5 Создание Вашего проекта

Эта глава включает

- Рефакторинг и как он улучшает программный код
- Разделение Модели (Model) и Представления (View)
- Применение класса Model
- Отладка модуля GUI-программы
- Тестирование событий пользователя

Код GUI-приложения известен своей тяжелой читабельностью, трудностью в сопровождении, и всегда выглядит похожим на длинное, волокнистое и запутанное спагетти. Один примечательный модуль GUI-программы на Python (написанный не на wxPython) содержит следующие слова в своём комментарии: "Почему же, несмотря на все усилия по поддержанию порядка GUI-кода, он всегда выходит похожим на месиво?" Этого не должно происходить. Нет никаких конкретных причин, из-за которых UI-код (интерфейсный код) может быть тяжелым для написания или обслуживания, чем любая другая часть Вашей программы. В этой главе мы обсудим три способа укрощения Вашего UI-кода.

Так как проектный код особенно восприимчив к низкому развитию своей структуры, мы обсудим *рефакторинг* этого кода, что облегчит его читабельность, обслуживание и сопровождение. Другая область, где UI-программист может запутаться – это взаимодействие между кодом представления и базовыми объектами бизнес-логики. Шаблон проекта *Model/View/Controller* (MVC) *(Модель/Представление/ Контроллер)* - это структура для раздельного хранения представления и данных, что позволяет изменять каждый из них без взаимного их влияния друг на друга. Наконец, мы обсудим приёмы испытания модуля с Вашим кодом на wxPython. Хотя все примеры в этой главе будут использовать wxPython, многие из принципов применимы и к любому комплекту инструментов пользовательского интерфейса (кстати, язык Python и комплект инструментов wxPython делают некоторые из этих приёмов особенно изящными).

Дизайн и архитектура Вашего кода определяют проект Вашей системы. А хорошо продуманный проект сделает Ваше приложение более простым для построения и более легким в обслуживании. Рекомендации этой главы помогут Вам спроектировать основательный проект Вашей программы.

5.1 Как рефакторинг помогает улучшить мой код?

Есть много причин, почему у хороших программистов может получиться неудачный интерфейс или код компоновки элементов. Даже простой интерфейс может потребовать много строк кода для вывода на экране всех его элементов. Программисты часто пытаются обойтись использованием единственного метода, и этот метод быстро становится длинным и трудно управляемым. Кроме того, интерфейсный код постоянно совершенствуется и изменяется, что может нанесите ущерб, если Вы не будете внимательно отслеживать все изменения. Поскольку написание кода размещения элементов интерфейса зачастую утомительно, программисту удобнее пользоваться комплектом инструментов для дизайна, генерирующим этот код. Без использования инструментов генерации кода, машинно-генерированный код покажется неуклюжим и трудным для восприятия.

В принципе, не тяжело держать UI-код под контролем. Главное это - *рефакторинг* или непрерывное улучшение проекта и структуры существующего кода. Цель *рефакторинга* - поддержание кода в состоянии, при котором он в будущем легко читается и обслуживается. Таблица 5.1 содержит описание

некоторых принципов, которые нужно иметь в виду при рефакторинге. Помните, что кто-то будет читать и разбирать Ваш код в будущем. Попытайтесь облегчить жизнь другим, в конце концов, это Вам под силу.

Таблица 5.1 Список наиболее важных принципов рефакторинга

Принцип	Описание
Отсутствие дублирования	Вы должны избегать многочисленных сегментов кода с похожим функциональным назначением. Это может вызвать проблемы, когда при эксплуатации функциональное назначению нужно будет изменить.
Одна операция за один раз	Метод должен выполнять одну и только одну операцию. Другие операции должны быть перемещены в отдельные методы. Методы должны быть предельно короткими.
Ограничение глубины вложенности	Попытайтесь использовать не более двух или трёх уровней вложенности кода. Глубоко вложенный код является хорошим кандидатом для отдельного метода.
Сокращение количества литералов	Строчные и числовые литералы (константы) должны быть сведены к минимуму. Хорошим приёмом является отделение литеральных данные от основной части Вашего кода и хранение их в списке или словаре.

Некоторые из этих принципов особенно важны в коде Python. Так как синтаксис Python основан на отступах, небольшие, компактные методы очень легко читаются. Более длинные методы, однако, могут быть трудны для разбора, особенно если они не в состоянии поместиться на одном экране. Так же, глубокая вложенность в Python может затруднить отслеживание начала и окончания кодовых блоков. Тем не менее, Python является особенно хорошим языком для избежания дублирования, главным образом из-за удобства с которым функции и методы могут быть переданы как аргументы.

5.1.1 Пример рефакторинга

Чтобы показать, как эти принципы работают, мы с Вами рассмотрим пример рефакторинга. Рисунок 5.1 показывает окно, которое может быть использовано как интерфейс к базе данных, подобной Microsoft Access. Его вид чуть-чуть сложнее, чем те, которые мы видели до сих пор, но оно остается совсем простым относительно стандарта реальных приложений. Листинг 5.1 показывает слабо структурированный способ вывести окно рисунка 5.1. Когда люди говорят, что UI-код стал похож на «месиво», они знают, что говорят. Может быть всё несколько преувеличено, но это отражает проблему, с которой Вы можете

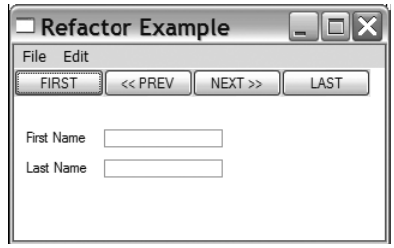

Рисунок 5.1 Образец окна как пример рефакторинга

столкнуться в коде размещения элементов интерфейса. И, несомненно, это отражает проблему, в которую попадаю я сам при записи кода размещения.

Листинг 5.1 Нерефакторный способ воспроизвести окно из рисунка 5.1

```
#!/usr/bin/env python

import wx

class RefactorExample(wx.Frame):

    def __init__(self, parent, id):
        wx.Frame.__init__(self, parent, id, 'Refactor Example',
                size=(340, 200))
        panel = wx.Panel(self, -1)
        panel.SetBackgroundColour("White")
        prevButton = wx.Button(panel, -1, "<< PREV", pos=(80, 0))
        self.Bind(wx.EVT_BUTTON, self.OnPrev, prevButton)
        nextButton = wx.Button(panel, -1, "NEXT >>", pos=(160, 0))
        self.Bind(wx.EVT_BUTTON, self.OnNext, nextButton)
        self.Bind(wx.EVT_CLOSE, self.OnCloseWindow)

        menuBar = wx.MenuBar()
        menu1 = wx.Menu()
        openMenuItem = menu1.Append(-1, "&Open", "Copy in status bar")
        self.Bind(wx.EVT_MENU, self.OnOpen, openMenuItem)
        quitMenuItem = menu1.Append(-1, "&Quit", "Quit")
        self.Bind(wx.EVT_MENU, self.OnCloseWindow, quitMenuItem)
        menuBar.Append(menu1, "&File")
        menu2 = wx.Menu()
        copyItem = menu2.Append(-1, "&Copy", "Copy")
        self.Bind(wx.EVT_MENU, self.OnCopy, copyItem)
        cutItem = menu2.Append(-1, "C&ut", "Cut")
        self.Bind(wx.EVT_MENU, self.OnCut, cutItem)
        pasteItem = menu2.Append(-1, "Paste", "Paste")
        self.Bind(wx.EVT_MENU, self.OnPaste, pasteItem)
        menuBar.Append(menu2, "&Edit")
        self.SetMenuBar(menuBar)

        static = wx.StaticText(panel, wx.NewId(), "First Name",
                pos=(10, 50))
        static.SetBackgroundColour("White")
        text = wx.TextCtrl(panel, wx.NewId(), "", size=(100, -1),
                pos=(80, 50))

        static2 = wx.StaticText(panel, wx.NewId(), "Last Name",
                pos=(10, 80))
        static2.SetBackgroundColour("White")
        text2 = wx.TextCtrl(panel, wx.NewId(), "", size=(100, -1),
                pos=(80, 80))

        firstButton = wx.Button(panel, -1, "FIRST")
        self.Bind(wx.EVT_BUTTON, self.OnFirst, firstButton)
```

```
        menu2.AppendSeparator()
        optItem = menu2.Append(-1, "&Options...", "Display Options")
        self.Bind(wx.EVT_MENU, self.OnOptions, optItem)

        lastButton = wx.Button(panel, -1, "LAST", pos=(240, 0))
        self.Bind(wx.EVT_BUTTON, self.OnLast, lastButton)

    # Just grouping the empty event handlers together
    def OnPrev(self, event): pass
    def OnNext(self, event): pass
    def OnLast(self, event): pass
    def OnFirst(self, event): pass
    def OnOpen(self, event): pass
    def OnCopy(self, event): pass
    def OnCut(self, event): pass
    def OnPaste(self, event): pass
    def OnOptions(self, event): pass

    def OnCloseWindow(self, event):
        self.Destroy()

if __name__ == '__main__':
    app = wx.PySimpleApp()
    frame = RefactorExample(parent=None, id=-1)
    frame.Show()
    app.MainLoop()
```

