X**. Если Вы собираетесь определить все четыре параметра (x, y, ширину и высоту), Вы можете применить более компактный вид параметра **LAYOUT_EXPLICIT**:

```
FXLabel.new(self, "Fixed Position and Size",
      :opts => LAYOUT_EXPLICIT,
      :x => 20, :y => 20,
      :width => 120, :height => 40)
```

Это имеет тот же самый эффект как и при применении всех параметров: **LAYOUT_FIX_X**, **LAYOUT_FIX_Y**, **LAYOUT_FIX_WIDTH** и **LAYOUT_FIX_HEIGHT**.

Теперь, забудьте все, что я только что сказал Вам, потому что фиксированная разметка не является лучшим решением.

Проблема с фиксированными разметками состоит в том, что они уменьшают мобильность из Вашего пользовательского интерфейса к средам выполнения с различными характеристиками экрана. Та разметка, которая выглядит хорошо с фиксированной шириной 120 пикселов на Вашей системе будет выглядеть гигантской в системе где пользователь применяет меньший шрифт чем тот, который Вы использовали. Хуже, если пользовательский шрифт больше чем тот, который Вы использовали, часть виджета строки может быть усечённой и нечитабельной. Чтобы увидеть это в действии, запустите пример программы, и увеличивайте или уменьшайте размер шрифта используя виджет spinner внизу экрана.

Подобная проблема имеет отношение к размеру дисплея. Если Вы, разрабатываете и запускаете приложение в полноэкранном режиме на 1024x768 и используете фиксированное размещение, выполнение программы при другом разрешении экрана почти наверняка будет пагубным. Если Вы не используете некоторые элементы,

которые всегда должны быть одного размера, например изображение фиксированного размера или иконка — Вы должны избегать использования фиксированной разметки в Ваших приложениях.

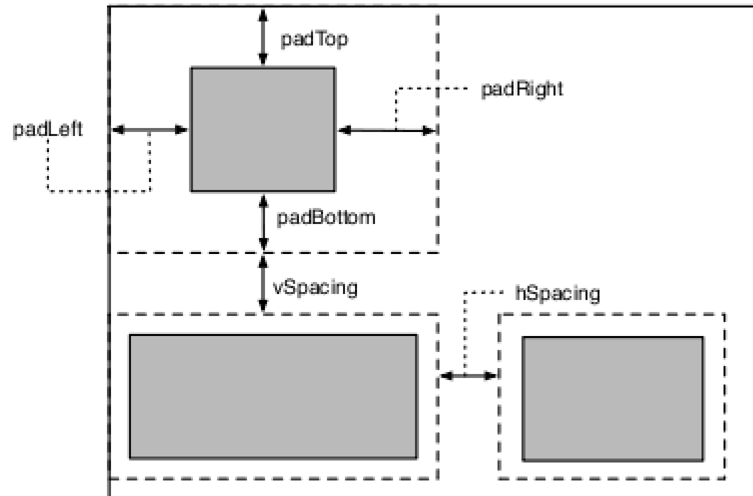



Figure 12.8: Padding and spacing

## Добавление Последних штрихов с Дополнением и Интервалом

Вероятно Вы заметили, что всегда существует некоторый промежуток между виджетом и контейнером в котором он находится. Также существуют интервалы между смежными дочерними виджетами в контейнере. Очевидно, **FOX** применяет некоторые разумные значения по умолчанию, чтобы убедиться, что расположение выглядит хорошо, и во многих случаях это не что Вы желаете видеть. В некоторых случаях, Вы должны самостоятельно установить разрывы между виджетами, и по этой причине мы собираемся рассказать о параметрах *padding* и *spacing*.

Рисунок 12.8 иллюстрирует, как используются *padding* и *spacing* во время компоновки. **Padding** для виджета - количество пространства вокруг его краев. Дополнение значения по умолчанию для большинства виджетов это составляет 2 пикселя по каждой стороне, но Вы можете изменить это значение для любой или для всех сторон. Вы можете определить дополнительные значения для виджета во время его создания, используя некоторую комбинацию ключей **:padLeft**, **:padRight**, **:padTop** и **:padBottom**:

```
spinner = FXSpinner.new(self, 4,
      :opts => SPIN_NORMAL|LAYOUT_SIDE_TOP|LAYOUT_LEFT,
      :padLeft => 0, :padTop => 0)
```

Довольно распространено желание так установить промежуток на всех четырех сторонах, чтобы обнулить его, чтобы не было никакого внутреннего свободного пространства. Для этих ситуаций Вы можете определить только **:padding**:

```
spinner = FXSpinner.new(self, 4,
      :opts => SPIN_NORMAL|LAYOUT_SIDE_TOP|LAYOUT_LEFT,
      :padding => 0)
```

Вы можете также изменить padding для виджета после того, как он был создан, используя атрибуты **padLeft**, **padRight**, **padTop** и **padBottom**:

```
spinner = FXSpinner.new(self, 4,
      :opts => SPIN_NORMAL|LAYOUT_SIDE_TOP|LAYOUT_LEFT)
spinner.padLeft = 0
spinner.padTop = 0
```

Параметры пространства по горизонтали и вертикали для компоновщика означают, как много свободного места будет между смежными дочерними виджетами. Вы можете определить эти значения, используя ключи **:hSpacing** и **:vSpacing** во время создания:

```
groupbox = FXGroupBox.new(self, "No horizontal or vertical spacing",
      :hSpacing => 0, :vSpacing => 0)
```

Отметьте, что *spacing* применяется только между смежными виджетами, он не используется для виджетов вдоль сторон компоновщика. Например, левый край крайнего левого виджета в горизонтальной области будет смещен только его внутренним дополнением (более определенно, его значением **padLeft**).

Как это имело место с *padding*, Вы можете также изменить значение *spacing* для контейнера после того, как он был создан:

```
groupbox = FXGroupBox.new(self, "No horizontal or vertical spacing")
groupbox.hSpacing = 0
groupbox.vSpacing = 0
```

Должно быть немного сложно приобрести навык работы с *padding* и *spacing* и понять как они взаимодействуют с друг другом. Главное помнить что, Вы определяете *spacing* в компоновщике (контейнер) и *padding* в дочерних виджетах. Так как все компоновщики могут быть дочерними элементами других компоновщиков, это означает, что Вы можете также определить *padding* для компоновщиков.

Теперь, когда мы рассмотрели все основы работы с компоновщиками, мы обратим ваше внимание к компоновщикам специального назначения, которые предоставляет FOX. Мы начнём с компоновщика **FXMatrix**, с которым мы ранее встречались, когда создавали приложение Picture Book.

## 12.2 Расположение Виджетов в Строках и Столбцах с FXMatrix

**FXMatrix** размещает свои дочерние виджеты в строках и столбцах. Некоторые другие инструментарии именуют этот вид расположения как "grid". Он особенно полезен для вывода на экран табличных данных вроде электронной таблицы. Однако Вы вероятно, предпочтёте использование виджета **FXTable**, который мы будем рассматривать в Разделе 9.4, «Displaying Tabular Data with FXTable», на страница 126.

Матрица может быть сконфигурирована с любым постоянным числом строк (**MATRIX_BY_ROWS**) или постоянным числом столбцов (**MATRIX_BY_COLUMNS**), и второй параметр для **FXMatrix.new** - это число строк (или столбцов).

```
matrix = FXMatrix.new(self, 3, :opts => MATRIX_BY_ROWS|LAYOUT_FILL)
```

Вы добавляете дочерние виджеты к матрице так же, как в любой другой компоновщик, передавая ссылку на матрицу как родителя для каждого дочернего виджета. Куда матрица фактически помещает эти виджеты, может удивить Вас. Рисунок 12.9 иллюстрирует расположение для виджета **FXMatrix**, сконфигурированного как **MATRIX_BY_ROWS** с тремя строками. Первый дочерний виджет, который Вы добавляете, становится первым виджетом в первой строке матрицы. Второй виджет, который Вы добавляете, становится первым виджетом во второй строке матрицы. Это продолжается, пока Вы не заполнили первый столбец матрицы (добавляя первый элемент каждой строки), и затем Вы начнёте снова со второго столбца. Если матрица сконфигурирована как **MATRIX_BY_COLUMNS**, Вы будете вместо этого заполнять строку за один раз.

| 1 | 4 | 7 | 10 |
|---|---|---|---|
| 2 | 5 | 8 | 11 |
| 3 | 6 | 9 | 12 |

Figure 12.9: Layout order for a matrix with three rows

То, как **FXMatrix** ведёт себя в ответ на изменение размеров может быть немного непривычно. Мы должны понять, как происходит размещение дочерних виджетов матрицы и как они влияют на вид матрицы.

В пределах определенной ячейки матрицы компоновка работает как ожидается. Если Вы определили **LAYOUT_FILL_X** для дочернего виджета, этот виджет будет простираться горизонтально, чтобы заполнить всю ширину матричной ячейки. Аналогично, если Вы определите **LAYOUT_CENTER_Y**, то этот дочерний элемент будет отцентрирован вертикально.

Во-первых, хорошие новости. При прочих равных условиях **FXMatrix** выделит достаточно места в каждой ячейке, чтобы гарантировать что дочерний виджет в этом расположении поместится. Но что, если матрица имеет пространство, чтобы сэкономить? Вашим первым предположением могло бы быть определение **LAYOUT_FILL_X** и/или **LAYOUT_FILL_Y** для матрицы непосредственно. Вы думаете, что всё содержимое будет простираться соответственно. Ну, это было хорошее предположение, но Вы были бы неправы.

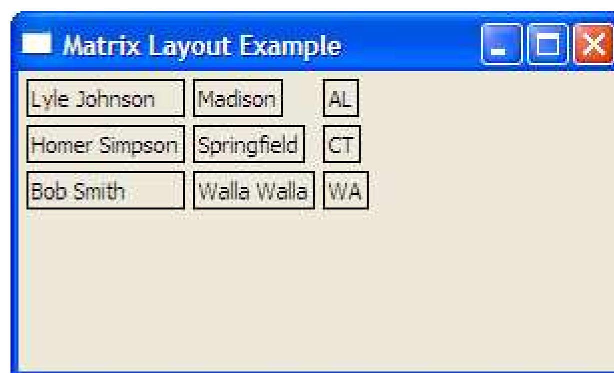



Figure 12.10: Matrix without LAYOUT_FILL_COLUMN

Сделайте эту простую матрицу с девятью дочерними виджетами расположенными в три столбца:

```
matrix = FXMatrix.new(self, 3, :opts => MATRIX_BY_COLUMNS|LAYOUT_FILL)
FXLabel.new(matrix, "Lyle Johnson" ,
      :opts => JUSTIFY_LEFT|FRAME_LINE|LAYOUT_FILL_X)
FXLabel.new(matrix, "Madison" , :opts => JUSTIFY_LEFT|FRAME_LINE)
FXLabel.new(matrix, "AL" , :opts => FRAME_LINE)
FXLabel.new(matrix, "Homer Simpson" ,
      :opts => JUSTIFY_LEFT|FRAME_LINE|LAYOUT_FILL_X)
FXLabel.new(matrix, "Springfield" , :opts => JUSTIFY_LEFT|FRAME_LINE)
FXLabel.new(matrix, "CT" , :opts => FRAME_LINE)
FXLabel.new(matrix, "Bob Smith" ,
      :opts => JUSTIFY_LEFT|FRAME_LINE|LAYOUT_FILL_X)
FXLabel.new(matrix, "Walla Walla" , :opts => JUSTIFY_LEFT|FRAME_LINE)
FXLabel.new(matrix, "WA" , :opts => FRAME_LINE)
```

Я добавил стиль фрейма **FRAME_LINE** для каждой из меток так, что Вы можете более ясно видеть границы каждого из этих виджетов. Рисунок 12.10 показывает, что получилось. Когда Вы выполняете этот пример, Ваша первая реакция могла бы состоять в том, чтобы подозревать это **FXRuby** не соблюдает атрибут **LAYOUT_FILL**, который Вы определили для **FXMatrix**. В конце концов четко видно, что матрица не расширилась, чтобы заполнить главное окно, правильно?

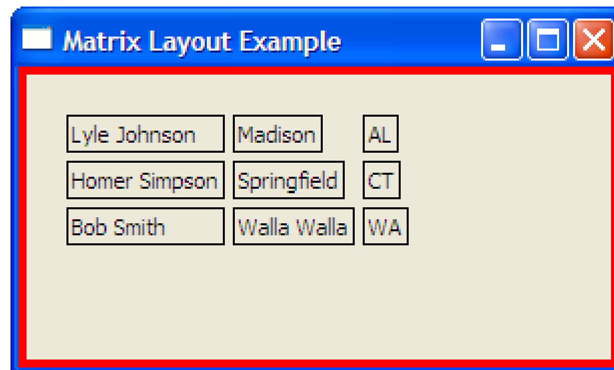



Figure 12.11: Matrix without LAYOUT_FILL_COLUMN (with red matte)

Ну, как это оказывается, матрица фактически заполняет все основное окно. Вы можете сделать изменения в коде, чтобы доказать это самостоятельно.

```
matte = FXPacker.new(self, :opts => LAYOUT_FILL)
matte.backColor = "red"
matrix = FXMatrix.new(matte, 3,
      :opts => MATRIX_BY_COLUMNS|LAYOUT_FILL, :padding => 20)
```

Представляя промежуточный **FXPacker** с красным фоном и помещая его в матрицу, мы можем видеть, что матрица действительно заполняет все свободное место (см. рисунок 12.11). Так, о чём это говорит?

Ключ к получению отдельных строк и столбцов **FXMatrix**, простирающихся на всё свободное место, это использование **LAYOUT_FILL_ROW** и **LAYOUT_FILL_COLUMN**. Если все виджеты в столбце определены с **LAYOUT_FILL_COLUMN**, тогда тот столбец будет простираться горизонтально.

```
matrix = FXMatrix.new(self, 3, :opts => MATRIX_BY_COLUMNS|LAYOUT_FILL)
FXLabel.new(matrix, "Lyle Johnson" ,
      :opts => JUSTIFY_LEFT|FRAME_LINE|LAYOUT_FILL_X|LAYOUT_FILL_COLUMN)
FXLabel.new(matrix, "Madison" ,
      :opts => JUSTIFY_LEFT|FRAME_LINE|LAYOUT_FILL_COLUMN)
FXLabel.new(matrix, "AL" , :opts => FRAME_LINE)
FXLabel.new(matrix, "Homer Simpson" ,
      :opts => JUSTIFY_LEFT|FRAME_LINE|LAYOUT_FILL_X|LAYOUT_FILL_COLUMN)
FXLabel.new(matrix, "Springfield" ,
      :opts => JUSTIFY_LEFT|FRAME_LINE|LAYOUT_FILL_COLUMN)
FXLabel.new(matrix, "CT" , :opts => FRAME_LINE)
FXLabel.new(matrix, "Bob Smith" ,
      :opts => JUSTIFY_LEFT|FRAME_LINE|LAYOUT_FILL_X|LAYOUT_FILL_COLUMN)
FXLabel.new(matrix, "Walla Walla" ,
:     opts => JUSTIFY_LEFT|FRAME_LINE|LAYOUT_FILL_COLUMN)
FXLabel.new(matrix, "WA" , :opts => FRAME_LINE)
```

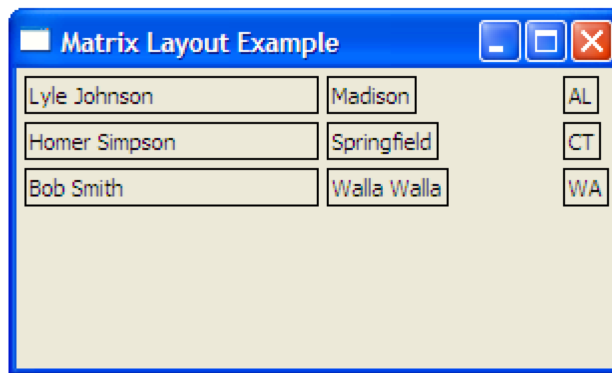



Figure 12.12: Matrix with LAYOUT_FILL_COLUMN on first column

Рисунок 12.12 показывает, как наш пример выглядит, когда **LAYOUT_FILL_COLUMN** определен для каждой из меток в первом столбце. Кроме того, если все виджеты во всех столбцах будут иметь этот атрибут, тогда всё пространство будет разделено пропорционально.

В следующем разделе мы будем рассматривать другой компоновщик, с которым мы уже встречались при создании приложения Picture Book, это **FXSplitter**.

## 12.3 Динамическое изменение размеров разметки с компоновщиком Splitter

Компоновщики, о которых мы говорили до сих пор, осуществляют размещение автоматически. Вы определяете соответствующее подсказки расположения, они определяют подходящие размеры и расположение для всех дочерних виджетов, без вмешательства с Вашей стороны. В некоторых случаях, однако, полезно предоставить пользователю инструменты для изменения размеров отдельных частей пользовательского интерфейса в интерактивном режиме, и компоновщик **FXSplitter** предназначен для этой цели.

Мы получили введение в класс **FXSplitter** в Главе 6, *Manage Multiple Albums*, на странице 62, когда мы использовали это, чтобы вывести на экран список альбомов и область просмотра рядом. Практически, **FXSplitter** обычно будет содержать только два дочерних окна, "левую" и "правую" подпанели (или верхнюю и нижнюю подпанели, если это - вертикальный разделитель). Фактически нет ограничение на число дочерних окон в разделителе, и разделитель позволит Вам изменить их размеры соответственно.

По умолчанию разделение слева направо; то есть, первый дочерний элемент **FXSplitter** располагается слева от разделителя, второй дочерний элемент справа (и так далее, если есть больше чем два дочерних элемента). Вы можете сконфигурировать разделение от верха к низу, передавая флаг **SPLITTER_VERTICAL**:

```
splitter.splitterStyle |= SPLITTER_VERTICAL
```

Вы можете изменить место, выделенное на любой стороне от разделения, захватывая дескриптор разделителя и перетаскивание его поперек. В то время как Вы, перетаскиваете дескриптор разделителя, полупрозрачная линия будет проведена через окно, чтобы показать Вам, куда разделение будет помещено, когда Вы остановите перетаскивание. Когда Вы отпускаете разделитель, смежные подпанели изменяются согласно новому расположению.

Если разделитель сконфигурирован в режиме отслеживания (tracking), панели по обе стороны от разделителя изменяются динамически, в то время как Вы перетаскиваете разделитель:

```
splitter.splitterStyle |= SPLITTER_TRACKING
```

Это немного более дорого, в вычислительном отношении, динамически перерисовывать панели.

Обычно, Вы будете устанавливать начальные размеры подпанелей к некоторым значениям по умолчанию и затем позволять пользователю изменять размеры их как необходимо. Важно сохранить размеры подпанелей в реестре, когда приложение завершается и затем восстановить их следующий раз когда приложение запускается. Передовая практика для того, как это сделать должна записать размер первой панели разделителя в реестре прежде, чем выйти из приложения:

```
app.reg.writeIntEntry("Settings" , "splitSize" , @splitter.getSplit(0))
app.exit(0)
```

*We don't cover the FOX registry in this book, but you can read more about it at http://www.fox-toolkit.org/registry.html.*

Затем, считайте размер разделения назад в метод **create()** Вашего главного окна и используйте это, чтобы сконфигурировать разделитель, прежде, чем показать основное окно:

```
def create
      super
      @splitter.setSplit(0, app.reg.readIntEntry("Settings" , "splitSize"))
      show(PLACEMENT_SCREEN)
end
```

Этот пример предполагает, что Ваш разделитель содержит только две подпанели. Очевидно, если Ваш разделитель будет содержать больше чем две подпанели, то Вы должны сохранить (и восстановить) размеры всех кроме последней.

Далее рассмотрим компоновщик **FXScrollWindow**.

## 12.4 Managing Large Content with Scrolling Windows

Мы рассматривали класс **FXScrollWindow** в Главе 5, «Display an Entire Album» на странице 43, когда мы использовали его в качестве для представления альбома. Как мы узнали, **FXRuby** обеспечивает класс **FXScrollWindow** как своего рода класс декоратора для других компоновщиков. Когда Вы создаете виджет как дочерний элемент окна прокрутки, окно прокрутки заботится об отображении горизонтальной и вертикальной прокрутки, и это гарантирует, что нужная часть контента будет видима когда пользователь просматривает это путем прокрутки.

Вы должны добавить только одно дочернее окно к **FXScrollWindow**. Почти всегда, это дочернее окно будет другим компоновщиком. Внутри, окно прокрутки создаст несколько дополнительных дочерних виджетов (для вертикали и горизонтальных полос прокрутки). По этой причине Вы должны всегда обращаться к содержимому окна прокрутки, используя атрибут **contentWindow**.

В большинстве случаев Вы будете добавлять своё содержимое в окно прокрутки и затем позволите пользователю решать, к какой части контента они хотят перейти, используя полосы прокрутки. Если необходимо программно скорректировать область просмотра окна прокрутки так, чтобы показать интересующую область, Вы можете использовать метод **setPosition()**, чтобы сделать это, но это немного сложно. Координаты, которые Вы должны передать в **setPosition()** x и y - это координаты верхнего левого угла содержимого, относительно верхнего левого угла окна области просмотра. Практически Вы будете всегда передавать отрицательные величины в **setPosition()**.

Так, например, если Вы хотите скорректировать контент так, чтобы в области просмотра окна прокрутки видна была часть содержимого, чей верхний левый угол в позиции (100, 100), Вы должны передать (-100,-100) в **setPosition()**:

```
scroll_window.setPosition(-100, -100)
```

Если Вы хотите переместить контент так, чтобы он центрировался в пределах область просмотра, Вы должны учесть различие в размерах между содержимым и областью просмотра. Следующая часть кода должна работать при любых обстоятельствах:

```
x = 0.5*(@scroll_window.contentWindow.width -
      @scroll_window.viewportWidth)
y = 0.5*(@scroll_window.contentWindow.height -
      @scroll_window.viewportHeight)
@scroll_window.setPosition(-x, -y)
```

## 12.5 Organizing Windows with Tabbed Notebooks

Компоновщик **FXTabBook** называют так, потому что это напоминает книгу снабженную вкладками. Когда Вы щелкаете по одной из вкладок, страницы связанные с этой вкладкой повышаются до вершины стека страниц, и хотя вся строка вкладок всегда видима, только одна страница (верхняя страница) в ноутбуке видима в это время. **FXTabBook** часто используют в диалоговых окнах, которые позволяют пользователю изменять настройки приложения; каждая вкладка в диалоговом окне представляет различную категорию настроек. Это также применяется в редактировании текста или текстовых процессорах, где каждая вкладка отображает различные документы.