Давайте, определим, как работает этот код относительно принципов таблицы 5.1. Хорошо то, что здесь нет глубокой вложенности. Плохо, что другие три идеи, указанные в таблице 5.1 совсем не выполняются. Таблица 5.2 суммирует все способы, в которых рефакторинг может улучшить данный код.

Таблица 5.2 Возможности рефакторинга в листинге 5.1

Принцип	Проблема в коде
Отсутствие дублирования	Многократно дублируются несколько структур, включая: «добавить кнопку и назначить действие», «добавить пункт меню и назначить действие» и «создать пару заголовок/текст».
Одна операция за один раз	Этот код делает несколько вещей. Дополнительно к основной настройке фрейма, он создает строку меню (menu bar), добавляет кнопки и добавляет текстовые поля. Хуже, что эти три функции перемешали код, как будто только что были добавлены последние изменения внизу метода.
Сокращение количества литералов	Каждая кнопка, пункт меню и текстовый блок содержит литеральную строку и литерал указан в конструкторе.

5.1.2 Начало рефакторинга

Листинг 5.2 содержит код, использованный только для создания панели кнопок из предшествующего листинга. В качестве первого шага рефакторинга мы выделим этот код в свой собственный метод.

Листинг 5.2 Панель кнопок как отдельный метод

```
def createButtonBar(self):
    firstButton = wx.Button(panel, -1, "FIRST")
    self.Bind(wx.EVT_BUTTON, self.OnFirst, firstButton)
    prevButton = wx.Button(panel, -1, "<< PREV", pos=(80, 0))
    self.Bind(wx.EVT_BUTTON, , self.OnPrev, prevButton)
    nextButton = wx.Button(panel, -1, "NEXT >>", pos=(160, 0))
    self.Bind(wx.EVT_BUTTON, self.OnNext, nextButton)
    lastButton = wx.Button(panel, -1, "LAST", pos=(240, 0))
    self.Bind(wx.EVT_BUTTON, self.OnLast, lastButton)
```

С таким обособленным кодом легко увидеть, сколько общего имеют все определения кнопок. Мы можем вынести (факторизовать) данную общую часть в отдельный метод и просто повторно его вызывать, как показано в листинге 5.3:

Листинг 5.3 Улучшенный и общий методы панели кнопок

```
def createButtonBar(self, panel):
    self.buildOneButton(panel, "First", self.OnFirst)
    self.buildOneButton(panel, "<< PREV", self.OnPrev, (80, 0))
    self.buildOneButton(panel, "NEXT >>", self.OnNext, (160, 0))
    self.buildOneButton(panel, "Last", self.OnLast, (240, 0))

def buildOneButton(self, parent, label, handler, pos=(0,0)):
    button = wx.Button(parent, -1, label, pos)
    self.Bind(wx.EVT_BUTTON, handler, button)
    return button
```

Есть несколько преимуществ применения второго примера вместо первого. Прежде всего, назначение кода становится ясным сразу после его прочтения - короткие методы с выразительными именами имеют большое значение в раскрытии смысла кода. Второй пример также лишен всех локальных переменных, которые нужны для связи с идентификаторами (ID) (надо сказать, Вы могли бы также избавиться от локальных переменных при помощи фиксации (hardwiring) идентификаторов, но это может вызвать проблемы их дублирования). Это полезно, поскольку делает код менее сложным, и также, поскольку это почти устраняет общую ошибку вырезания и вставки нескольких строк кода, забывая при этом изменять все имена переменных. (В реальном приложении Вам, вероятно, понадобится хранить кнопки как отдельные переменные, чтобы позже иметь к ним доступ, но в данном примере в этом нет необходимости.) Кроме того,

метод `buildOneButton()` может быть легко перемещен в модуль для утилит и повторно использован в других фреймах или других проектах. Комплект утилит общего применения – это та вещь, которую всегда полезно иметь.

5.1.3 Дальнейший рефакторинг

Сделав существенное улучшение, мы могли бы на этом остановиться. Но в коде все еще остается много «магических» литералов (жестко запрограммированных констант, которые используются в различных местах). Литералы указания позиции могут привести к ошибкам, прежде всего, когда в панель добавляется другая кнопка, особенно если новая кнопка размещается в середине панели. Итак, продвигаемся на один шаг вперед, и отделяем литеральные данные от обрабатывающего кода. Листинг 5.4 показывает другой механизм управления данными для создания кнопок.

Листинг 5.4 Создание кнопок при помощи изолированных от кода данных

```
def buttonData(self):
    return (("First", self.OnFirst),
            ("<< PREV", self.OnPrev),
            ("NEXT >>", self.OnNext),
            ("Last", self.OnLast))

def createButtonBar(self, panel, yPos=0):
    xPos = 0
    for eachLabel, eachHandler in self.buttonData():
        pos = (xPos, yPos)
        button = self.buildOneButton(panel, eachLabel,
                 eachHandler, pos)
        xPos += button.GetSize().width

def buildOneButton(self, parent, label, handler, pos=(0,0)):
    button = wx.Button(parent, -1, label, pos)
    self.Bind(wx.EVT_BUTTON, handler, button)
    return button
```

В листинге 5.4 данные для отдельных кнопок хранятся в виде встроенного кортежа внутри метода `buttonData()`. Такой выбор структуры данных и использование константного метода не случайно. Данные можно было бы сохранить в виде переменной класса или модуля, что покажется лучше, чем возвращать их как результат метода, или они могли бы быть сохранены во внешнем файле. Главное преимущество в использование метода состоит в сравнительно простом переходе, т.е. если вдруг Вы захотите сохранить кнопочные данные в другом месте, то просто измените метод, так чтобы вместо возвращения константы, он возвращал внешние данные.

Метод `createButtonBar()` итеративно обрабатывает возвращенный `buttonData()` список и создает каждую кнопку из этих данных. Теперь метод автоматически при прохождении списка вычисляет позицию кнопок по оси x. Это

полезное, поскольку гарантируется, что порядок кнопок в коде будет идентичен их порядку на экране, к тому же делает код более ясным и менее подверженным появлению ошибок. Если теперь Вам нужно добавить кнопку в середину панели, Вы можете просто добавить данные в середину списка, и код гарантирует, что кнопка будет установлена правильно.

Разделение данных имеет и другие преимущества. В более сложном примере, данные могли бы храниться вовне - в ресурсе или файле XML. Это позволяет изменять интерфейс даже без пересмотра кода и также облегчает интернационализацию, упрощая изменение текста. На данный момент мы все еще жестко ограничены шириной кнопки, но это можно также легко добавить к методу данных. (В действительности, мы должны, вероятно, использовать входящий в wxPython объект `Sizer`, который рассматривается в главе 11). `createButtonBar` стал теперь полезным методом и может легко использоваться в другом фрейме или проекте, не обязательно связанном с обработкой базы данных.

После выполнения тех же шагов объединения, факторинга общего процесса и разделения данных для меню и кода текстового поля, получим результат, показанный в листинге 5.5.

Листинг 5.5 Рефакторизованный пример

```
#!/usr/bin/env python

import wx

class RefactorExample(wx.Frame):

    def __init__(self, parent, id):
        wx.Frame.__init__(self, parent, id, 'Refactor Example',
                size=(340, 200))
        panel = wx.Panel(self, -1)
        panel.SetBackgroundColour("White")
        self.Bind(wx.EVT_CLOSE, self.OnCloseWindow)
        self.createMenuBar()
        self.createButtonBar(panel)
        self.createTextFields(panel)

    def menuData(self):
        return (("&File",
                ("&Open", "Open in status bar", self.OnOpen),
                ("&Quit", "Quit", self.OnCloseWindow)),
            ("&Edit",
                ("&Copy", "Copy", self.OnCopy),
                ("C&ut", "Cut", self.OnCut),
                ("&Paste", "Paste", self.OnPaste),
                ("", "", ""),
                ("&Options...", "DisplayOptions", self.OnOptions)))
```

Упрощенный метод инициализации

Данные для меню

```
def createMenuBar(self):
    menuBar = wx.MenuBar()
    for eachMenuData in self.menuData():
        menuLabel = eachMenuData[0]
        menuItems = eachMenuData[1:]
        menuBar.Append(self.createMenu(menuItems), menuLabel)
    self.SetMenuBar(menuBar)

def createMenu(self, menuData):
    menu = wx.Menu()
    for eachLabel, eachStatus, eachHandler in menuData:
        if not eachLabel:
            menu.AppendSeparator()
            continue
        menuItem = menu.Append(-1, eachLabel, eachStatus)
        self.Bind(wx.EVT_MENU, eachHandler, menuItem)
    return menu

def buttonData(self):
    return (("First", self.OnFirst),
            ("<< PREV", self.OnPrev),
            ("NEXT >>", self.OnNext),
            ("Last", self.OnLast))

def createButtonBar(self, panel, yPos = 0):
    xPos = 0
    for eachLabel, eachHandler in self.buttonData():
        pos = (xPos, yPos)
        button = self.buildOneButton(panel, eachLabel,
            eachHandler, pos)
        xPos += button.GetSize().width

def buildOneButton(self, parent, label, handler, pos=(0,0)):
    button = wx.Button(parent, -1, label, pos)
    self.Bind(wx.EVT_BUTTON, handler, button)
    return button

def textFieldData(self):
    return (("First Name", (10, 50)),
            ("Last Name", (10, 80)))

def createTextFields(self, panel):
    for eachLabel, eachPos in self.textFieldData():
        self.createCaptionedText(panel, eachLabel, eachPos)

def createCaptionedText(self, panel, label, pos):
    static = wx.StaticText(panel, wx.NewId(), label, pos)
    static.SetBackgroundColour("White")
    textPos = (pos[0] + 75, pos[1])
    wx.TextCtrl(panel, wx.NewId(), "", size=(100, -1),
       pos=textPos)

# Just grouping the empty event handlers together
def OnPrev(self, event): pass
```

Создание меню

Данные панели кнопок

Создание кнопок

Текстовые данные

Создание текста

```
        def OnNext(self, event): pass
        def OnLast(self, event): pass
        def OnFirst(self, event): pass
        def OnOpen(self, event): pass
        def OnCopy(self, event): pass
        def OnCut(self, event): pass
        def OnPaste(self, event): pass
        def OnOptions(self, event): pass
        def OnCloseWindow(self, event):
            self.Destroy()

if __name__ == '__main__':
    app = wx.PySimpleApp()
    frame = RefactorExample(parent=None, id=-1)
    frame.Show()
    app.MainLoop()
```

Усилия для преобразования листинга 5.1 в листинг 5.5 были минимальными, но вознаграждение колоссальные – базовый код стал значительно более ясным и более устойчивым к появлению ошибок. Представление кода логически соответствует формату данных. Исключены несколько основных путей, при которых слабо структурный код может привести к ошибкам, таким как после серий копирования и вставки при создании нового объекта. Основной объем функциональности может теперь легко быть перемещен в суперкласс или во вспомогательный модуль, что сохраняет код для будущего использования. Как дополнительный бонус, разделение данных позволяет свободно использовать проект в качестве шаблона с разными данными, включая интернационализацию данных.

Ключ к успешному рефакторингу лежит в небольших приращениях при написании Вашего кода. Пока Вы не предпримете усилий по регулярному улучшению плохо организованного кода, он, подобно грязной посуде, может быстро нагромоздиться в беспорядочную массу. Как только Вы прочувствуете, что действующий код является только промежуточным шагом по отношению к конечной цели хорошо факторизованного рабочего кода, Вы сможете превратить рефакторизацию в часть Вашего обычного процесса разработки.

Тем не менее, даже после проведенной в листинге 5.5 рефакторизации, все еще не достает чего-то важного: фактических данных пользователя. В основном, Ваше приложение будет зависеть от управляющих данных в откликах на запросы пользователя. Структура Вашей программы может расшириться, что придаст ей гибкости и стабильности. Шаблон MVC является общепринятым стандартом управления взаимодействием между интерфейсом и данными.

5.2 Как в программе раздельно хранить Модель и Представление?

Начиная примерно с 1970 года, появившись в концептуальном языке Smalltalk-80, модель MVC вероятно является старейшей широко используемой объектно-ориентированной проектной моделью. Она также является одной из общепринятых моделей, так или иначе поддерживаемых почти каждым пакетом разработчика GUI (не говоря уже о большом числе других систем, например web приложений). Модель MVC является стандартом для структурирования программ, которые манипулируют информацией и отображают ее.

5.2.1 Что собой представляет система Модель-Представление-Контроллер?

Система MVC имеет три подсистемы. Подсистема *Модель* содержит то, что часто называют деловой логикой, или все данные и информацию Вашей системы. Подсистема *Представление* содержит объекты, которые отображают данные, а *Контроллер* управляет взаимодействием с пользователем и связывает между собой *Модель* и *Представление*. Таблица 5.3 резюмирует эти компоненты.

Таблица 5.3 Компоненты стандарта архитектуры MVC

Принцип	Проблема в коде
Модель	Деловая логика. Содержит все обрабатываемые системой данные. Она может включать интерфейс во внешнее хранилище, например, базу данных. Обычно модель предоставляет всего лишь открытый API к другим компонентам.
Представление	Код для отображения. Это виджеты (widget) – элементы интерфейса, размещающие информацию в удобном для пользователя виде. В wxPython, в основном, вся часть подсистемы представления реализована в иерархии `wx.Window`.
Контроллер	Логика взаимодействия. Код, который принимает события пользователя и гарантирует, что они будут обработаны системой. В wxPython эта подсистема представлена иерархией `wx.EvtHandler`.

В большинстве современных пакетов для разработки интерфейса пользователя компоненты Представления и Контроллера слегка переплетены. Дело в том, что на экране должны быть отображены компоненты самого Контроллера, и к тому же часто Вам необходимы виджеты, которые отображают данные и также должны реагировать на события пользователя. В wxPython эта взаимосвязь сохранена изначально, и все объекты `wx.Window` являются также подклассами `wx.EvtHandler`, и это означает, что они функционируют и как элементы Представления, и как элементы Контроллера. В отличие от этого, большинство каркасов web-

приложений имеют более строгое разделение между Представлением и Контроллером, так как логика взаимодействия происходит вне пользовательских форм на стороне сервера.

Рисунок 5.2 показывает, как в архитектуре MVC проходят данные и информация.

Сообщения о событиях обрабатываются системой Контроллера, которая отправляет их соответствующим потребителям. Как мы видели в главе 3, wxPython управляет этим механизмом, используя метод `ProcessEvent()`, относящийся к `wx.EvtHandler`. В проекте, который строго соответствует модели MVC, Ваши функции-обработчики могут быть фактически объявлены в отдельном объекте-контроллере, а не в самом фреймовом классе.