Не очень трудно добавить снабженный вкладками ноутбук к Вашему приложению. Начните с создания виджета **FXTabBook**:

```
tabbook = FXTabBook.new(self, :opts => LAYOUT_FILL)
```

Затем, добавьте элементы вкладок и страницы во вкладки, образуя пары. Элемент вкладки это экземпляр класса **FXTabItem**. Страниц во вкладке - это обычно некоторый компоновщик, но это может фактически быть любой вид виджета.

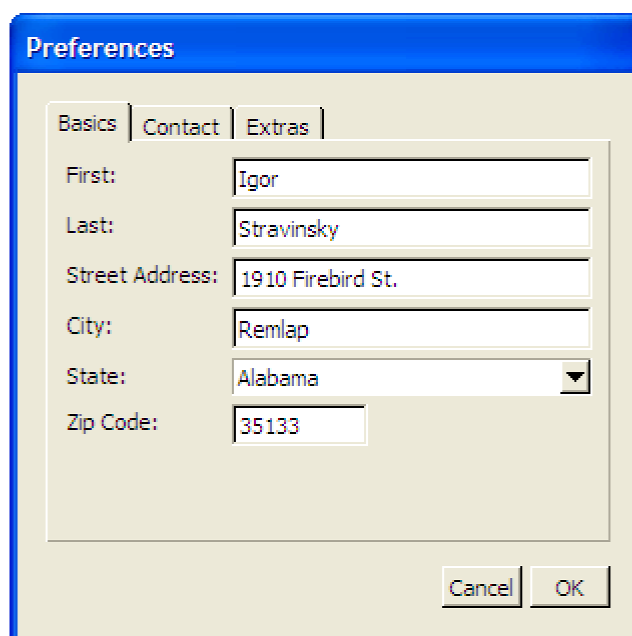



Figure 12.13: A preferences dialog box that uses a tabbed notebook

```
basics_tab = FXTabItem.new(tabbook, " Basics ")
basics_page = FXVerticalFrame.new(tabbook,
      :opts => FRAME_RAISED|LAYOUT_FILL)
contact_tab = FXTabItem.new(tabbook, " Contact ")
contact_page = FXVerticalFrame.new(tabbook,
      :opts => FRAME_RAISED|LAYOUT_FILL)
extras_tab = FXTabItem.new(tabbook, " Extras ")
extras_page = FXVerticalFrame.new(tabbook,
      :opts => FRAME_RAISED|LAYOUT_FILL)
```

На рисунке 12.13 показано, на что похож этот **FXTabBook**, когда он выводит на экран первую вкладку (вкладка Basics).

*Выборка кода, которую я привёл здесь, не показывает весь код, требуемый для создания формы, что Вы видите на рисунке. Полный исходный код доступен онлайн.*

Если Вы просто интересуетесь отображением снабженных вкладками страниц, это - действительно все что есть. Если Вы получать уведомление всякий раз, когда пользователь выбирает новую вкладку, Вы можете получать сообщение **SEL_COMMAND** которое **FXTabBook** передает к его цели после того, как новая страница выведена на экран:

```
tabbook.connect(SEL_COMMAND) do |sender, sel, data|
      puts "User selected tab number #{data}"
end
```

Вы могли бы хотеть использовать этот метод, если это не является ресурсоемким, чтобы вывести на экран контент связанный с определенной страницей в снабженном вкладками ноутбуке. Для примера, если страница выводит на экран некоторый динамически обновляемый контент, такой как счетчик или изображение с анимацией, Вы могли бы хотеть приостановить обновление того содержимого, когда пользователь выбирает другую вкладку и продолжить обновления, когда пользователь перемещается назад к той странице.

Мы заканчиваем наш краткий обзор некоторых из наиболее специальных компоновщиков, доступные в FXRuby. Прежде, чем мы закончим эту главу, мы собираемся заняться более общим вопросом как использовать все эти компоновщики вместе, чтобы решить некоторые общие проблемы.

## 12.6 Стратегия использования различных компоновщиков вместе

По большей части эта глава рассказывает, как изолировать используемые компоновщики. Хотите разметить группу виджетов в строки и столбцы? Используйте **FXMatrix**. Действительно ли виджет является слишком большим, чтобы поместиться на экране? Рассмотрите **FXScrollWindow**. И в то же время Вы должны знать, как все эти компоновщики работают автономно прежде, чем Вы сможете использовать их вообще. Реальная забава начинается когда Вы начинаете использовать их вместе, чтобы решить более сложные проблемы расположения.

Мы затронули эту идею в Разделе 12.1, «*Nesting Layout Managers Inside Each Other*», на странице 166, когда мы смотрели на то, как мы могли вложить **FXHorizontalFrame** в **FXPacker**, чтобы достигнуть расположения которое было бы невозможно достигнуть при использовании любого из этих компоновщиков по отдельности. Действительно, это - один из наиболее важных навыков, которые Вы приобретёте, когда начнёте разрабатывать своё приложение: как сделать сложное форматирование проще, более управляемым и как склеить эти части компоновщиков.

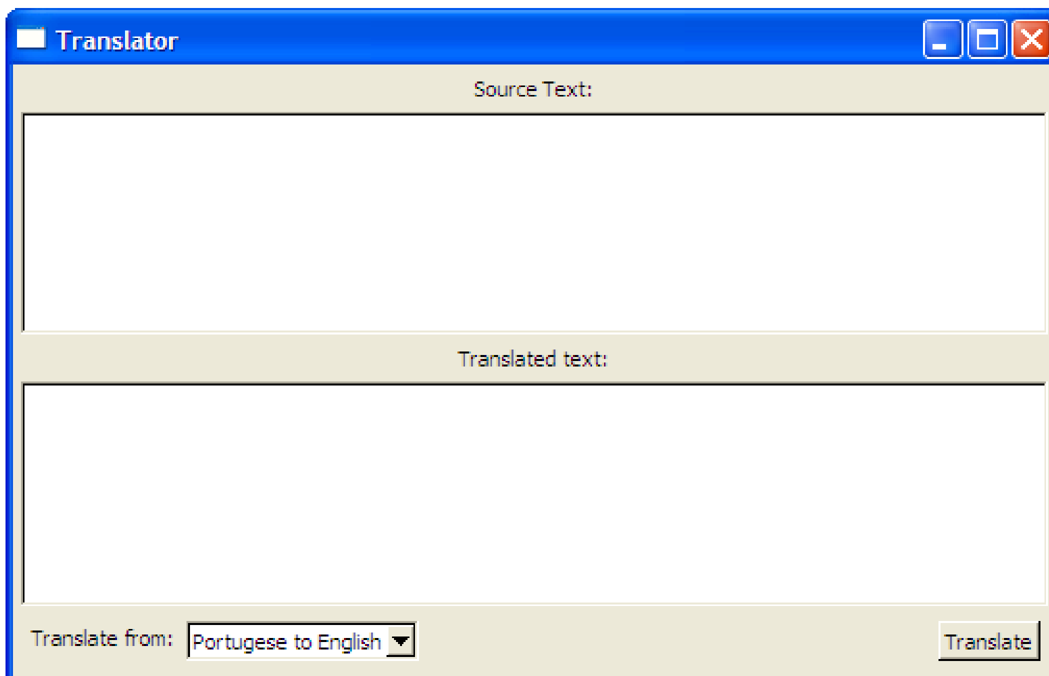



Figure 12.14: Mock-up of a GUI front end to a translation service

Например, если Вы желаете располагать виджеты сверху вниз, Вы, вероятно, будете использовать **FXVerticalFrame**. Позвольте нам рассмотреть относительно простой пример для начала. Рисунок 12.14 показывает макет программы, которая обеспечивает фронтэнд GUI для службы переводчика.

Когда Вы смотрите на главное окно этого приложения, Вы должны видеть пять элементов, сложенных друг в друга. Вверху у нас есть метка (“Исходный текст”), текстовая область, вторая метка (“Преобразованный текст”), вторая текстовая область, и наконец строка средств управления. Когда Вы смотрите на это, Вы думаете о **FXVerticalFrame**. Разумная первая попытка этого расположения могла бы выглядеть как следующее:

```
frame = FXVerticalFrame.new(self)
FXLabel.new(frame, "Source Text:")
source_text = FXText.new(frame)
FXLabel.new(frame, "Translated text:")
translated_text = FXText.new(frame, :opts => TEXT_READONLY)
controls = FXHorizontalFrame.new(frame)
```

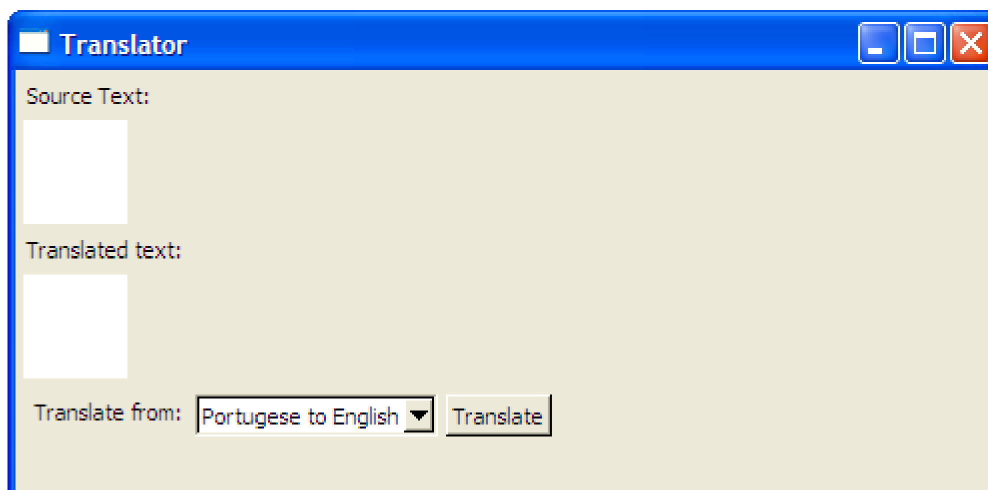



Figure 12.15: A first attempt at the layout

Горизонтальный фрейм средств управления в конце содержит поле комбинированного списка переводчика так же как кнопка *Translate* в нижнем правом углу:

```
FXLabel.new(controls, "Translate from:")
translations = FXComboBox.new(controls, 15,
      :opts => COMBOBOX_STATIC|FRAME_SUNKEN|FRAME_THICK)
translate_button = FXButton.new(controls, "Translate" ,
      :opts => BUTTON_NORMAL)
```

Давайте выполним эту первую редакцию и посмотрим, что получилось.

Рисунок 12.15 показывает то, на что эта версия похожа под Windows. Хорошо уже то, что виджеты сложены как ожидалось.

В большинстве разметок Вы примените по крайней мере один из параметров "заливки" (**LAYOUT_FILL_X** или **LAYOUT_FILL_Y**). В частности Вы будете часто находить, что Вы хотите использовать **LAYOUT_FILL_X** для каждого из дочерних элементов в **FXVerticalFrame**. Позвольте нам сделать это для дочерних виджетов в этом примере. В то же время, также давайте установим **LAYOUT_RIGHT** для кнопки *Translate* так, чтобы она была размещена против правой стороны горизонтального фрейма.

```
frame = FXVerticalFrame.new(self, :opts => LAYOUT_FILL)
FXLabel.new(frame, "Source Text:" , :opts => LAYOUT_FILL_X)
source_text = FXText.new(frame, :opts => LAYOUT_FILL_X)
FXLabel.new(frame, "Translated text:" , :opts => LAYOUT_FILL_X)
translated_text = FXText.new(frame, :opts => TEXT_READONLY|LAYOUT_FILL_X)
controls = FXHorizontalFrame.new(frame, :opts => LAYOUT_FILL_X)
FXLabel.new(controls, "Translate from:" )
translations = FXComboBox.new(controls, 15,
      :opts => COMBOBOX_STATIC|FRAME_SUNKEN|FRAME_THICK)
translate_button = FXButton.new(controls, "Translate" ,
      :opts => BUTTON_NORMAL|LAYOUT_RIGHT)
```

Рисунок 12.16 показывает то, на что эта версия похожа под Windows. Это больше похоже на конечный продукт, но есть все еще несколько проблем. Мы хотели бы что бы эти две текстовые области расли, чтобы взять так много пространства насколько возможно, в то время как другие виджеты (две метки, и строка средств управления вдоль нижней части), располагались так же, как сейчас. Это поведение должно сохраняться даже если мы изменим размеры главного окна. Если Вы экспериментируете с этой последней версией, Вы быстро найдете, что это не имеет место.

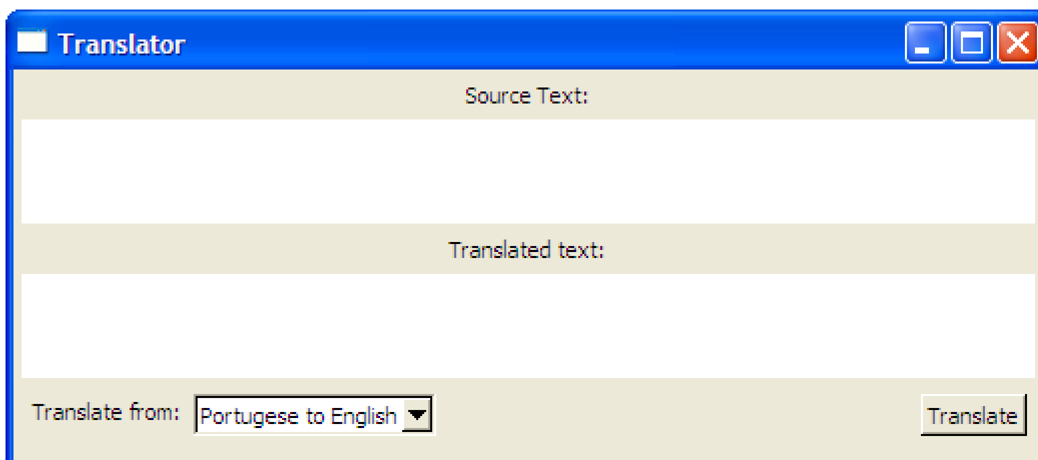



Figure 12.16: After adding layout hints

Давайте делать попытку фиксации, изменяя **LAYOUT_FILL_X** на двух виджетах **FXText** на **LAYOUT_FILL** (который, как Вы вспоминаете, включает оба **LAYOUT_FILL_X** и **LAYOUT_FILL_Y**):

```
frame = FXVerticalFrame.new(self, :opts => LAYOUT_FILL)
FXLabel.new(frame, "Source Text:" , :opts => LAYOUT_FILL_X)
source_text = FXText.new(frame, :opts => LAYOUT_FILL)
FXLabel.new(frame, "Translated text:" , :opts => LAYOUT_FILL_X)
translated_text = FXText.new(frame, :opts => TEXT_READONLY|LAYOUT_FILL)
controls = FXHorizontalFrame.new(frame, :opts => LAYOUT_FILL_X)
FXLabel.new(controls, "Translate from:" )
translations = FXComboBox.new(controls, 15,
      :opts => COMBOBOX_STATIC|FRAME_SUNKEN|FRAME_THICK)
translate_button = FXButton.new(controls, "Translate" ,
      :opts => BUTTON_NORMAL|LAYOUT_RIGHT)
```

Для одного последнего штриха позвольте нам для каждого из виджетов **FXText** создать толстую границу вокруг. В этом случае, не имеет значения, используем ли мы горизонтальный фрейм или вертикаль фрейм, так как у каждого фрейма есть только один дочерний виджет:

```
source_text_frame = FXHorizontalFrame.new(frame,
      :opts => FRAME_SUNKEN|FRAME_THICK|LAYOUT_FILL,
      :padding => 0)
source_text = FXText.new(source_text_frame,
      :opts => LAYOUT_FILL)
FXLabel.new(frame, "Translated text:" , :opts => LAYOUT_FILL_X)
translated_text_frame = FXHorizontalFrame.new(frame,
      :opts => FRAME_SUNKEN|FRAME_THICK|LAYOUT_FILL,
      :padding => 0)
translated_text = FXText.new(translated_text_frame,
      :opts => TEXT_READONLY|LAYOUT_FILL)
```

Если Вы выполняете пример в этой точке, расположение должно наконец напомнить наше исходное как показано в рисунке 12.14, на странице 182.

В этом последнем шаге, который мы сделали, вложение текстовых областей внутрь дополнительных фреймов имеет единственную цель обеспечить некоторое художественное оформление вокруг них. Это может казаться Вам расточительным. Ну, смотрите сами.

Поскольку Вы считаете различные варианты разметок вложения, Вы никогда не должны проводить слишком много времени, волнуясь о число виджетов, которые Вы создаете. Если решение, которое имеет больше смысла и является самым легким для Вас включает использование большого количества горизонтальных и вертикальных вложенных фреймов друг в друге смело идите на это. Одно из преимуществ использования FOX является то, что это очень благоприятно для ресурсов, и "очень дешево" создать и уничтожить виджеты на лету, когда Вы должны сделать это.

В начале этой главы я отметил, что большинство пользователей не пытаются узнать о роли, которую компоновщики играют в работе приложения. Это также справедливо для выпадающих меню и панелей инструментов, которые являются настолько банальными в приложениях GUI, что часто недооценивается роль, которую они играют. В следующей главе мы собираемся более внимательно рассмотреть как Вы можете лучше всего интегрировать меню и панели инструментов в Ваши Приложения FXRuby.

Глава 13

# Advanced Menu Management

Легко недооценить полноценность хорошо разработанных меню, но они - важная функция любого приложения GUI. Первая остановка для новые пользователи Вашего приложения почти всегда будут строкой меню. Для пользователей меню обеспечивает своего рода изучение и инструмент открытия, дает им краткий обзор функций приложения. Другие пользователи могут работали с приложением в течение некоторого времени, но они используют его настолько нечасто, что они не могут сразу вспомнить, как получить доступ к некоторым функциям приложения. Для этих пользователей, меню обеспечит "курсы повышения квалификации" в том, как добиться цели.

В Главе 5, «*Display an Entire Album*», на странице 43, когда мы расширяли приложение Иллюстрированной книги, чтобы учесть создание нового фотоальбомы и импорта фотографий в эти альбомы, мы получили введение в процесс добавления строк меню с выпадающими меню к приложению FXRuby. В этой главе мы будем смотреть на усовершенствованные опции для того, чтобы представить функциональность приложения. Мы рассмотрим некоторые альтернативы стандартной области меню, которые позволяют обеспечить расположение каскадом и прокрутку меню. После этого мы будем рассмотрим некоторые другие виды пунктов меню. Мы закончим главу, рассмотрев тесно связанный предмет, как добавить панель инструментов (или панели инструментов) к приложению.

## 13.1 Creating Cascading and Scrolling Menus

После работы через пример Иллюстрированной книги Вы знаете основные шаги добавления выпадающего меню к приложению:

1. Создайте виджет **FXMenuBar** как дочерний элемент главного окна.
2. Создайте один или более виджетов **FXMenuPane**, принадлежавших основному окну, для каждого меню, которое Вы хотите вывести на экран. Так, например, у Вас могла бы быть одна область меню для меню *File* и другая область меню для меню *Edit*.
3. Создайте виджет **FXMenuTitle** как дочерний элемент строки меню и установите ссылку между строкой меню и областью меню.
4. Добавьте один или более виджетов **FXMenuCommand** к области меню, каждое из которых представляет действие, которое может выполнить пользователь. Используйте метод *connect()*, чтобы связать блок команд Ruby с каждой из команд меню.

На стандартном **FXMenuPane** есть несколько полезных виджетов, которые мы использовали. *Cascading menus* - области меню вложены в других областях меню. Например, уже предположите Вас имейтся меню *File*, с командами *New*, *Open...*, *Save* и *Save As...*:

```
file_menu_pane = FXMenuPane.new(self)
file_new_command = FXMenuCommand.new(file_menu_pane, "New")
file_open_command = FXMenuCommand.new(file_menu_pane, "Open...")
file_save_command = FXMenuCommand.new(file_menu_pane, "Save")
file_save_as_command = FXMenuCommand.new(file_menu_pane, "Save As...")
file_menu_title = FXMenuTitle.new(menu_bar, "File" ,
      :popupMenu => file_menu_pane)
```

Теперь Вы хотели бы добавить подменю Export с командами Export as GIF, Export as PNG, и так далее. Создайте новую область меню, как Вы делали для меню File:

```
export_menu_pane = FXMenuPane.new(self)
export_gif_command =
      FXMenuCommand.new(export_menu_pane, "Export as GIF")
export_jpeg_command =
      FXMenuCommand.new(export_menu_pane, "Export as JPEG")
export_png_command =
      FXMenuCommand.new(export_menu_pane, "Export as PNG")
export_tiff_command =
      FXMenuCommand.new(export_menu_pane, "Export as TIFF")
```

Теперь, добавьте виджет **FXMenuCascade** к области меню File, рядом с существующими виджетами **FXMenuCommand**:

```
export_cascade = FXMenuCascade.new(file_menu_pane, "Export" ,
      :popupMenu => export_menu_pane)
```

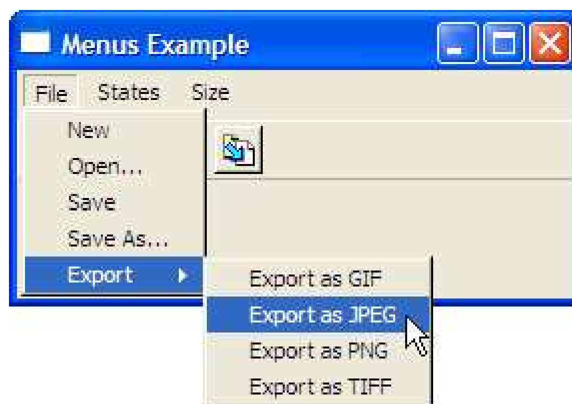



Figure 13.1: A cascading menu pane

Рисунок 13.1 показывает то, на что похожа эта область подменю. Как Вы можете видеть, **FXMenuCascade** подобен виджету **FXMenuTitle**, который мы используем, чтобы присоединить меню непосредственно к панели меню. Вы можете даже вложить другое подменю в *export_menu_pane* если Вы хотели бы, но не увлекайтесь этим. Глубоко вложенные меню действительно трудно использовать, и Вы должны использовать их экономно.

Другой прием, который Вы можете счесть полезным, когда у Вас есть много информации, чтобы попытаться упаковать в меню - это область прокрутки. Это особенно полезно, когда пункты меню сгенерированы программно и Вы не знайте заранее, сколько элементов область меню будет содержать.