В ответ на событие, объекты модели могут выполнять некоторую обработку прикладных данных. Когда эта обработка завершается, модель посылает уведомление об обновлении. И если это объект контроллера, уведомление обычно возвращается контроллеру, и он уведомляет соответствующие объекты представления, чтобы они выполнили самостоятельное обновление. В небольшой системе или системе с упрощенной архитектурой уведомление часто принимается непосредственно объектами представления. В wxPython используется строгий механизм обновления данных, начиная с модели и заканчивая Вами. Его возможности включают инициацию моделью или контроллером прикладного события wxPython, при этом модель поддерживает список объектов, которые получают уведомления об обновлении, или она содержит представления, которые зарегистрировали себя в модели.

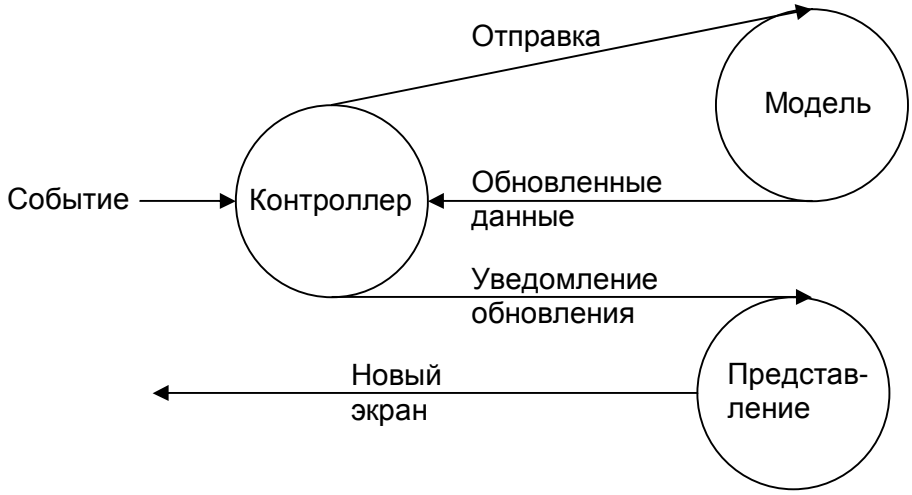

Рисунок 5.2 Поток данных в модели MVC

Ключевая особенность успешного MVC проекта состоит не в обеспечении того, чтобы каждый объект знал о любом другом объекте. Наоборот, хорошая MVC программа явно скрывает информацию об одной части программы от других ее

частей. Цель для такой системы состоит в минимальном взаимодействии, и к тому же с помощью строго определенного набора методов. Особенно компонент Модели должен быть полностью изолирован от Представления и Контроллера. Вы должны иметь возможность выполнять произвольные изменения в любой из этих систем, не изменяя Ваши классы Модели. Замечательно, если Вы даже будете использовать одни и те же классы Модели для управления интерфейсами, не относящимися к wxPython, но вот, что могло бы помешать, так это использование событий wxPython для нотификации изменений.

Вам нужно сделать произвольные изменения в реализации объектов Модели со стороны Представления, не изменяя Представления или Контроллера. До тех пор, пока Представление будет зависеть от существующих глобальных методов, оно не должно когда-нибудь увидеть частной внутренней организации Модели. Правда, в Python это осуществляться трудно, но есть один способ, который поможет в этом, и он состоит в том, чтобы создать абстрактный класс Модели, который определяет API, что может увидеть Представление. Подклассы Модели могут действовать или в качестве представителей для внутреннего класса, который может быть изменен, или могут просто содержать работающие элементы прямо внутри себя. Первая возможность более структурирована, вторая легче в реализации.

В следующей части мы рассмотрим один из встроенных в wxPython классов Модели – `wx.grid.PyGridTableBase`. Этот класс делает возможным использование в пределах проектного каркаса MVC элемента управления сетки (grid). После этого мы взглянем на построение и использование пользовательского класса модели для пользовательского виджета.

5.2.2 Модель wxPython: PyGridTableBase

Класс `wx.grid.Grid` является элементом управления wxPython для представленной электронной таблицы в виде строк и колонок. Вы вероятно знакомы с основной концепцией, но рисунок 5.3 показывает, как это выглядит в wxPython.

Рисунок 5.3 Пример сетки в wxPython

Элемент управления сетка (grid) имеет много интересных возможностей, включая возможность создания пользовательской прорисовки и редакторов, а также перетаскиваемые строки и колонки. Эти возможности будут подробно обсуждены в главе 13. В этой главе, мы рассмотрим только основы и покажем использование модели для заполнения сетки. Листинг 5.6 показывает простой немодельный способ установки в сетке значений ячейки. В этом случае, значениями сетки выступает состав команды Chicago Cubs 1984.

Листинг 5.6 Заполнение сетки без использования модели

```
import wx
import wx.grid
```

```
class SimpleGrid(wx.grid.Grid):
    def __init__(self, parent):
        wx.grid.Grid.__init__(self, parent, -1)
        self.CreateGrid(9, 2)
        self.SetColLabelValue(0, "First")
        self.SetColLabelValue(1, "Last")
        self.SetRowLabelValue(0, "CF")
        self.SetCellValue(0, 0, "Bob")
        self.SetCellValue(0, 1, "Dernier")
        self.SetRowLabelValue(1, "2B")
        self.SetCellValue(1, 0, "Ryne")
        self.SetCellValue(1, 1, "Sandberg")
        self.SetRowLabelValue(2, "LF")
        self.SetCellValue(2, 0, "Gary")
        self.SetCellValue(2, 1, "Matthews")
        self.SetRowLabelValue(3, "1B")
        self.SetCellValue(3, 0, "Leon")
        self.SetCellValue(3, 1, "Durham")
        self.SetRowLabelValue(4, "RF")
        self.SetCellValue(4, 0, "Keith")
        self.SetCellValue(4, 1, "Moreland")
        self.SetRowLabelValue(5, "3B")
        self.SetCellValue(5, 0, "Ron")
        self.SetCellValue(5, 1, "Cey")
        self.SetRowLabelValue(6, "C")
        self.SetCellValue(6, 0, "Jody")
        self.SetCellValue(6, 1, "Davis")
        self.SetRowLabelValue(7, "SS")
        self.SetCellValue(7, 0, "Larry")
        self.SetCellValue(7, 1, "Bowa")
        self.SetRowLabelValue(8, "P")
        self.SetCellValue(8, 0, "Rick")
        self.SetCellValue(8, 1, "Sutcliffe")

class TestFrame(wx.Frame):
    def __init__(self, parent):
        wx.Frame.__init__(self, parent, -1, "A Grid",
                size=(275, 275))
        grid = SimpleGrid(self)

if __name__ == '__main__':
    app = wx.PySimpleApp()
    frame = TestFrame(None)
    frame.Show(True)
    app.MainLoop()
```

В листинге 5.6, мы имеем класс `SimpleGrid`, подкласс класса wxPython `wx.grid.Grid`. Как упомянуто ранее, `wx.grid.Grid` имеет уйму методов, которые мы собираемся обсудить позже. Сейчас же, мы сфокусируемся на методах `SetRowLabelValue()`, `SetColLabelValue()` и `SetCellValue()`, которые действительно устанавливают отображаемые в сетке значения. Как видите, при сравнении рисунка 5.3 и листинга 5.6, метод `SetCellValue()` берет индекс строки,

индекс колонки и значение, тогда как другие два метода берут только индекс и значение. При назначении индексов ячеек, заголовки строк и колонок не считаются частью сетки.

Этот код непосредственно помещает в сетку значения при помощи устанавливающих методов. До тех пор, пока этот метод будет оставаться таким прямолинейным, он может стать статичным и будет подвержен появлению ошибок в больших сетках. И даже, если мы создадим вспомогательные служебные методы, код все еще будет испытывать проблему, которую мы видели в разделе рефакторинга этой главы. Данные будут смешаны с кодом отображения, что затруднит будущую модификацию кода, например, добавление столбца или полный обмен данных.