Так как **FXScrollPane** это подкласс от **FXMenuPane**, он может использоваться везде где это область меню может использоваться. Первый параметр **FXScrollPane.new** - это окно владельца для области меню, и второй параметр - число видимых элементов, которые должны быть выведены на экран:

```
states_menu_pane = FXScrollPane.new(self, 8)
$state_names.each do |state_name|
      FXMenuCommand.new(states_menu_pane, state_name)
end
states_menu_title = FXMenuTitle.new(menu_bar, "States" ,
      :popupMenu => states_menu_pane)
```

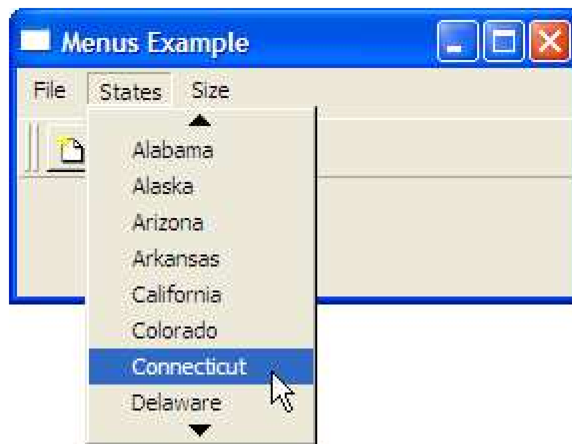



Figure 13.2: A scrolling menu pane

Рисунок 13.2 показывает то, на что похожа эта область меню прокрутки. Отметьте стрелки с обоих концов области меню. Если Вы поместите курсор мыши на любую из этих стрелок, область меню прокрутит свое содержание или вниз. Как и подменю, прокручивание области меню должно использоваться экономно, потому что они являются немного трудными для пользователей.

Расположение каскадом и прокрутка меню являются только парой примеров вариаций в стандартных меню, которые мы использовали в Иллюстрированной книге. В следующем разделе мы будем смотреть на некоторые альтернативы к стандартной кнопке **FXMenuCommand**, которую мы узнали.

## 13.2 Adding Separators, Radio Buttons, and Check Buttons to Menus

Вы уже знаете, как использовать виджет **FXMenuCommand**, чтобы обеспечить пользовательский интерфейс к командам, таким как Open или Save. Вы может добавить виджет **FXMenuSeparator** к области меню, чтобы создать визуальное разделение между группами связанных команд:

```
FXMenuSeparator.new(menu_pane)
```

Виджеты **FXMenuRadio** и **FXMenuCheck** дают Вам путь к той же самой функциональности что обеспечивают виджеты **FXRadioButton** и **FXCheck**. Как Вы научились в Разделе 8.1, «*Making Choices with Radio Button*», на странице 106, лучший способ гарантировать, что варианты для радио-элементов остаются взаимоисключающими, это связать каждого из них с тем же самым **FXDataTarget**:

```
@size = FXDataTarget.new(1)
size_pane = FXMenuPane.new(self)
FXMenuRadio.new(size_pane, "Small" ,
      :target => @size, :selector => FXDataTarget::ID_OPTION)
FXMenuRadio.new(size_pane, "Medium" ,
      :target => @size, :selector => FXDataTarget::ID_OPTION+1)
FXMenuRadio.new(size_pane, "Large" ,
      :target => @size, :selector => FXDataTarget::ID_OPTION+2)
FXMenuRadio.new(size_pane, "Jumbo" ,
      :target => @size, :selector => FXDataTarget::ID_OPTION+3)
size_menu_title = FXMenuTitle.new(menu_bar, "Size" ,
      :popupMenu => size_pane)
@size.connect(SEL_COMMAND) do
      # @size.value holds the index of the selected size
end
```

Вот подобный пример для элемента **FXMenuCheck**:

```
@fit_to_screen = FXDataTarget.new(false)
FXMenuCheck.new(size_pane, "Fit Contents to Screen" ,
      :target => @fit_to_screen, :selector => FXDataTarget::ID_VALUE)
```

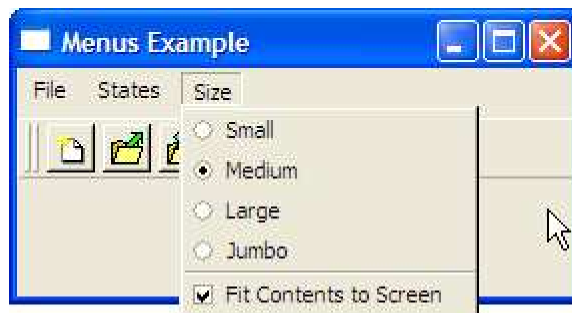



Figure 13.3: A menu with check and radio button items

Рисунок 13.3 показывает меню, содержащее группу кнопок **FXMenuRadio**, отделенных от единственной кнопки **FXMenuCheck** элементом **FXMenuSeparator**.

Выпадающие меню как виды, которые мы создавали до сих пор, важны для всего кроме самого простого приложения GUI. Они полезны для новичков или нечастых пользователей приложения, потому что они предоставляют своего рода обучающий инструмент людям, которые учатся как использовать приложение. Поскольку пользователи становятся более знакомыми с приложениями, они желают получить более быстрый доступ к использованию функций, и один из способов обеспечить этот немедленный доступ это панели инструментов. В следующих нескольких разделах мы будем учиться как добавить панели инструментов в приложение FXRuby.

## 13.3 Adding Toolbars to an Application

Это - короткий раздел, потому что нечего много говорить о виджете **FXToolBar**. По большей части Вы можете думать об нём как о другом компоновщике (и это фактически получено из класса **FXPacker**).

Обычное использование для **FXToolBar** это заполнение его иконками - украшение кнопок, которые обеспечивают легкий доступ к обычно используемым функциям. Они – это собственно виджет **FXButon**, но с надлежащей комбинацией атрибутов расположения иконки и размеров, чтобы гарантировать, что все имеют одинаковый размер. Во первых надо удостовериться, что **PACK_UNIFORM_WIDTH** установлен для панели инструментов:

```
tool_bar = FXToolBar.new(top_dock_site, tool_bar_shell,
      :opts => PACK_UNIFORM_WIDTH|FRAME_RAISED|LAYOUT_FILL_X)
```

Вы можете также включать опцию **PACK_UNIFORM_HEIGHT**, чтобы видеть эффект, который имеет появление панели инструментов. Затем, Вы можете добавить один или более виджетов **FXButton** для различных команд панели инструментов:

```
new_button = FXButton.new(tool_bar,
      "\tNew\tCreate new document." ,
      :icon => new_icon)
open_button = FXButton.new(tool_bar,
      "\tOpen\tOpen document file." ,
      :icon => open_icon)
save_button = FXButton.new(tool_bar,
      "\tSave\tSave document." ,
      :icon => save_icon)
save_as_button = FXButton.new(tool_bar,
      "\tSave As\tSave document to another file." ,
      :icon => save_as_icon)
```

Figure 13.4: Use a toolbar for immediate access to commonly used commands.

Рисунок 13.4 показывает то, на что похожа эта панель инструментов, когда программа выполняется под Windows.

Отметьте формат строки метки для каждой из кнопок. В секции 8.3, «*Providing Hints with Tooltips and the Status Bar*», на странице 113, мы говорили о том, как Вы можете встроить подсказки для кнопки, чтобы разделить метку кнопки от ее подсказки и сообщения строки состояния. Для этих кнопок на панели инструментов, строка метки начинается с символа табуляции, что означает, что они не будут выводить на экран метку кнопки. Например, первая кнопка будет выводить на экран подсказку «*New*», и сообщение в строке состояния “*Create new document*” но поверхность кнопки покажет только свой значок.

Действительно значительная разница между **FXToolBar** и компоновщиками - это то, что может быть сконфигурировано так, чтобы пользователь мог перетащить это из главного окна так, чтобы это "плавало" в другом месте экрана. Давайте посмотрим как это работает.

## 13.4 Creating Floating Menu Bars and Toolbars

До сих пор мы имели дело только со стационарной строкой меню или панелью инструментов. Она может быть более узкой или широкой, поскольку Вы изменяете размеры главного окна, но она остается на том же самом месте. **FXRuby** также предусматривает плавающие строки меню и панели инструментов которые могут перемещаться в другое место на экране. В зависимости от типа приложения, которое Вы разрабатываете, это может быть чрезвычайно полезной функцией.

Классы **FXMenuBar** и **FXToolBar** производные от **FXDockBar**. *Dock bar* может быть прикреплена в *dock site*, она может быть расположена где-нибудь на главном окне, или за пределами главного окна в некотором другом родительском контейнере (обычно, окно **FXToolBarShell**).

Давайте посмотрим краткий пример, чтобы увидеть, как это работает. Во-первых, мы нуждаемся в создании виджета **FXToolBarShell**, который будет действовать как дом панели инструментов далеко от дома, когда это плавает вокруг:

```
tool_bar_shell = FXToolBarShell.new(self)
```

Затем, создадим один или более сайтов прикрепления, которые обеспечивают места для toolbar, чтобы приземлиться, когда это готово возвратиться домой. Обычно совершенно приемлемо определять только один сайт прикрепления, но в целях этого примера мы установим два сайта прикрепления: один вдоль главной стороны главного окна, другой вдоль нижней части:

```
top_dock_site = FXDockSite.new(self,
      :opts => LAYOUT_FILL_X|LAYOUT_SIDE_TOP)
bottom_dock_site = FXDockSite.new(self,
      :opts => LAYOUT_FILL_X|LAYOUT_SIDE_BOTTOM)
```

Пользователь может перетащить плавающую панель инструментов на любую из этих позиций на главном окне. Важно, что когда панель инструментов плавает, сайты прикрепления скрыты и позволяют другим соседним виджетам занимать их место. Теперь мы можем создать **FXToolBar** непосредственно:

```
tool_bar = FXToolBar.new(top_dock_site, tool_bar_shell,
      :opts => PACK_UNIFORM_WIDTH|FRAME_RAISED|LAYOUT_FILL_X)
```

Первый параметр **FXToolBar.new** - начальный сайт прикрепления, и второй параметр - shell. Теперь мы должны добавить "grip", который пользователь может захватить, чтобы оторвать панель инструментов с сайта прикрепления:

```
FXToolBarGrip.new(tool_bar,
      :target => tool_bar, :selector => FXToolBar::ID_TOOLBARGRIP,
      :opts => TOOLBARGRIP_DOUBLE)
```

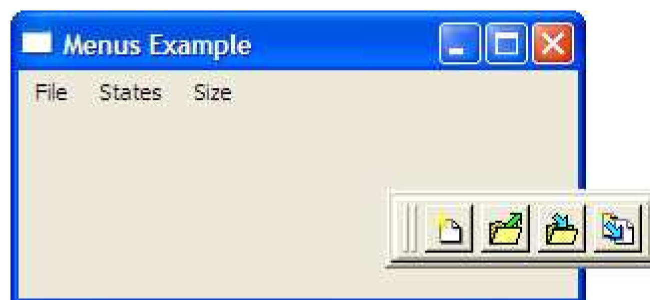



Figure 13.5: Floating toolbar

Я предпочитаю grip ввиде двойной линии, но если Вы хотели бы одну линию, передайте **TOOLBARGRIP_SINGLE** как опцию вместо этого. В этой точке, Вы можете добавить все специализированные виджеты к панели инструментов. Не обязательно делать виджет **FXToolBarGrip** первым дочерним виджетом в **FXToolBar**, но, вероятно, лучше сделать его или первым или последним виджетом, чтобы избежать загромождения кнопок.

Вы уже видели то, на что панель инструментов похожа в прикрепленной позиции, на рисунке 13.4, на странице 193. Рисунок 13.5 показывает, как выглядит панель инструментов после того, как это было "расстыковано" и перетащено прочь.