Решением будет `wx.grid.PyGridTableBase`. Как и в других классах, которые мы видели до сих пор, префикс `Py` указывает на то, что это – Python-оболочка (wrapper) для класса C++. Подобно классу `PyEvent`, который мы видели в главе 3, класс `PyGridTableBase` реализован, как простая Python оболочка для класса C++ wxWidgets для возможности объявления подклассов Python. `PyGridTableBase` - класс модели для сетки. То есть, он содержит методы, которые объект сетки использует для своей прорисовки без сведений о внутренней структуре этих данных.

Методы PyGridTableBase

`wx.grid.PyGridTableBase` имеет много методов, с большинством из которых Вы явно не будете иметь дело. Это абстрактный класс и его экземпляр не может создаваться непосредственно. Каждый раз при создании `PyGridTableBase` Вы должны обеспечивать реализацию пяти необходимых методов. Таблица 5.4 описывает эти методы.

Таблица 5.3 Методы, необходимые для wx.grid.PyGridTableBase

Метод	Описание
GetNumberRows()	Возвращает целое, показывающее число строк сетки
GetNumberCols()	Возвращает целое, показывающее число колонок сетки
IsEmptyCell(row, col)	Возвращает True, если элемент с индексом (row, col) пустой
GetValue(row, col)	Возвращает значение, отображаемое в ячейке (row, col)
SetValue(row, col, value)	Устанавливает значение ячейки (row, col). Если Вам необходима модель «только чтение», этот метод нужно включить с директивой `pass`.

Таблица закрепляется за сеткой при помощи метода сетки `SetTable()`. После того, как это свойство будет установлено, объект сетки будет вызывать методы таблицы для получения информации, необходимой ему для прорисовки сетки. Сетке больше не нужно явно устанавливать значения методами сетки.

Использование PyGridTableBase

В основном, имеется два способа использования `PyGridTableBase`. У Вас может быть явно класс модели, который будет подклассом `PyGridTableBase`, или Вы можете создать отдельный подкласс `PyGridTableBase`, подключаемый к Вашему фактическому классу модели. Первая возможность легче и имеет смысл, когда Ваши данные не очень сложны. Вторая возможность осуществляет жесткое разделение между Моделью и Представлением, которое предпочтительно в случае сложных данных. Вторая возможность также предпочтительна, если у Вас уже есть существующий класс данных, который Вы хотите адаптировать к wxPython, поэтому Вы сможете создать таблицу без изменения существующего кода. В этом разделе мы покажем пример обеих способов.

Использование PyGridTableBase: подкласс, зависящий от приложения

Наш первый пример в качестве модели будет использовать зависящий от приложения подкласс `PyGridTableBase`. Это работает, поскольку наш пример сравнительно прост, так что мы можем непосредственно включить данные в класс, производный от `PyGridTableBase`. Мы установим фактические данные в двумерном списке Python и настроим другие методы на чтение этого списка. Листинг 5.7 показывает состав команды Cubs, сгенерированный из класса Модели.

Листинг 5.7 Таблица, сгенерированная из модели PyGridTableBase

```
import wx
import wx.grid

class LineupTable(wx.grid.PyGridTableBase):

    data = (("CF", "Bob", "Dernier"), ("2B", "Ryne", "Sandberg"),
            ("LF", "Gary", "Matthews"), ("1B", "Leon", "Durham"),
            ("RF", "Keith", "Moreland"), ("3B", "Ron", "Cey"),
            ("C", "Jody", "Davis"), ("SS", "Larry", "Bowa"),
            ("P", "Rick", "Sutcliffe"))

    colLabels = ("Last", "First")

    def __init__(self):
        wx.grid.PyGridTableBase.__init__(self)

    def GetNumberRows(self):
        return len(self.data)

    def GetNumberCols(self):
        return len(self.data[0]) - 1

    def GetColLabelValue(self, col):
        return self.colLabels[col]

    def GetRowLabelValue(self, row):
```

```
        return self.data[row][0]

    def IsEmptyCell(self, row, col):
        return False

    def GetValue(self, row, col):
        return self.data[row][col + 1]

    def SetValue(self, row, col, value):
        pass

class SimpleGrid(wx.grid.Grid):
      def __init__(self, parent):
            wx.grid.Grid.__init__(self, parent, -1)
            self.SetTable(LineupTable())

class TestFrame(wx.Frame):
      def __init__(self, parent):
            wx.Frame.__init__(self, parent, -1, "A Grid",
                  size=(275, 275))
            grid = SimpleGrid(self)

if __name__ == '__main__':
      app = wx.PySimpleApp()
      frame = TestFrame(None)
      frame.Show(True)
      app.MainLoop()
```

Таблица устанавливается здесь

В листинге 5.7 мы определили все методы, необходимые для `PyGridTableBase`, плюс дополнительные методы `GetColLabelValue()` и `GetRowLabelValue()`. Надеемся, Вы не слишком удивитесь, если узнаете, что эти методы позволяют определить метки соответственно для колонки и строки сетки. Так же, как и в подразделе рефакторинга, результат использования класса модели состоит в отделении данных от их отображения. Мы здесь также преобразовали данные в более структурированный формат, которым они могут быть легко перенесены во внешний файл или ресурс (особенно просто здесь может быть использована база данных).

Использование PyGridTableBase: основной пример

Фактически, мы уже почти имеем базовую таблицу, которая способна прочитать любой двумерный список Python. Листинг 5.8 показывает, как должна выглядеть базовая модель.

Листинг 5.8 Базовая таблица для двумерных списков

```
import wx
import wx.grid

class GenericTable(wx.grid.PyGridTableBase):
```

```
    def __init__(self, data, rowLabels=None, colLabels=None):
        wx.grid.PyGridTableBase.__init__(self)
        self.data = data
        self.rowLabels = rowLabels
        self.colLabels = colLabels

    def GetNumberRows(self):
        return len(self.data)

    def GetNumberCols(self):
        return len(self.data[0])

    def GetColLabelValue(self, col):
        if self.colLabels:
            return self.colLabels[col]

    def GetRowLabelValue(self, row):
        if self.rowLabels:
            return self.rowLabels[row]

    def IsEmptyCell(self, row, col):
        return False

    def GetValue(self, row, col):
        return self.data[row][col]

    def SetValue(self, row, col, value):
        pass
```

Класс `GenericTable` берет двумерный список данных и дополнительный список заголовков колонки и/или столбца. Он годится для импортирования в любую программу wxPython. Как показано в листинге 5.9, с легким изменением в формате данных, теперь мы можем использовать эту базовую таблицу для вывода состава команды.

Листинг 5.9 Отображение состава команды с помощью базовой таблицы

```
import wx
import wx.grid
import generictable

data = (("Bob", "Dernier"), ("Ryne", "Sandberg"),
        ("Gary", "Matthews"), ("Leon", "Durham"),
        ("Keith", "Moreland"), ("Ron", "Cey"),
        ("Jody", "Davis"), ("Larry", "Bowa"),
        ("Rick", "Sutcliffe"))

colLabels = ("Last", "First")
rowLabels = ("CF", "2B", "LF", "1B", "RF", "3B", "C", "SS", "P")

class SimpleGrid(wx.grid.Grid):

    def __init__(self, parent):
```

```
        wx.grid.Grid.__init__(self, parent, -1)
        tableBase = genericTable.GenericTable(data, rowLabels,
                    colLabels)
        self.SetTable(tableBase)

class TestFrame(wx.Frame):

    def __init__(self, parent):
        wx.Frame.__init__(self, parent, -1, "A Grid",
        size=(275, 275))
        grid = SimpleGrid(self)

if __name__ == '__main__':
    app = wx.PySimpleApp()
    frame = TestFrame(None)
    frame.Show(True)
    app.MainLoop()
```

Использование PyGridTableBase: самостоятельный класс Модели

Рискуя повториться, возможно имеет смысл показать здесь еще один способ использования `PyGridTableBase`. Существует вторая упомянутая ранее возможность, при которой данные хранятся в доступном из `PyGridTableBase` отдельном классе модели. Здесь очень полезны возможности самопроверки Python, позволяющие Вам составить список атрибутов, отображаемых в каждом столбце, а затем использовать встроенную функцию `getattr()` для извлечения фактического значения. В этом случае модель получает список элементов. Структурирование в wxPython Вашей программы с отдельными объектами модели имеет одно большое преимущество. При нормальных обстоятельствах Вы можете просто один раз вызвать для сетки метод `SetTable()`, т.е. если Вам нужно изменить таблицу, необходимо создать новую сетку, и это не очень-то удобно. Тем не менее, если, как в следующем примере, Ваш класс `PyGridTableBase` хранит ссылки на экземпляры Вашего реального класса данных, Вы можете обновить таблицу новыми данными, просто изменяя в таблице основной объект данных.

Листинг 5.10 показывает использование вместе с `PyGridTableBase` отдельного класса данных для отображения записей командного состава (остальной код собственно фрейма и создания данных мы сохранили прежним).

Листинг 5.10 Отображение состава команды с помощью класса данных пользователя

```
import wx
import wx.grid

class LineupEntry:

    def __init__(self, pos, first, last):
        self.pos = pos
        self.first = first
        self.last = last
```

```
class LineupTable(wx.grid.PyGridTableBase):

    colLabels = ("First", "Last")
    colAttrs = ("first", "last")

    def __init__(self, entries):
        wx.grid.PyGridTableBase.__init__(self)
        self.entries = entries

    def GetNumberRows(self):
        return len(self.entries)

    def GetNumberCols(self):
        return 2

    def GetColLabelValue(self, col):
        return self.colLabels[col]

    def GetRowLabelValue(self, row):
        return self.entries[row].pos

    def IsEmptyCell(self, row, col):
        return False

    def GetValue(self, row, col):
        entry = self.entries[row]
        return getattr(entry, self.colAttrs[col])

    def SetValue(self, row, col, value):
        pass
```


1. Этот список содержит атрибуты, которые должны быть указаны для отображения колонкой своих значений.
2. Модель берет список записей, где каждая запись является экземпляром класса `LineupEntry`. (Мы не производим здесь какой-либо проверки на ошибки или проверки достоверности).
3. Для получения заголовка колонки мы ищем в соответствующей строке атрибут `pos`.
4. Сначала по номеру строки получаем соответствующую запись. Из списка в строке 1 берем атрибут, а затем используем встроенный метод `getattr()` для ссылки на фактическое значение. Этот механизм обеспечивает гибкость даже в случае, когда Вы не знаете, указывает ли имя на атрибут или метод, проверяя, является ли `<объект>.<атрибут>` вызываемым. Если это так, тогда вызовите его, используя обычный функциональный синтаксис Python, и Вы получите возвращаемое значение.

5.2.3 Пользовательская модель

Основная идея создания Ваших объектов модели проста. Создавайте Ваши классы данных, не думая о том, как они будут отображаться. Затем определите общедоступный (public) интерфейс для того класса, который будет доступен объектам отображения. Очевидно то, что формат этого общедоступного определения, будет определять размер и сложность проекта. В небольшом проекте, с простыми объектами, вероятно, достаточно поступить просто, и разрешить объектам Представления прямой доступ к атрибутам модели. В более сложном объекте, Вы, может быть, захотите определить конкретные методы или создать отдельный класс модели, являющийся тем единственным, что видно Представлению (как мы и поступили в листинге 5.10).

Вам также необходим определенный механизм извещения Представления об изменениях в модели. Листинг 5.11 показывает простой абстрактный базовый класс, который Вы можете использовать в качестве родительского класса для любого из Ваших классов модели. Вы можете представить его, как аналог `PyGridTableBase` для использования, когда отображение не является сеткой.

Листинг 5.11 Модель пользователя для обновления представления

```
class AbstractModel(object):

    def __init__(self):
        self.listeners = []

    def addListener(self, listenerFunc):
        self.listeners.append(listenerFunc)

    def removeListener(self, listenerFunc):
        self.listeners.remove(listenerFunc)

    def update(self):
        for eachFunc in self.listeners:
            eachFunc(self)
```

Потребителями в этом случае выступают вызываемые объекты, которые могут принимать в качестве аргумента `self`, а так как фактический класс `self` вероятно может измениться, то Ваш потребитель должен быть гибким. Также, мы установили `AbstractModel` как класс Python нового стиля, что подтверждает тот факт, что он является подклассом `object`. Следовательно, этот пример требует для своей работы Python 2.2 или выше.

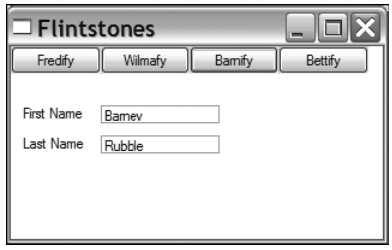

Рисунок 5.4 Пример окна, показывающего как работают модели

Как мы можем использовать абстрактный класс модели? Рисунок 5.4 показывает новое окно, подобное окну, которое мы использовали ранее в этой главе для рефакторинга. Окно

простое. Текстовые поля только для чтения. Щелчок одной из кнопок заставляет отображать в текстовых полях имя соответствующего игрока.

Выполняющая это окно программа использует простую структуру MVC. Методы-обработчики кнопки изменяют модель, а изменение структуры модели заставляет текстовые поля изменяться. Листинг 5.12 это подробно показывает.

Листинг 5.12 MVC-программа, «флинтстонизирующая» («Flintstonize») Ваше окно

```
#!/usr/bin/env python

import wx
import abstractmodel

class SimpleName(abstractmodel.AbstractModel):

    def __init__(self, first="", last=""):
        abstractmodel.AbstractModel.__init__(self)
        self.set(first, last)

    def set(self, first, last):
        self.first = first
        self.last = last
        self.update()

class ModelExample(wx.Frame):

    def __init__(self, parent, id):
        wx.Frame.__init__(self, parent, id, 'Flintstones',
            size=(340, 200))
        panel = wx.Panel(self)
        panel.SetBackgroundColour("White")
        self.Bind(wx.EVT_CLOSE, self.OnCloseWindow)
        self.textFields = {}
        self.createTextFields(panel)
        self.model = SimpleName()
        self.model.addListener(self.OnUpdate)
        self.createButtonBar(panel)

    def buttonData(self):
        return (("Fredify", self.OnFred),
                ("Wilmafy", self.OnWilma),
                ("Barnify", self.OnBarney),
                ("Bettify", self.OnBetty))
    def createButtonBar(self, panel, yPos = 0):
        xPos = 0
        for eachLabel, eachHandler in self.buttonData():
            pos = (xPos, yPos)
        button = self.buildOneButton(panel, eachLabel, eachHandler,
                    pos)
        xPos += button.GetSize().width

    def buildOneButton(self, parent, label, handler, pos=(0,0)):
        button = wx.Button(parent, -1, label, pos)
```

1 **Обновление**

2 **Создание модели**

```
        self.Bind(wx.EVT_BUTTON, handler, button)
        return button

    def textFieldData(self):
        return (("First Name", (10, 50)),
                ("Last Name", (10, 80)))

    def createTextFields(self, panel):
        for eachLabel, eachPos in self.textFieldData():
            self.createCaptionedText(panel, eachLabel, eachPos)

    def createCaptionedText(self, panel, label, pos):
        static = wx.StaticText(panel, wx.NewId(), label, pos)
        static.SetBackgroundColour("White")
        textPos = (pos[0] + 75, pos[1])
        self.textFields[label] = wx.TextCtrl(panel, wx.NewId(),
            "", size=(100, -1), pos=textPos,
            style=wx.TE_READONLY)

    def OnUpdate(self, model):
        self.textFields["First Name"].SetValue(model.first)
        self.textFields["Last Name"].SetValue(model.last)

    def OnFred(self, event):
        self.model.set("Fred", "Flintstone")

    def OnBarney(self, event):
        self.model.set("Barney", "Rubble")

    def OnWilma(self, event):
        self.model.set("Wilma", "Flintstone")

    def OnBetty(self, event):
        self.model.set("Betty", "Rubble")

    def OnCloseWindow(self, event):
        self.Destroy()

if __name__ == '__main__':
    app = wx.PySimpleApp()
    frame = ModelExample(parent=None, id=-1)
    frame.Show()
    app.MainLoop()
```

3 **Установка текстовых полей**

4 **Обработчики нажатия кнопок**

1. Эта строка выполняет обновление
2. Эти две строки создают объект модели и регистрируют в качестве потребителя метод `OnUpdate()`. Теперь этот метод будет вызываться всякий раз, когда потребуется обновление.