Вплоть до этой точки мы рассматривали, как обработать взаимодействие с главным окном приложения, но также иногда полезно изолировать определенные взаимодействия в отдельном высокоуровневом окне, известном как диалоговое окно. **FXRuby** обеспечивает много встроенных стандартных диалоговых окон, такие как **FXFileDialog**, который мы использовали, когда мы создавали Иллюстрированную книгу. Вы можете также создать пользовательские диалоговые окна для того, чтобы обработать настройки приложения или другую функциональность. Мы рассмотрим все эти темы в следующей главе.

Глава 14

# Providing Support with Dialog Boxes

До сих пор мы говорили о том, как использовать **FXRuby**, чтобы создать основное окно пользовательского интерфейса Вашего приложения. Всё сосредоточилось на основном взаимодействии пользователя с приложением. **FXRuby**, также обеспечивает различные высокоуровневые окна, известные как диалоговые окна, и это тема этой последней главы.

Диалоговое окно подобно главному окну, в котором оно "плавает" это высокоуровневый контейнер и служит для набора других виджетов. Как главное окно, диалоговое окно может включать строку заголовка, строку меню, строку состояния, и другие виды художественных оформлений для изменения размеров или закрытия окна. Несмотря на эти общие черты, Вы не должны перепутать диалоговые окна с основной программой. В большинстве случаев они - переходный процесс и остаются на экране только в течение короткого времени, в то время как пользователь взаимодействует с ними, и они всегда играют вторичные, поддерживающие роли в приложении.

Вы уже встретились с одним видом диалогового окна в Разделе 5.3, *«Импорт Фотографии из Файлов»*, на странице 50, когда мы использовали **FXFileDialog**, чтобы запросить от пользователя имена файлов фотографий для импорта. **FXRuby** включает много других стандартных диалоговых окон для использования в Ваших приложениях. В этой главе мы проведём краткий тур по большинству стандартных диалоговых окон обеспеченных в **FXRuby**. Мы будем также говорить о создании пользовательских диалоговых окон для тех ситуаций, где ни одно из стандартных диалоговых окон не отвечает всем требованиям. Давайте повторно посетим нашего старого друга, диалоговое окно **FXFileDialog**.

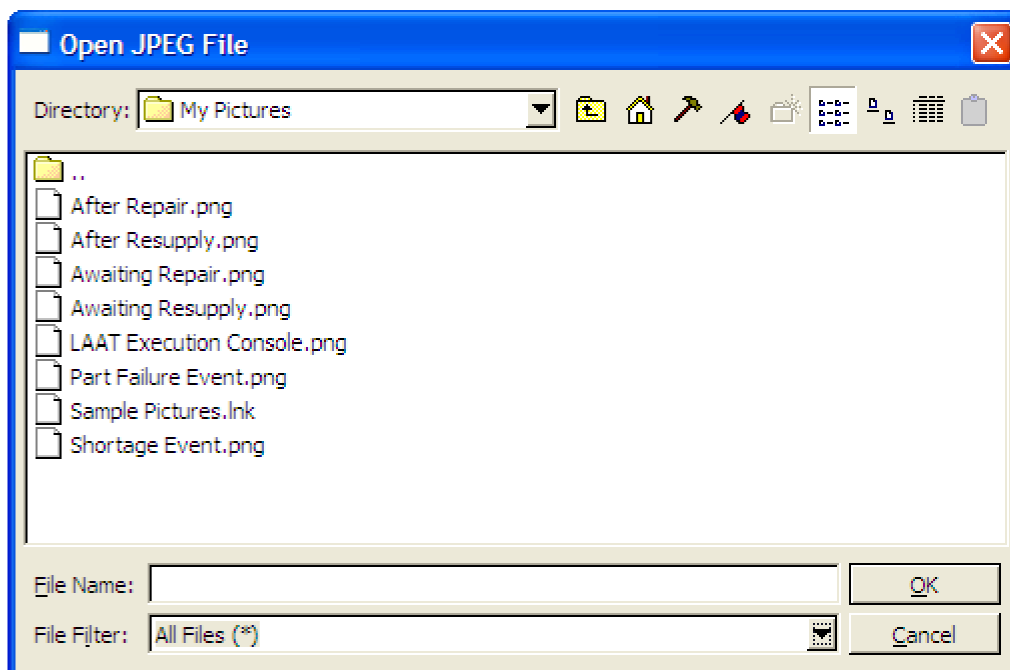



Figure 14.1: Use FXFileDialog to select files.

## 14.1 Selecting Files with the File Dialog Box

**FXFileDialog**, показанный в рисунке 14.1, вероятно, наиболее часто используемое стандартное диалоговое окно. Вы можете использовать **FXFileDialog** когда Вы хотите, чтобы пользователь выбрал существующий файл или файлы (например, во время Операция открытия), или когда Вы хотите попросить у пользователя имя файла, который Вы собираетесь записать (например, во время работы Сохранения). Например, предположите, что Вы хотите, чтобы пользователь выбрал существующий файл JPEG на диске:

```
dialog = FXFileDialog.new(self, "Open JPEG File")
dialog.patternList = [
      "All Files (*)",
      "JPEG Files (*.jpg, *.jpeg)"
]
dialog.selectMode = SELECTFILE_EXISTING
if dialog.execute != 0
      open_jpeg_file(dialog.filename)
end
```

Когда Вы вызываете **execute()** на **FXFileDialog**, он выведет на экран себя, позволит пользователю выбрать файл, и затем ожидает от пользователя, чтобы он щелкнул или по OK или кнопке отмены. Если *execute()* возвратил ноль, пользователь щелкнул по Отмене; другой пользователь нажал OK. В этой точке Вы можете проверить значение атрибута имени файла диалогового окна, чтобы считать полный путь выбранного файла.

Диалоговое окно выбора файла имеет множество функций которые Вы ожидали бы от этого вид виджета и некоторые Вы, возможно, не видели прежде (такие как установка закладки для часто посещаемых каталогов). Как показано в предыдущем примере, Вы можете инициализировать **patternList** массив строк - это укажет на доступные фильтры файла. Режим выбора файла указывает выбраете ли вы только единственный файл или множество файлов. Выбор режима **SELECTFILE_EXISTING** означает, что пользователь может выбрать только существующий файл; это было бы соответствующей установкой для того, чтобы загрузить документ. Мы могли бы вместо этого учесть выбор множества файлов.

```
dialog = FXFileDialog.new(self, "Open JPEG File(s)")
dialog.patternList = [
      "All Files (*)",
      "JPEG Files (*.jpg, *.jpeg)"
]
dialog.selectMode = SELECTFILE_MULTIPLE
if dialog.execute != 0
      dialog.filenames.each do |filename|
            open_jpeg_file(filename)
      end
end
```

**FXFileDialog** предоставляет много возможностей, которые позволяют Вам настраивать его появление и поведение; см. документацию API для **FXFileDialog** и классы **FXFileSelector** для большего количества деталей.

Стоит отметить здесь, что **FXFileDialog** определенно приспособлен к работе с отдельными файлами (или группой их, в зависимости от обстоятельств). Хотя Вы можете использовать режим выбора **SELECTFILE_DIRECTORY**, чтобы ограничить выбор в диалоговом окне файла только каталогами. Для тех ситуаций, где Вы желаете, чтобы пользователь выбрал каталог, и только каталог, **FXDirDialog** может быть лучшим выбором. Мы рассмотрим это диалоговое окно.

## 14.2 Selecting a Directory with the Directory Dialog Box

Вы можете использовать **FXDirDialog**, когда пользователь должен выбрать один каталог, и только каталог. Для чего-либо сложного более, Вы вероятно, примените **FXFileDialog**.

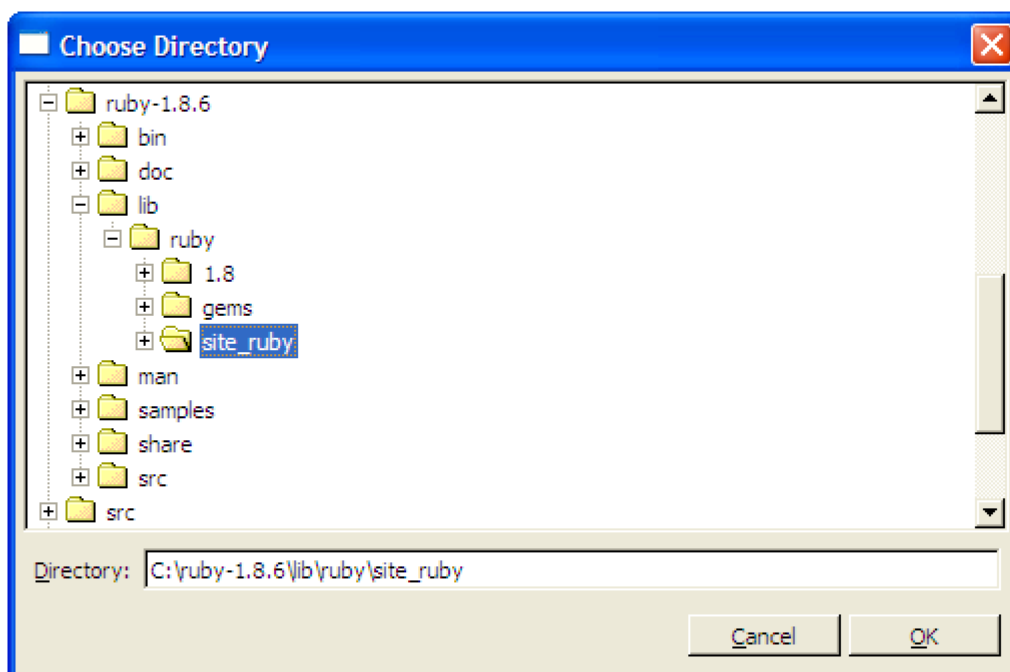



Figure 14.2: Use a directory dialog box to select a single directory.

Поскольку у **FXDirDialog** есть одна очень простая функция, его просто сконфигурировать и использовать. В большинстве случаев Вы просто инициализируете атрибут **directory** к некоторому пути в файловой системе, выведите на экран диалоговое окно, и затем получите выбранный каталог от атрибута **directory**:

```
dialog = FXDirDialog.new(self, "Choose Directory")
dialog.directory = "/Users/lyle"
if dialog.execute != 0
      open_directory(dialog.directory)
end
```

Отметьте что, если Вы не инициализируете атрибут **directory** перед отображением диалогового окна, он принимает значение по умолчанию текущего рабочего каталога.

Рисунок 14.2 показывает, что диалоговое окно каталога выводит на экран файловую систему как древовидную структуру. Чтобы выбрать существующий каталог, просто переместитесь к нему в списке дерева каталогов, и щелкните кнопку OK. Чтобы создать новый каталог, щелкните правой кнопкой по родительскому каталогу для нового каталога, и выберите команду New из раскрывающегося меню.

Диалоговые окна не просто полезны для собирания информации о файлах и каталогах. В следующих немногих разделах мы рассмотрим стандарт на диалоговые окна для того, чтобы иметь дело с цветами и шрифтами.