3. Метод `OnUpdate()` самостоятельно устанавливает значения текстовых полей, используя для этого модель, которая поступает как часть обновления. Данный код мог бы использовать вместо этого `self.model` (это тот же объект). Использование такого аргумента метода более предпочтительно в том случае, где один и тот же код обслуживает множество объектов.
4. Эти обработчики нажатия кнопок изменяют значение инициирующего обновление объекта модели.

В этом небольшом примере механизм обновления модели может показаться чрезмерно изощренным. Например, здесь не было оснований не давать обработчикам кнопок непосредственно устанавливать значения текстовых полей. Однако показанный механизм модели становится наиболее ценным, когда класс модели имеет более сложное внутреннее состояние и режим работы. Вы можете, например, не делая каких-либо изменений в Представлении, перейти от внутреннего использования словаря Python к внешней базе данных.

Если Вы имеете дело с существующим классом, который не можете или не хотите изменять, тогда почти тем же способом, как и с `LineupTable` в листинге 5.10, для наделения существующего класса полномочиями может быть использован класс `AbstractModel`.

Кроме того, wxPython содержит две отдельных реализации механизмов обновления, аналогичных MVC, которые имеют больше возможностей, чем описанный нами механизм. Первый из них – модуль `wx.lib.pubsub`, практически аналогичный по структуре приведенному ранее классу `AbstractModel`. Это класс модели, названный `Publisher` и позволяющий объектам реагировать только на определенные типы сообщений. Другая система обновления, `wx.lib.evtmgr.eventManager`, построена на основе `pubsub` и имеет несколько дополнительных возможностей, включая более сложный объектно-ориентированный дизайн, легкое связывание и удаление событийных связей.

5.3 Как сделать тест-модуль GUI-программы?

Ключевое достоинство хорошего рефакторинга и проектной модели MVC состоит в том, что они с помощью тестового модуля гарантируют продуктивность Вашей программы. Тестовый модуль представляет собой тест отдельной, специфической функции Вашей программы. Поскольку при рефакторинге и при использовании проектной модели MVC происходит разбиение Вашей программы на небольшие фрагменты, Вам будет проще написать специальный тестовый модуль, нацеленный на отдельные части Вашей программы.

Готовый тестовый модуль позволяет Вам проверить, что в процессе работы с кодом Вы не ввели каких-либо ошибок, поэтому в сочетании с рефакторингом он является особенно полезным инструментом.

Неизбежный вопрос при тестировании модуля состоит в том, как тестировать UI-код. Тестирование модели сравнительно просто, так как в большинстве своем функциональность модели не зависит от ввода пользователя. Тестирование же

собственно функциональности интерфейса может вызвать трудности, поскольку поведение интерфейса зависит от поведения пользователя, которое труднее поддается обобщению. В этом подразделе мы покажем Вам, как использовать в wxPython модуль тестирования, и особенно использование вручную сгенерированных событий для координации поведения при тесте модуля.

5.3.1 Модуль тестирования

Чтобы избавиться от повторной записи кода для выполнения Ваших тестов, при написании пользовательских тестов полезно использовать существующий тестовый механизм (движок). Начиная с версии 2.1, Python поставляется с модулем `unittest`. Модуль `unittest` реализует тестовый каркас, названный `PyUnit` (базирующийся на Tkinter, интерфейс пользователя для `unittest` и некоторые другие полезные вещи доступны по адресу: http://pyunit.sourceforge.net/). Модуль `PyUnit` состоит из одиночных тестов, тест-контейнеров (TestCase) и тест-комплектов (TestSuite). Эти три группы описывает таблица 5.5.

Таблица 5.5 Три уровня абстракции в модуле unittest

Уровень	Определение
Одиночный тест (Test)	Это отдельный метод, вызываемый движком PyUnit. По соглашению об именах, имя метода теста начинается с test. Метод теста обычно исполняет некоторый код, а затем выполняет одну или более контрольных директив (assert statements), чтобы проверить, получены ли ожидаемые результаты.
Тест-контейнер (TestCase)	Это класс, определяет один или более индивидуальных тестов, совместно использующих общую настройку. Данный класс определен в PyUnit для управления группой подобных тестов. Класс `TestCase` имеет средства для выполнения общей настройки как до, так и после каждого теста, гарантируя работу каждого теста независимо от других. Класс `TestCase` также определяет несколько специальных контрольных директив, как например `assertEqual`.
Тест-комплект (TestSuite)	Это один или более тестирующих методов или объектов TestCase, группированных вместе (в комплект) с целью начала одновременного выполнения. Когда Вы фактически заставляете PyUnit запустить тесты, Вы передаете этот комплект на выполнение объекту `TestSuite`.

Одиночный тест PyUnit возвращает один из трех результатов: успех, неудача или ошибка. Успех указывает, что тестовый метод отработал до конца, все контрольные директивы дали положительный (true) результат, и никаких ошибок не было инициировано. Конечно же, это желательный результат. Неудача и ошибка указывают на различные проблемы в коде. Неудачный результат означает, что одна из Ваших контрольных директив вернула false(ложь), указывая, что код

выполняется успешно, но не делает того, что Вы ожидаете. Ошибочный результат означает, что где-то в выполняемом тесте инициировано исключение Python и Ваш код из-за этого не выполняется успешно. Первая неудача или ошибка в одиночном тесте закончит выполнение этого теста, даже если код содержит другие контрольные директивы, а исполнитель (runner) теста перейдет к следующему тесту.

5.3.2 Пример модуля unittest

Листинг 5.13 показывает пример модуля `unittest`, в этом случае тестируется образец модели из листинга 5.12.

Листинг 5.13 Пример тестового модуля для образцовой модели

```
import unittest
import modelExample
import wx

class TestExample(unittest.TestCase):

    def setUp(self):
        self.app = wx.PySimpleApp()
        self.frame = modelExample.ModelExample(parent=None, id=-1)

    def tearDown(self):
        self.frame.Destroy()

    def testModel(self):
        self.frame.OnBarney(None)
        self.assertEqual("Barney", self.frame.model.first,
            msg="First is wrong")
        self.assertEqual("Rubble", self.frame.model.last)

def suite():
    suite = unittest.makeSuite(TestExample, 'test')
    return suite

if __name__ == '__main__':
    unittest.main(defaultTest='suite')
```

1 Объявление тест-контейнера

2 Настройка каждого теста

3 Завершение каждого теста

4 Объявление теста

5 Контрольная директива, которая может потерпеть неудачу

6 Создание тест-комплекта

7 Запуск теста

1. Данный тест-контейнер является подклассом `unittest.TestCase`. Исполнитель теста создает экземпляр класса для каждого из тестов, что делает их более независимыми друг от друга.
2. Метод `setUp()` вызывается перед каждым запускаемым на выполнение тестом. Это гарантирует, что запуск каждого теста начинается с одного и того же состояния Вашего приложения. В данном случае, мы создаем экземпляр фрейма, который мы тестируем.