## 14.3 Choosing Colors with the Color Dialog Box

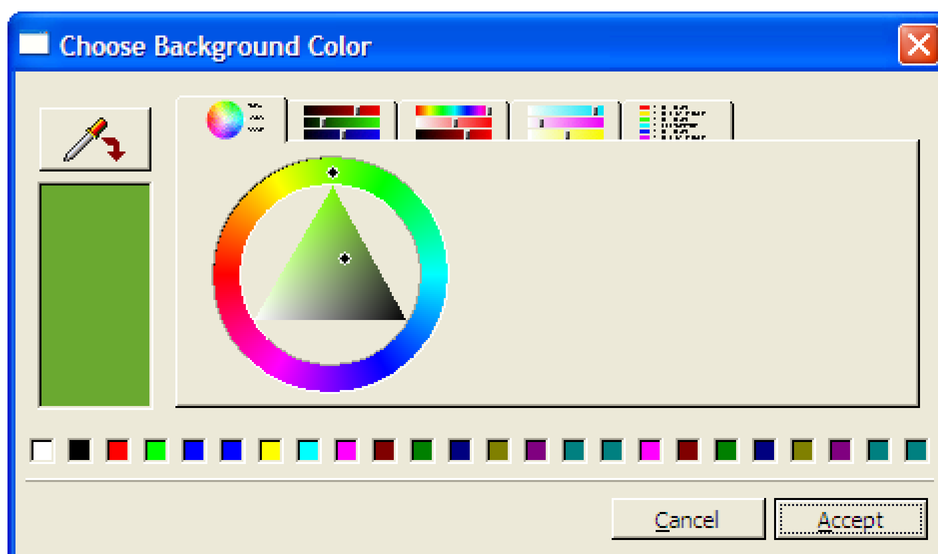



Figure 14.3: Color dialog box in HSV dial mode

Вы можете использовать **FXColorDialog**, когда Вы нуждаетесь чтобы пользователь выбрал (или изменил) значение цвета.

```
dialog = FXColorDialog.new(self, "Choose Background Color")
dialog.rgba = FXRGB(255, 0, 0)    # initialize color to red
if dialog.execute != 0
     self.backColor = dialog.rgba
end
```

Если Вы не хотите использовать начальный (черный) цвет по умолчанию, Вы можете инициализировать значение цвета прежде, чем вывести на экран диалоговое окно установкой значения атрибута **rgba**.

Хотя цветное диалоговое окно удобно в общем, его большое разнообразие опций может сделать его неудобным для конечных пользователей. Цветное диалоговое окно содержит пять вкладок, каждое из которых выводит на экран в настоящий момент выбранный цвет, используя различную цветовую модель.

- Первая вкладка выводит на экран набор для того, чтобы скорректировать Оттенок, Насыщенность, и Значение (HSV) компоненты цвета. Вы можете видеть что эта страница из **FXColorDialog** похожа на рисунке 14.3, на предыдущей странице.
- Вторая вкладка выводит на экран ряд ползунков для того, чтобы установить Красный, Зеленый, Синий, и компоненты Альфы цвета.
- Третья вкладка выводит на экран ряд ползунков, снова для того, чтобы установить Компоненты HSV цвета.
- Четвертая вкладка выводит на экран ряд ползунков для того, чтобы установить Синий, Пурпурный, Желтый и Ключ (CMYK) компоненты цвета.
- Последняя вкладка выводит на экран список имен цветов.

Кроме того, цветное диалоговое окно включает на его левой стороне кнопку “color picker”, которая позволяет Вам выбрать цвет отовсюду на экране как новый цвет, и вдоль нижней части, набор предопределенных цветов.

## 14.4 Selecting Fonts with the Font Dialog Box

Мы ссылались на **FXFontDialog** в Разделе 11.1, «*Использование Пользовательских Шрифтов*», на странице 143, когда мы обсуждали какую информацию возвращает диалоговое окно выбора шрифта, используя объект **FXFontDesc**. Как и другие диалоговые окна, которые мы рассмотрели на в этой главе, образец использования для **FXFontDialog** должен инициализировать свои настройки, вывести на экран диалоговое окно, чтобы позволить пользователю взаимодействовать с ним, и затем получить информацию о выборе пользователя. Рисунок 14.4, на следующей странице, показывает диалоговое окно выбора шрифта.

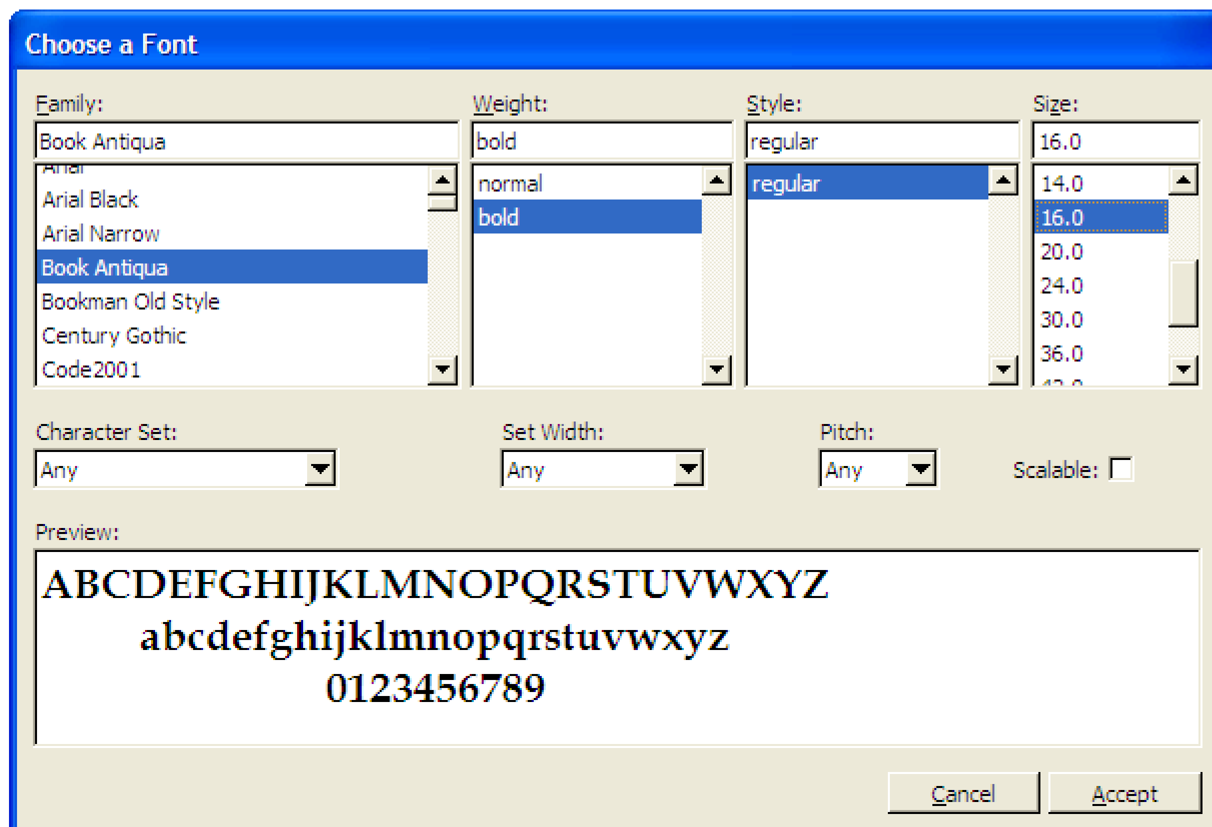



Figure 14.4: Selecting fonts in a font dialog box

Работа с диалоговым окном выбора шрифта немного более хитра чем работа с файлом, каталогом или выбором цвета, так как мы передаем назад и вперед объект **FXFontDesc** а не фактические объекты **FXFont**. Следующая выборка кода демонстрирует типичный образец взаимодействия с **FXFontDialog**:

```
dialog = FXFontDialog.new(self, "Choose a Font")
dialog.fontSelection = button.font.fontDesc
if dialog.execute != 0
      new_font = FXFont.new(app, dialog.fontSelection)
      new_font.create
      button.font = new_font
end
```

Первый шаг должен извлечь начальные настройки шрифта (как **FXFontDesc** объект) от существующего шрифта через его атрибут **fontDesc**. Мы можем использовать это описание шрифта, чтобы инициализировать атрибут **fontSelection** **FXFontDialog**. Этот шаг не строго необходим; диалоговое окно шрифта принимает некоторые настройки по умолчанию, если Вы не инициализируете атрибут **fontSelection**.

Как только пользователь закончил взаимодействие с диалоговым окном выбора шрифта и сделал выбор, мы должны получить описание шрифта от диалогового атрибута **fontSelection** и использовать его, чтобы создать новый объект **FXFont.** Поскольку мы обсуждали в Главе 11, «*Создание Визуально Богатого Пользовательского Интерфейса*», на странице 142, что крайне важно, чтобы мы вызвали, create() на недавно созданном объекте **FXFont** прежде, чем мы присвоим его виджету.

## 14.5 Alerting the User with Message Boxes

По сравнению с другими стандартными диалоговыми окнами окно сообщения самое простое. Оно предусматривает простое взаимодействие с пользователем. Например, следующий **FXMessageBox** выводит на экран предупреждающее сообщение показанное на рисунке 14.5:

Figure 14.5: Displaying a warning using a message box

```
FXMessageBox.warning(
      self,
      MBOX_OK,
      "Buyer Beware",
      "All Sales are Final!"
)
```

В отличие от других диалоговых окон, что мы рассмотрели, **FXMessageBox** обычно создается и выводится на экран за один раз, используя метод класса **warning()**. Так как мы сконфигурировали это окно сообщения, используя опцию **MBOX_OK**, оно выводит на экран только кнопку OK, и нет никакой потребности проверять возвращаемое значение метода **warning()**. Если окно сообщения включает больше чем одну кнопку завершения, Вы должны получить возвращаемое значение метода, чтобы определить, какую кнопку пользователь нажал.

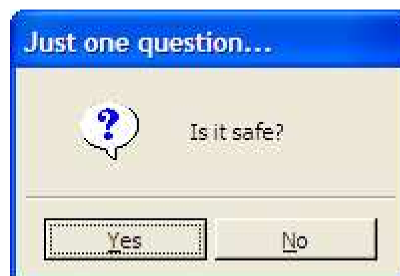



Figure 14.6: Asking a question using a message box

Например, окно сообщения, показанное в рисунке 14.6, задает вопрос. Можно ответить YES или NO:

```
answer = FXMessageBox.question(
      self,
      MBOX_YES_NO,
      "Just one question...",
      "Is it safe?"
)
if answer == MBOX_CLICKED_YES
      ask_again()
end
```

Класс **FXMessageBox** также предоставляет методы *information()* и *error()* для того, чтобы вывести на экран эти виды сообщений. Для полного списка опции окна сообщения и возможных возвращаемых значений, смотрите API для класса **FXMessageBox**.

## 14.6 Creating Custom Dialog Boxes

Если у Вашего приложения есть требование, которое может быть удовлетворено при использовании одного из стандартных диалоговых окон предпочтительно придерживаться стандарта так, чтобы пользователи Вашего приложения получили непротиворечивый и знакомый пользовательский интерфейс. Для многих приложений, однако, Вы должны будете разработать одно или более пользовательских диалоговых окон, чтобы обработать специализированную функциональность. В этом разделе мы создадим типичное Привилегированное диалоговое окно, которое Вы могли бы включать в приложение.