3. Метод `tearDown()` вызывается после выполнения каждого теста. Это позволяет Вам провести любую необходимую очистку, гарантирующую, что от теста к тесту состояние системы останется согласованным. Обычно это включает сброс глобальных данных, закрытие соединений с базой данных и т.п. В нашем случае, мы вызываем для фрейма `Destroy()`, который заставляет wxWidgets завершиться и сохраняет систему в подходящем для следующего теста состоянии.
4. Имя тестового метода обычно начинается с префикса `test`, хотя это и определяется на Ваше усмотрение (смотрите строку 6). Тестовые методы не принимают никаких аргументов. Этот метод начинается с прямого вызова обработчика события `OnBarney`, чтобы протестировать его выполнение.
5. Эта строка использует метод `assertEqual()` для тестирования того, что объект модели изменен корректно. Метод `assertEqual()` принимает два аргумента и тест терпит неудачу, если они не равны. Все контрольные директивы PyUnit принимают необязательный аргумент `msg`, который отображенается, если контрольная директива терпит неудачу (установки по умолчанию для `assertEqual()` почти всегда достаточно полезны).
6. Этот метод создает тестовый комплект посредством удобного механизма - метода `makeSuite()`. Этот метод принимает в качестве аргументов объект класса Python и строку префикса, а возвращает тестовый комплект, содержащий все тестовые методы в этом классе, чьи имена начинают с указанного префикса. Имеются и другие механизмы, учитывающие более явную установку содержания тестового комплекта, но представленного метода обычно Вам будет достаточно. Метод `suite()` – это стандартный шаблон, который может быть использован во всех Ваших тестовых модулях.
7. Эта строка запускает ориентированный на обработку текста исполнитель PyUnit. Аргумент является именем метода, который возвращает тестовый комплект. Затем этот комплект запускается и результаты выводятся на консоль. Если Вы хотите использовать исполнитель теста с графическим интерфейсом, Вам необходимо изменить эту строку и вызвать основной метод этого модуля.

Результаты этого выполняемого в консольном окне теста PyUnit выглядят следующим образом:

```
.
----------------------------------------------------------------------
Ran 1 test in 0.190s
OK
```

Это - успешный тест. Верхняя строка с точкой указывает, что успешно отработал один тест. Каждый тест образует на экране один символ, причем «.» означает успех, «F» указывает на аварийную ситуацию, а «E» указывает на ошибку. Затем приводится простой листинг количества тестов и полное время выполнения, OK указывает, что выполнены все тесты.

В случае аварии или ошибки Вы получаете стек вызовов, показывающий, как Python дошел до проблемной точки. Например, если мы изменим последнюю проверку имени на `Fife`, то должны получить следующий результат:

```
F
======================================================================
FAIL: testModel (__main__.TestExample)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\wxPyBook\book\1\Blueprint\testExample.py", line 18, in
testModel
    self.assertEqual("Fife", self.frame.model.last)
  File "c:\python23\lib\unittest.py", line 302, in
failUnlessEqual
    raise self.failureException, \
AssertionError: 'Fife' != 'Rubble'

----------------------------------------------------------------------
Ran 1 test in 0.070s

FAILED (failures=1)
```

Здесь первая строка говорит об аварии, выдает имя метода, в котором она произошла, трассировка ошибки (traceback) показывает, что к аварии привела контрольная проверка в строке 18 и то, чем она вызвана. Обычно, чтобы определить, где фактическая произошла авария, Вам нужно пройти на несколько уровней вниз по трассе стека; последняя одна-две строки, вероятно, будут относиться собственно к модулю `unittest`.

5.3.3 Тестирование событий пользователя

Конечно, этот тест - не полный тест системы. Мы могли бы также протестировать, что вслед за обновлением модели, в фрейме были обновлены значениями экземпляры TextField. Такой тест должен быть достаточно простым. Другой тест, который Вам может понадобиться, должен автоматически сгенерировать само событие нажатия кнопки, и проверить, что вызывается соответствующий обработчик. Этот тест немного сложнее. Листинг 5.14 показывает такой пример:

Листинг 5.14 Пример теста на основе генерации события пользователя

```
def testEvent(self):
    panel = self.frame.GetChildren()[0]
    for each in panel.GetChildren():
        if each.GetLabel() == "Wilmafy":
            wilma = each
            break
    event = wx.CommandEvent(wx.wxEVT_COMMAND_BUTTON_CLICKED,
            wilma.GetId())
```

```
        wilma.GetEventHandler().ProcessEvent(event)
        self.assertEqual("Wilma", self.frame.model.first)
        self.assertEqual("Flintstone", self.frame.model.last)
```

Первые несколько строк этого примера находят соответствующую кнопку (в данном случае, кнопку "Wilmafy"). Так как мы не храним кнопки явно как экземпляры переменных, нам просто нужно пройти по списку дочерних элементов панели, пока мы не найдем нужную кнопку. (При желании, Вы можете это сделать также с помощью списка Python). Следующие две строки создают `wx.CommandEvent` для отправки из кнопки. Параметр `wx.wxEVT_COMMAND_BUTTON_CLICKED` представляет собой необходимую объекту `EVT_BUTTON` целочисленную константу типа события. (Вы можете найти целочисленные константы в исходном файле wxPython wx.py). Затем, мы установили ID события на ID кнопки Wilmafy. В этой точке, данное событие имеет все необходимые характеристики фактического события, каким его создает wxPython. Теперь мы вызываем `ProcessEvent()`, чтобы послать это в систему. Если этот код отработает, как запланировано, имя и фамилия модели будут изменены на "Wilma" и "Flintstone".

С помощью генерации событий Вы можете от начала и до конца протестировать ответную реакцию Вашей системы. Теоретически, для того, чтобы проверить, что в качестве ответной реакции создается событие нажатия кнопки, Вы могли бы вызвать события нажатия и отпускания кнопки мыши внутри Вашей кнопки. На практике, со стандартными виджетами это работать не будет, так как им передаются низкоуровневые события `wx.Events`, не переведенные назад в стандартные системные события. Тем не менее, подобный процесс может быть полезным при тестировании пользовательских виджетов (как например, элемент управления с двумя кнопками из главы 3). Такой способ тестирования модулей подтвердит ответную реакцию Вашего приложения.

5.4 Резюме

- В связи с неупорядоченностью и трудностями в обслуживании код GUI имеет плохую репутацию. Это может быть преодолено незначительными дополнительными усилиями, которые оплатятся сполна, когда придет пора вносить в Ваш код изменения.
- Рефакторинг улучшает существующий код. Основными задачами рефакторинга являются: устранение дублирования, устранение «магических» литералов и создание коротких, делающих что-то одно, методов. Непрерывное стремление этим целям сделают Вашим кодом легче для чтения и понимания. Кроме того, хороший рефакторинг значительно снижает вероятность появления определенных типов ошибок (например, ошибок вырезания-вставки).

- Отделение Ваших данных от проектного кода облегчает работу с этими данными и самим проектом в целом. Стандартным механизмом для управления таким разделением выступает механизм MVC. В терминах wxPython, *Представление* – это объект `wx.Window`, который отображает Ваши данные, *Контроллер* является объектом `wx.EvtHandler`, транслирующим события, а *Модель* представляет собственно Ваш код, который содержит необходимую для отображения информацию.

- Возможно, самый яркий пример структуры MVC в основных классах wxPython – это `wx.grid.PyGridTableBase`, использующийся для моделирования данных при отображении в элементе управления `wx.grid.Grid`. Данные для таблицы могут или поставляться из самого класса, или класс может ссылаться на другой содержащий соответствующие данные объект.

- Вы можете создать собственную настройку MVC с простым механизмом уведомления представления при обновлении модели. В wxPython существуют модули, которое помогут Вам это сделать.

- Модульное тестирование является полезным способом контроля правильности Вашей программы. Модуль Python `unittest` - один из стандартных путей выполнения модульных тестов. В некоторых пакетах тестирование GUI затруднено, но wxPython при помощи программного создания событий делает это сравнительно легко. Это позволяет Вам от начала и до конца протестировать поведение Вашего приложения при обработке событий.

В следующей главе мы покажем Вам, как построить небольшое приложение, и как сделать другие вещи, которые станут общими для многих создаваемых Вами приложений wxPython.