Как отмечено во введении к этой главе, диалоговые окна походят на главные окна разными способами. Первое шаг в создании пользовательского диалогового окна это подкласс **FXDialogBox**:

```
class PreferencesDialog < FXDialogBox
      def initialize(owner)
            super(owner, "Preferences", DECOR_TITLE|DECOR_BORDER|DECOR_RESIZE)
end
```

Теперь мы должны добавить строку завершающих кнопок вдоль нижней части диалогового окна. Мы говорим о кнопках, что Вы используете, чтобы отклонить диалоговое окно, как только Вы всё сделали. Довольно распространено видеть OK и кнопки Cancel и возможно также кнопку Apply, если это имеет смысл для диалогового окна. Мы собираемся добавить горизонтальный фрейм вдоль нижней стороны окна, и затем добавляем сначала кнопку OK, сопровождаемую кнопкой Cancel. Вот код для метода **add_terminating_buttons()**:

```
def add_terminating_buttons
      buttons = FXHorizontalFrame.new(self,
            :opts => LAYOUT_FILL_X|LAYOUT_SIDE_BOTTOM|PACK_UNIFORM_WIDTH)
      FXButton.new(buttons, "OK",
            :target => self, :selector => FXDialogBox::ID_ACCEPT,
            :opts => BUTTON_NORMAL|LAYOUT_RIGHT)
      FXButton.new(buttons, "Cancel",
            :target => self, :selector => FXDialogBox::ID_CANCEL,
            :opts => BUTTON_NORMAL|LAYOUT_RIGHT)
end
```

Поскольку мы добавляем кнопку *OK* и передаём ей параметр **LAYOUT_RIGHT**, эта кнопка будет расположена против правой стороны горизонтального фрейма. Когда мы впоследствии добавим кнопку *Cancel*, она также расположится против правой стороны

остающегося пространства в горизонтали фрейма, что означает, что она появится налево от кнопки *OK*. Это - довольно стандартное расположение для этих двух кнопок, но если Вы предпочли бы, чтобы кнопка *OK* была слева и кнопка *Cancel* справа, Вы можете изменить порядок этих двух операторов.
Мы также используем в своих интересах факт что класс **FXDialogBox**, который является подклассом **PreferencesDialog**, определяет два идентификатора сообщений **ID_ACCEPT** и **ID_CANCEL**, которые мы можем отправить непосредственно от кнопок *OK* и *Cancel* к диалоговому окну, чтобы отклонить его. Если пользователь щелкает по кнопке OK, это отправит сообщение типа **SEL_COMMAND** с идентификатором **ID_ACCEPT**, назад к объекту диалогового окна. Когда диалоговое окно получает это сообщение, оно закроется и гарантирует что вызов *execute()* возвратит ненулевое значение. Если диалоговое окно получает сообщение **ID_CANCEL** вместо этого, оно гарантирует что *execute()* возвратит ноль.

Теперь, когда у нас есть завершающие кнопки, мы можем вернуться к основной деятельности. Мы собираемся вывести на экран привилегированные настройки в **FXTabBook**, используя тот же самый пример, который мы представляли в Разделе 12.5, *«Organizing Windows with Tabbed Notebooks»*, на странице 179. Давайте напишем метод *add_tab_book()*, который создает **FXTabBook** и добавляет несколько вкладок и пустых страниц. Мы озаботимся о содержимом страниц через мгновение.

```
tabbook = FXTabBook.new(self, :opts => LAYOUT_FILL)
basics_tab = FXTabItem.new(tabbook, " Basics ")
basics_page = FXVerticalFrame.new(tabbook,
      :opts => FRAME_RAISED|LAYOUT_FILL)
contact_tab = FXTabItem.new(tabbook, " Contact ")
contact_page = FXVerticalFrame.new(tabbook,
      :opts => FRAME_RAISED|LAYOUT_FILL)
extras_tab = FXTabItem.new(tabbook, " Extras ")
extras_page = FXVerticalFrame.new(tabbook,
      :opts => FRAME_RAISED|LAYOUT_FILL)
```

Ранее в этой главе, когда мы говорили о том, как использовать стандартные диалоговые окна, мы установили образец инициализации диалогового окна с некоторыми данными по умолчанию, выводя на экран это пользователю, чтобы собрать их вводы, и затем получении этих вводов, когда диалоговое окно закроется. Мы возьмём тот же самый курс с нашими пользовательскими диалоговыми окнами, хотя мы должны будем сделать немного больше телодвижений, так как мы имеем дело с нашим собственным типом данных и настройками вместо некоторого встроенного типа (как **FXFontDesc**, который мы использовали с **FXFontDialog**).

Важно помнить что присутствие кнопки *Cancel* на диалоговое окно подразумевает своего рода контракт с пользователем. Если пользователь решает он не хотит сохранять изменения, которые он произвёл в настройках, них можно всегда отказаться, отменяя диалоговое окно и это гарантирует, что реальные параметры настройки приложения не будут изменены. По этой причине я никогда не позволяю диалоговому окну непосредственно изменять настройки приложения. Когда надо вывести на экран Привилегированное диалоговое окно или некоторых другой вид пользовательского диалогового окна, я делаю копию текущих настроек и копирую в диалоговое окно. Если пользователь впоследствии щелкает по кнопке OK, чтобы закрыть диалоговое окно, я получаю эту копию и тогда извлекаю необходимую информацию из этого. Если с другой стороны пользователь нажимает кнопку *Cancel*, чтобы отклонить диалоговое окно, я могу только забыть о копии, которую диалоговое окно использовало, и я не должен волноваться об отмене любых изменений.

Чтобы сохранить вещи простыми, давайте сосредоточимся, как мы могли бы обработать настройки для вкладки *Basics*, которая имеет дело с информацией об имени и адресе. Мы собираемся использовать в своих интересах класс **FXDataTarget**, чтобы обработать получение данные из отдельных виджетов в форме. Внутри **PreferencesDialog** будет только использовать хеш экземпляров **FXDataTarget**:

```
@prefs = {
      :first_name => FXDataTarget.new,
      :last_name => FXDataTarget.new,
      :street => FXDataTarget.new,
      :city => FXDataTarget.new,
      :state => FXDataTarget.new,
      :zip_code => FXDataTarget.new
}
```

Теперь давайте создавать содержание для первой страницы, связанной с вкладкой *Basics*. Чтобы сохранить этот код диалогового окна методом *initialize()*, мы поместим это в отдельный метод экземпляра, названный *construct_basics_page()*:

```
def construct_basics_page(page)
      form = FXMatrix.new(page, 2,
            :opts => MATRIX_BY_COLUMNS|LAYOUT_FILL_X)
      FXLabel.new(form, "First:" )
      FXTextField.new(form, 20,
            :target => @prefs[:first_name], :selector => FXDataTarget::ID_VALUE,
            :opts => TEXTFIELD_NORMAL|LAYOUT_FILL_X|LAYOUT_FILL_COLUMN)
      FXLabel.new(form, "Last:" )
      FXTextField.new(form, 20,
            :target => @prefs[:last_name], :selector => FXDataTarget::ID_VALUE,
            :opts => TEXTFIELD_NORMAL|LAYOUT_FILL_X|LAYOUT_FILL_COLUMN)
      FXLabel.new(form, "Street Address:" )
      FXTextField.new(form, 20,
            :target => @prefs[:street], :selector => FXDataTarget::ID_VALUE,
            :opts => TEXTFIELD_NORMAL|LAYOUT_FILL_X|LAYOUT_FILL_COLUMN)
      FXLabel.new(form, "City:" )
      FXTextField.new(form, 20,
            :target => @prefs[:city], :selector => FXDataTarget::ID_VALUE,
            :opts => TEXTFIELD_NORMAL|LAYOUT_FILL_X|LAYOUT_FILL_COLUMN)
      FXLabel.new(form, "State:" )
      states = FXListBox.new(form,
            :target => @prefs[:state], :selector => FXDataTarget::ID_VALUE,
            :opts => (LISTBOX_NORMAL|FRAME_SUNKEN|
                        LAYOUT_FILL_X|LAYOUT_FILL_COLUMN))
      FXLabel.new(form, "Zip Code:" )
      FXTextField.new(form, 10,
            :target => @prefs[:zip_code], :selector => FXDataTarget::ID_VALUE,
            :opts => TEXTFIELD_NORMAL|LAYOUT_FILL_COLUMN)
end
```

Отметьте что каждый из виджетов **FXTextField**, так же как **FXListBox** содержит имя состояния, мы используем одну из целей данных от *@prefs* хеш. Виджеты возьмут свои начальные установки от данных в тех цели данных, и всякий раз, когда пользователь изменяет некоторые настройки в виджете, целевое значение данных будет обновлено автоматически.

Чтобы интегрировать это диалоговое окно с приложением, мы должны были бы добавить команду меню *Preferences...* в одном из меню приложения. Когда эта команда меню вызвана, она создаст новый экземпляр *PreferencesDialog*:

```
dialog = PreferencesDialog.new(self)
```

Затем, нужно инициализировать копию диалогового окна привилегированных данных текущими параметрами настройки приложения:

```
dialog.prefs[:first_name].value = user_name.first_name
dialog.prefs[:last_name].value  = user_name.last_name
dialog.prefs[:street].value     = user_address.street
dialog.prefs[:city].value       = user_address.city
dialog.prefs[:state].value      = user_address.state
dialog.prefs[:zip_code].value   = user_address.zip_code
```

Последний шаг - вызов *execute()* на диалоговом окне, чтобы вывести на экран это. Если *execute()* возвращает не ноль, мы извлечем измененные параметры настройки приложения из копия диалогового окна и применим их к модели:

```
if dialog.execute != 0
      user_name.first_name = dialog.prefs[:first_name].value
      user_name.last_name  = dialog.prefs[:last_name].value
      user_address.street  = dialog.prefs[:street].value
      user_address.city    = dialog.prefs[:city].value
      user_address.state   = dialog.prefs[:state].value
      user_address.zip_code = dialog.prefs[:zip_code].value
end
```

Для наглядности этого примера, мы положились на Ruby хеш, структура действительно исходных данных. В результате код, требуемый для получения данных для диалогового окна довольно многословен. В зависимости от объем данных, с которым Вы имеете дело в пользовательском диалоговом окне, Вы можете счесть полезным создать пользовательские типы данных, которые позволят взаимодействовать с диалоговым окном более компактным способом.

## 14.7 Looking Ahead

Как я отметил в начале книги, это не всестороннее книга по разработке на FXRuby. Я надеюсь, что теперь, когда Вы закончили читать эту книгу, у Вас есть основа, чтобы обратиться к документации и узнать о некоторых из более усовершенствованных аспектов FOX и FXRuby.

Например, одна из многих сложных вещей о FOX, которую мы не рассмотрели вообще - своя поддержка основанной на OpenGL 2-D и 3-D графики приложения. Вы можете использовать виджет **FXRUBY FXGLViewer**, чтобы создать сложный 3-D график сцены, который поддерживает выбор, вращение, изменение масштаба, и много других сложных функций, или Вы можете использовать виджет **FXGLCanvas**, когда Вы нуждаетесь в осуществлении большего контроля над тем, как сцена представлена пользователю.

FOX также обеспечивает много виджетов специального назначения, таких как dials, spinners и sliders, которые Вы можете использовать вместе с другими виджетами как мы рассматривали в Главе 8, *«Создание Простых Виджетов»*, на странице 100. Вы знаете, как искать документацию для классов, чтобы узнать больше об их определенных возможностях, и Вы можете применить методы которым научились в этой книге для того, чтобы ответить на сообщения от этих виджетов и соединить их с целями данных.

В заключение, позвольте мне поощрить Вас присоединиться к нам на списке рассылки по FOX и FXRuby, где Вы можете задать вопросы или совместно использовать Ваши собственные наработки совместно с другими разработчиками программного обеспечения. Раздел 1.4, *«Где Получить Справку»*, на странице 15, обеспечивает информацией о том, как подписаться на эти списки.